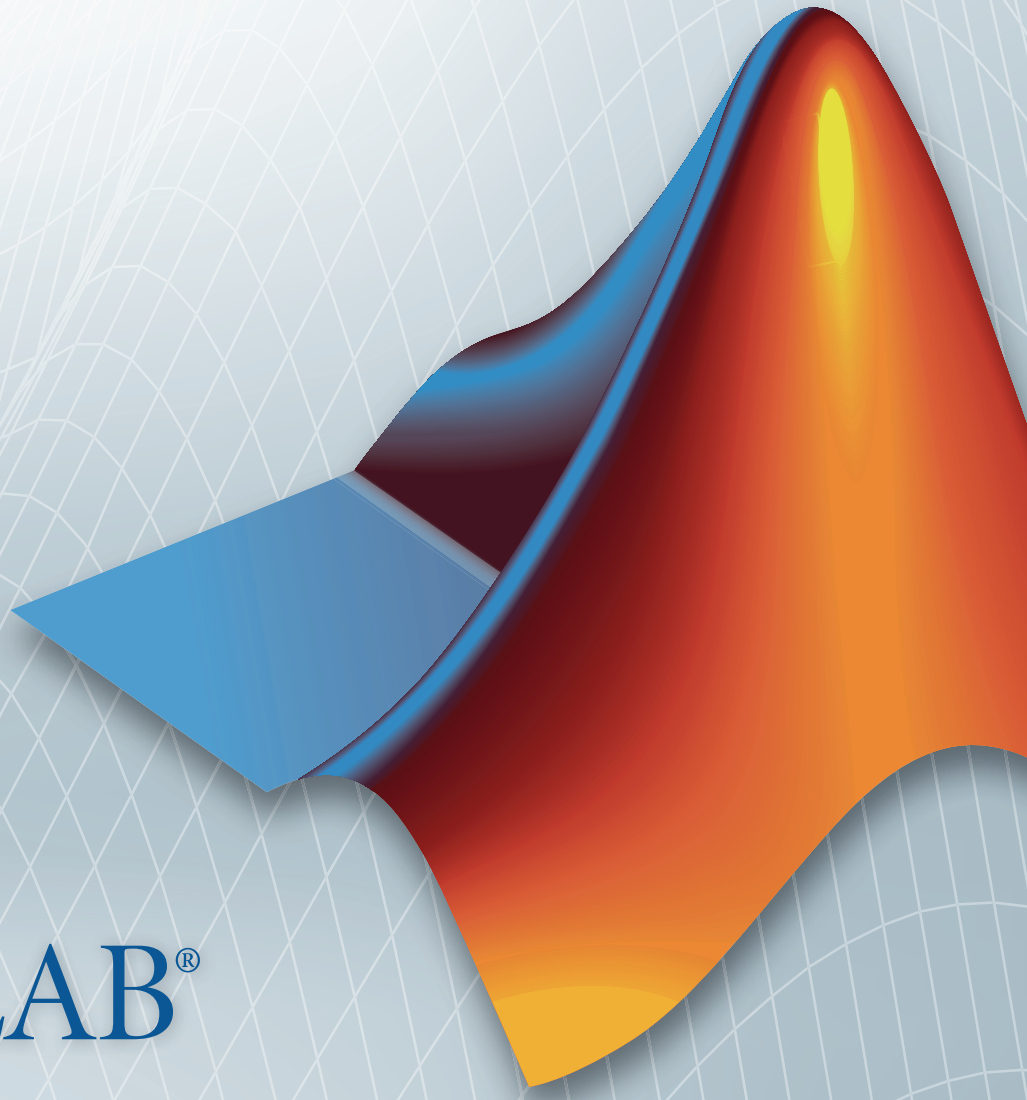
Symbolic Math Toolbox™ 5

MuPAD® Tutorial



MATLAB®

MathWorks®

How to Contact MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Symbolic Math Toolbox™ MuPADÈ Tutorial

Trademarks

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

Patents

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

Preface

This book explains the basic use of the MuPAD® computer algebra system, and gives some insight into the power of the system. MuPAD is available as part of the Symbolic Math Toolbox™ in MATLAB®.

The book does not include the information about the latest features available in MuPAD. The updated documentation is available in the MuPAD Help Browser. To open the Help Browser, start MuPAD and click the Open Help button on the desktop toolbar or use the Help menu. Also see the Release Notes section in the MuPAD Help Browser for the short summary of the new features, bug fixes, and upgrade issues.

This introduction addresses mathematicians, engineers, computer scientists, natural scientists and, more generally, all those in need of mathematical computations for their education or their profession. Generally speaking, this book addresses anybody who wants to use the power of a modern computer algebra package.

There are two ways to use a computer algebra system. On the one hand, you may use the mathematical knowledge it incorporates by calling system functions interactively. For example, you can compute symbolic integrals or generate and invert matrices by calling appropriate functions. They comprise the system's mathematical intelligence and may implement sophisticated algorithms. Chapters 2 through 14 discuss this way of using the MuPAD engine.

On the other hand, with the help of the MuPAD programming language, you can easily add functionality to the system by implementing your own algorithms as MuPAD procedures. This is useful for special purpose applications if no appropriate system functions exist. Chapters 15 through 17 are an introduction to MuPAD programming.

You can read this book in the standard way "linearly" from the first to the last page. However, there are reasons to proceed otherwise. This may be the case, e.g., if you are interested in a particular problem, or if you already know something about MuPAD.

For beginners, we recommend to start reading Chapter 2, which gives a first survey of MuPAD. The description of the online help system in Section 2.2 is probably the most important part of this chapter. The help system provides information about details of system functions, their syntax, their calling

parameters, etc. It is available online whenever the MuPAD notebook interface is running. In the beginning, requesting a help page is probably your most frequent query to the system. After you have grown familiar with the help system, you may start to experiment with MuPAD. Chapter 2 demonstrates some of the most important system functions "at work." You will find further details about these functions in later parts of the book or in the help pages. For a deeper understanding of the data structures involved, you may consult the corresponding sections in Chapter 4.

Chapter 3 discusses the MuPAD libraries and their use. They contain many functions and algorithms for particular mathematical topics.

The basic data types and the most important system functions for their manipulation are introduced in Chapter 4. It is not necessary to study all of them in the same depth. Depending on your intended application, you may selectively read only the passages about the relevant topics.

Chapter 5 explains how MuPAD evaluates objects; we strongly recommend to read this chapter.

Chapters 6 through 11 discuss the use of some particularly important system functions: substitution, differentiation, symbolic integration, equation solving, random number generation, and graphic commands.

Several useful features such as the history mechanism, input and output routines, or the definition of user preferences are described in Chapters 13.2 through 13. Preferences can be used to configure the system's interactive behavior after the user's fancy to a certain extent.

Chapters 15 through 17 give an introduction to the basic concepts of the MuPAD programming language.

MuPAD provides algorithms that can handle a large class of mathematical objects and computational tasks related to them. Upon reading this introduction, it is possible that you encounter unknown mathematical notions such as rings or fields. This introduction is not intended to explain the mathematical background for such objects. Basic mathematical knowledge is helpful but not mandatory to understand the text. Sometimes you may ask what algorithm MuPAD uses to solve a particular problem. The internal mode of operation of the MuPAD procedures is not addressed here: we do not intend to give a general introduction to computer algebra and its algorithms. The interested reader may consult text

books such as, e.g., [GCL 92] or [GG 99].

This book gives an *elementary* introduction to MuPAD. Somewhat more abstract mathematical objects such as, e.g., field extensions, are easy to describe and to handle in MuPAD. However, such advanced aspects of the system are not discussed here. The mathematical applications that are mentioned in the text are intentionally kept on a rather elementary level. This is to keep this text plain for readers with little mathematical background and to make it applicable at school level.

We cannot explain the complete functionality of MuPAD in this introduction. Some parts of the system are mentioned only briefly. It is beyond the scope of this tutorial to go into the details of the full power of the MuPAD programming language. You find these in the MuPAD help system, available online during a MuPAD session.

This tutorial refers to MuPAD version 5 and later. Since the development of the system advances continuously, some of the details described may change in the future. Future versions will definitely provide additional functionality through new system functions and application packages. In this tutorial, we mainly present the basic tools and their use, which will probably remain essentially unchanged. We try to word all statements in the text in such a way that they stay basically valid for future MuPAD versions.

Contents

3 The MuPAD® Libraries

4 MuPAD® Objects

9 Manipulating Expressions

10 Chance and Probability

11 Graphics

1

Introduction

To explain the notion of computer algebra, we compare algebraic and numerical computations. Both kinds are supported by a computer, but there are fundamental differences, which are discussed in what follows.

Numerical Computations

Many a mathematical problem can be solved approximately by numerical computations. The computation steps operate on *numbers*, which are stored internally in *floating-point representation*. This representation has the drawback that neither computations nor solutions are exact due to rounding errors. In general, numerical algorithms find approximate solutions as fast as possible. Often such solutions are the only way to handle a mathematical problem computationally, in particular if there is no "closed form" solution known. (The most popular example for this situation are roots of polynomials of high degrees.) Moreover, approximate solutions are useful if exact results are unnecessary (e.g., in visualization).

Computer Algebra

In contrast to numerical computations, there are *symbolic* computations in computer algebra. [Hec 93] defines them as *"computations with symbols representing mathematical objects."* Here, an *object* may be a number, but also a polynomial, an equation, an expression or a formula, a function, a group, a ring, or any other mathematical object. Symbolic computations with numbers are carried out *exactly*. Internally, numbers are represented as quotients of integers of arbitrary length (limited by the amount of storage available, of course). These kinds of computations are called *symbolic* or *algebraic*. [Hec 93] gives the following definitions:

1. "Symbolic" emphasizes that in many cases the ultimate goal of mathematical problem solving is expressing the answer in a closed formula or finding a symbolic approximation.
2. "Algebraic" means that computations are carried out exactly, according to the rules of algebra, instead of using approximate floating-point arithmetic.

Sometimes "symbolic manipulation" or "formula manipulation" are used as synonyms for computer algebra, since computations operate on symbols and formulae. Examples are symbolic integration or differentiation such as

$$\int x\,\mathrm{d}x = \frac{x^2}{2}, \quad \int_1^4 x\,\mathrm{d}x = \frac{15}{2}, \quad \frac{\mathrm{d}}{\mathrm{d}x}\ln\ln x = \frac{1}{x\ln x}$$

or symbolic solutions of equations. For example, we consider the equation $x^4 + p\,x^2 + 1 = 0$ in x with one parameter p. Its solution set is

$$\left\{ \pm\frac{\sqrt{2}\,\sqrt{-p-\sqrt{p^2-4}}}{2},\ \pm\frac{\sqrt{2}\,\sqrt{-p+\sqrt{p^2-4}}}{2} \right\}.$$

The symbolic computation of an exact solution usually requires more computing time and more storage than the numeric computation of an approximate solution. However, a symbolic solution is exact, more general, and often provides more information about the problem and its solution. The above formula expresses the solutions of the equation in terms of the parameter p. It shows how the solutions

depend functionally on p. This information can be used, for example, to examine how sensitive the solutions behave when the parameter changes.

Combinations of symbolic and numeric methods are useful for special applications. For example, there are algorithms in computer algebra that benefit from efficient hardware floating-point arithmetic. On the other hand, it may be useful to simplify a problem from numerical analysis symbolically before applying the actual approximation algorithm.

Characteristics of Computer Algebra Systems

Most of the current computer algebra systems can be used interactively. The user enters some formulae and commands, and the system *evaluates* them. Then it returns an answer, which can be manipulated further if necessary.

In addition to exact symbolic computations, most computer algebra packages can approximate solutions numerically as well. The user can set the precision to the desired number of digits. In MuPAD®, the global variable `DIGITS` handles this. For example, if you enter the simple command `DIGITS:=100`, then MuPAD performs all floating-point calculations with a precision of 100 decimal digits. Of course, such computations need more computing time and more storage than the use of hardware floating-point arithmetic.

Moreover, modern computer algebra systems provide a powerful programming language[1] and tools for visualization and animation of mathematical data. Also, many systems can produce layouted documents (known as *notebooks* or *worksheets*). The MuPAD system has such a notebook concept, but we will only sketch it briefly in this tutorial. Please check the user interface documentation instead for more details. The goal of this book is to give an introduction to the *mathematical* power of the MuPAD language.

[1] The MuPAD programming language is structured similarly to Pascal, with extensions such as language constructs for object oriented programming.

MuPAD® Software

In comparison to other computer algebra systems, MuPAD® has the following noteworthy features, which are not discussed in detail in this book:

- There are MuPAD language constructs for object oriented programming. You can define your own data types, e.g., to represent mathematical structures. Almost all existing operators and functions can be overloaded.
- There is a highly interactive MuPAD source code debugger.
- Programs written in C or C++ can be added to the kernel by the MuPAD dynamic module concept.

The heart of the MuPAD engine is its *kernel*, which is implemented in C++. It comprises the following main components:

– The *parser* reads the input to the system and performs a syntax check. If no errors are found, it converts the input to a MuPAD data type.

– The *evaluator* processes and simplifies the input. Its mode of operation is discussed later.

– The *memory management* is responsible for the efficient storage of MuPAD objects.

– Some frequently used algorithms such as, e.g., arithmetical functions are implemented as *kernel functions* in C.

Parser and evaluator define the MuPAD language as described in this book. The MuPAD libraries, which contain most of the mathematical knowledge of the system, are implemented in this language.

In addition, MuPAD offers comfortable user interfaces for generating *notebooks* or graphics, or to debug programs written in the MuPAD language. The MuPAD help system has hypertext functionality. You can navigate within documents and execute examples by a mouse click.

Symbolic Math Toolbox™ also offers a way of using the MuPAD engine to solve problems directly from the MATLAB® environment, using either commands as

presented here or, for a selected subset of commands, alternatives fitting more closely into a MATLAB language program. This book does not discuss these options, but limits itself to using the MuPAD engine directly, along with the MuPAD GUIs.

2

First Steps in MuPAD®

Computer algebra systems such as MuPAD® are often used interactively. For example, you can enter an instruction to multiply two numbers and have MuPAD compute the result and print it on the screen.

The following section will give a short description of the user interface. The next one after that describes the help system. Requesting a help page is probably the most frequently used command for the beginner. The section after that is about using MuPAD as an “intelligent pocket calculator”: calculating with numbers. This is the easiest and the most intuitive part of this tutorial. Afterwards we introduce some system functions for symbolic computations. The corresponding section is written quite informally and gives a first insight into the symbolic features of the system.

Notebook interface

After you call the MuPAD® program, you will see the welcome dialog. From there, you can enter the help system (which is, of course, also available later and will be described on page 2-4), open an existing file, create a new MuPAD source file or open a new notebook. Usually, you will wish to open an existing or a new notebook.

After starting the program, you can enter commands in the MuPAD language. These commands should be typed into “input regions,” which look like this:

If you press the <Return> key, this finishes your input and MuPAD evaluates the command that you have entered. Holding <Shift> while pressing <Return> provokes a linefeed, which you can use to format your input.

After invoking the command, you will see its result printed in the same bracket, below the input, and if you were at the end of the notebook, a new input region will be opened below:

```
sin(3.141)
```

$$0.0005926535551$$

The system evaluated the usual sine function at the point 3.141 and returned a floating-point approximation of the value, similar to the output of a pocket calculator.

You can at any time go back to older inputs and edit them. Also, you can click between the brackets and add text. Editing output is not possible, but you can copy formulas from the output and paste them someplace else. (When copying a formula to an input region, it will be automatically translated into an equivalent command in textual form.)

If you terminate your command with a colon, then MuPAD executes the command without printing its result on the screen. This enables you to suppress the output of irrelevant intermediate results.

You can enter more than one command in one line. Two subsequent commands have to be separated by a semicolon or a colon, if the result of the first command is to be printed or not, respectively:

```
diff(sin(x^2), x); int(%, x)
```

$$2\,x\,\cos\left(x^2\right)$$

$$\sin\left(x^2\right)$$

Here `x^2` denotes the square of `x`, and the MuPAD functions `diff` and `int` perform the operations "differentiate" and "integrate" (Chapter 7). The character `%` returns the previous expression (in the example, this is the derivative of $\sin(x^2)$). The concept underlying `%` is discussed in Chapter 13.2.

In the following example, the output of the first command is suppressed by the colon, and only the result of the second command appears on the screen:

```
equations := {x + y = 1, x - y = 1}:
```

```
solve(equations)
```

$$\{[x = 1, y = 0]\}$$

In the previous example, a set of two equations is assigned to the identifier `equations`. The command `solve(equations)` computes the solution. Chapter 8 discusses the solver in more detail.

To quit a MuPAD session, use the corresponding entry in the "File" menu.

Explanations and Help

If you do not know the correct syntax of a MuPAD® command, you can obtain this information directly from the online help system. For a brief explanation, simply type the command name, point your mouse cursor at the command and let it rest there for a moment. You will get a tooltip with a brief explanation.

The *help page* of the corresponding function provides more detailed information. You can request it by entering `help("name")`, where `name` is the name of the function. The function `help` expects its argument to be a string, which is generated by double quotes `"` (Section 4.11). The operator `?` is a short form for `help`. It is used without parenthesis or quotes:

```
?solve
```

The help system is a hypertext system, i.e., similar to browsing the web. Active keywords are underlined. If you click on them, you obtain further information about the corresponding notion. The tooltips mentioned above also work in the help system. You can edit and execute the examples in the help system or copy and paste them to a notebook.

Exercise 2.1: Find out how to use the MuPAD differentiator `diff`, and compute the fifth derivative of $\sin(x^2)$.

Computing with Numbers

To compute with numbers, you can use MuPAD® like a pocket calculator. The result of the following input is a rational number:

```
1 + 5/2
```

$$\frac{7}{2}$$

You see that the returns are exact results (and not rounded floating-point numbers) when computing with integers and rational numbers:

```
(1 + (5/2*3))/(1/7 + 7/9)^2
```

$$\frac{67473}{6728}$$

The symbol ^ represents exponentiation. MuPAD can compute big numbers efficiently. The length of a number that you may compute is only limited by the available main storage. For example, the 123rd power of 1234 is a fairly big integer:[1]

```
1234^123
```

```
  17051580621272704287505972762062628265430231311106829\
  04705296193221839138348680074713663067170605985726415\
  92314554345900570589670671499709086102539904846514793\
  13561730556366999395010462203568202735575775507008323\
  84441477783960263870670426857004040032870424806396806\
  96865587865016699383883388831980459159942845372414601\
  80942971772610762859524340680101441852976627983806720\
  3562799104
```

Besides the basic arithmetic functions, MuPAD provides a variety of functions operating on numbers.

[1] In this printout, the "backslash" \ at the end of a line indicates that the result is continued on the next line. To break long lines in this way, you need to switch off typesetting in the "Notebook" menu.

A simple example is the factorial $n! = 1 \cdot 2 \cdots n$ of a nonnegative integer, which can be entered in mathematical notation:

```
100!
```

$$933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536946859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000000$$

The function `isprime` checks whether a positive integer is prime. It returns either `TRUE` or `FALSE`.

```
isprime(123456789)
```

FALSE

Using `ifactor`, you can obtain the prime factorization:

```
ifactor(123456789)
```

$$3^2 \cdot 3607 \cdot 3803$$

Exact Computations

Now suppose that we want to "compute" the number $\sqrt{56}$. The problem is that the value of this irrational number cannot be expressed as a quotient numerator/denominator of two integers exactly. Thus "computation" can only mean to find an exact representation that is *as simple as possible*. When you input $\sqrt{56}$ via `sqrt`, MuPAD returns the following:

```
sqrt(56)
```

$$2\sqrt{14}$$

The result of the simplification of $\sqrt{56}$ is the exact value $2 \cdot \sqrt{14}$. In MuPAD, $\sqrt{14}$ (or, sometimes, `14^(1/2)`) represents the positive solution of the equation $x^2 = 14$. Indeed, this is probably the most simple representation of the result. We stress that $\sqrt{14}$ is a genuine object with certain properties (e.g., that its square can be simplified to 14). The system applies them automatically when computing with such objects. For example:

```
sqrt(14)^4
```

$$196$$

As another example for exact computation, let us determine the limit

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n.$$

We use the function `limit` and the symbol `infinity`:

```
limit((1 + 1/n)^n, n = infinity)
```

$$e$$

To enter this number in MuPAD, you have to use `E` or `exp(1)`, where `exp` represents the exponential function. MuPAD knows exact rules of manipulation for this object. For example, using the natural logarithm `ln` we find:

```
ln(1/exp(1))
```

$$-1$$

We will encounter more exact computations later in this tutorial.

Numerical Approximations

Besides exact computations, MuPAD can also perform numerical approximations. For example, you can use the function `float` to find a decimal approximation to $\sqrt{56}$. This function computes the value of its argument in *floating-point representation*:

```
float(sqrt(56))
```

$$7.483314774$$

The precision of the approximation depends on the value of the global variable `DIGITS`, which determines the number of decimal digits for numerical computations. Its default value is 10:

```
DIGITS; float(67473/6728)
```

$$10$$

$$10.02868609$$

Global variables such as DIGITS affect the behavior of MuPAD, and are also called *environment variables*.[2] You find a complete list of all environment variables in Section "Environment Variables" of the MuPAD Quick Reference in the online documentation. The variable DIGITS can assume any integral value between 1 and $2^{32} - 1$, although 1000 can already be considered very large and it is unlikely that using more than $10\,000$ digits is reasonable for anything but the most unusual calculations:

```
DIGITS := 100: float(67473/6728); DIGITS := 10:
```

$$10.028686087990487514863258026159334126040428061831153388822829964328180737217598097502972651605231 87$$

We have reset the value of DIGITS to 10 for the following computations. This can also be achieved via the command `delete DIGITS`. For arithmetic operations with numbers, MuPAD automatically uses approximate computation as soon as *at least one* of the numbers involved is a floating-point value:

```
(1.0 + (5/2*3))/(1/7 + 7/9)^2
```

$$10.02868609$$

Please note that none of the two following calls

```
2/3*sin(2), 0.6666666666*sin(2)
```

results in an approximate computation of $\sin(2)$, since, technically, `sin(2)` is an expression representing the (exact) value of $\sin(2)$ and *not* a number:

```
2/3*sin(2), 0.6666666666*sin(2)
```

$$\frac{2\,\sin(2)}{3},\ 0.6666666666\,\sin(2)$$

(The separation of both values by a comma generates a special data type, namely a *sequence*, which is described in Section 4.5.)

[2] You should be particularly cautious when the same computation is performed with different values of DIGITS. Some of the more intricate numerical algorithms in MuPAD employ the option "remember." This implies that they store previously computed values to be used again (Section 17.9), which can lead to inaccurate numerical results if the remembered values were computed with lower precision. To be safe, you should restart the MuPAD session using `reset()` before increasing the value of DIGITS. This command clears memory and resets all environment variables to their default values (Section 13.4).

You have to use the function `float` to compute floating-point approximations of the above expressions:[3]

```
float(2/3*sin(2)), 0.6666666666*float(sin(2))
```

$$0.6061982846,\ 0.6061982845$$

Most arithmetic functions,such as the square root, the trigonometric functions, the exponential function, or the logarithm, automatically return approximate values when their argument is a floating-point number:

```
sqrt(56.0), sin(3.14)
```

$$7.483314774,\ 0.001592652916$$

The constants π and e are denoted by `PI` and `E = exp(1)`, respectively. MuPAD can perform exact computations with them:

```
cos(PI), ln(E)
```

$$-1,\ 1$$

If desired, you can obtain numerical approximations of these constants by applying `float`:

```
DIGITS := 100: float(PI); float(E); delete DIGITS:
  3.14159265358979323846264338327950288419716939937510\
  58209749445923078164062862089986280348253421170 68

  2.71828182845904523536028747135266249775724709369995\
  95749669676277240766303535475945713821785251664 27
```

Exercise 2.2: Compute $\sqrt{27} - 2\sqrt{3}$ and $\cos(\pi/8)$ exactly. Determine numerical approximations to a precision of 5 digits.

Complex Numbers

The imaginary unit $\sqrt{-1}$ is represented in MuPAD by the symbol `I` in the input and an upright i in the typeset output:

[3] Take a look at the last digits. The second command yields a slightly less accurate result, since $0.666\ldots$ is already an approximation of $2/3$ and the rounding error is propagated to the final result.

```
sqrt(-1), I^2
```

$$\mathrm{i},\ -1$$

You can input complex numbers in the usual mathematical notation $x + y\,\mathrm{i}$. Both the real part x and the imaginary part y may be integers, rational numbers, or floating-point numbers:

```
(1 + 2*I)*(4 + I), (1/2 + I)*(0.1 + I/2)^3
```

$$2 + 9\,\mathrm{i},\ 0.073 - 0.129\,\mathrm{i}$$

If you use symbolic expressions such as, e.g., `sqrt(2)`, MuPAD may not return the result of a calculation in Cartesian coordinates:

```
1/(sqrt(2) + I)
```

$$\frac{1}{\sqrt{2} + \mathrm{i}}$$

The function `rectform` (short for: rectangular form) ensures that the result is split into its real and imaginary parts:

```
rectform(1/(sqrt(2) + I))
```

$$\frac{\sqrt{2}}{3} - \frac{\mathrm{i}}{3}$$

The functions `Re` and `Im` return the real part x and the imaginary part y, respectively, of a complex number $x + y\,\mathrm{i}$. The functions `conjugate`, `abs`, and `arg` compute the complex conjugate $x - y\,\mathrm{i}$, the absolute value $|x + y\,\mathrm{i}| = \sqrt{x^2 + y^2}$, and the polar angle, respectively:

```
Re(1/(sqrt(2) + I)), Im(1/(sqrt(2) + I)),
conjugate(1/(sqrt(2) + I)),
abs(1/(sqrt(2) + I)), arg(1/(sqrt(2) + I)),
rectform(conjugate(1/(sqrt(2) + I)))
```

$$\frac{\sqrt{2}}{3},\ -\frac{1}{3},\ \frac{1}{\sqrt{2} - \mathrm{i}},\ \frac{\sqrt{3}}{3},\ -\arctan\left(\frac{\sqrt{2}}{2}\right),\ \frac{\sqrt{2}}{3} + \frac{\mathrm{i}}{3}$$

Symbolic Computation

This section comprises some examples of MuPAD® sessions that illustrate a small selection of the system's power for symbolic manipulation. The mathematical knowledge is contained essentially in the MuPAD functions for differentiation, integration, simplification of expressions etc. This demonstration does not proceed in a particularly systematic manner: we apply the system functions to objects of various types, such as sequences, sets, lists, expressions etc. They are explained in detail one by one in Chapter 4.

Introductory Examples

A symbolic expression may contain undetermined quantities (identifiers). The following expression contains two unknowns x and y:

```
f := y^2 + 4*x + 6*x^2 + 4*x^3 + x^4
```

$$x^4 + 4\,x^3 + 6\,x^2 + 4\,x + y^2$$

Using the assignment operator `:=`, we have assigned the expression to an identifier `f`, which can now be used as an abbreviation for the expression. We say that the latter is the *value* of the identifier `f`. We note that MuPAD has exchanged the order of the terms.[4]

MuPAD offers the system function `diff` for differentiating expressions:

```
diff(f, x), diff(f, y)
```

$$4\,x^3 + 12\,x^2 + 12\,x + 4,\ 2\,y$$

Here, we have computed both the derivative with respect to x and to y. You may obtain higher derivatives either by nested calls of `diff`, or by a single call:

```
diff(diff(diff(f, x), x), x) = diff(f, x, x, x)
```

$$24\,x + 24 = 24\,x + 24$$

[4] Internally, symbolic sums are ordered according to certain rules that enable the system to access the terms faster. Of course, such a reordering of the input happens only for commutative operations such as, e.g., addition or multiplication, where changing the order of the operands yields a mathematically equivalent object.

Alternatively, you can use the differential operator `'`, which maps a function to its derivative:

```
sin', sin'(x)
```

$$\cos,\ \cos(x)$$

The symbol `'` for the derivative is a short form of the differential operator `D`. The call `D(function)` returns the derivative:

```
D(sin), D(sin)(x)
```

$$\cos,\ \cos(x)$$

Note: MuPAD uses a mathematically strict notation for the differential operator: `D` (or, equivalently, the `'` operator) differentiates functions, while `diff` differentiates expressions. In the example, the `'` maps the (name of the) function to the (name of the) function representing the derivative. You often find a sloppy notation such as, e.g., $(x + x^2)'$ for the derivative of the function $F : x \mapsto x + x^2$. This notation confuses the map F and the image point $f = F(x)$ at a point x. MuPAD has a strict distinction between the *function* F and the *expression* $f = F(x)$, which are realized as different data types. The map corresponding to f can be defined by

```
F := x -> x + x^2:
```

Then

```
diff(F(x), x) = F'(x)
```

$$2\,x + 1 = 2\,x + 1$$

are equivalent ways of obtaining the derivative as expressions. The call `f:=x+x^2; f';` does not make sense in MuPAD.

You can compute integrals by using `int`. The following command computes a definite integral on the real interval between 0 and 1:

```
int(f, x = 0..1)
```

$$y^2 + \frac{26}{5}$$

The next command determines an indefinite integral and returns an expression containing the integration variable x and a symbolic parameter y:

```
int(f, x)
```

$$\frac{x^5}{5} + x^4 + 2\,x^3 + 2\,x^2 + x\,y^2$$

Note that `int` returns a special antiderivative, not a general one (with additive constant).

If you try to compute the indefinite integral of an expression and it cannot be represented by elementary functions, then `int` returns the call symbolically:

```
integral := int(1/(exp(x^2) + 1), x)
```

$$\int \frac{1}{\mathrm{e}^{x^2} + 1}\,\mathrm{d}x$$

Nevertheless, this object has mathematical properties. The differentiator recognizes that its derivative is the integrand:

```
diff(integral, x)
```

$$\frac{1}{\mathrm{e}^{x^2} + 1}$$

Definite integrals may also be returned symbolically by `int`:

```
int(1/(exp(x^2) + 1), x = 0..1)
```

$$\int_0^1 \frac{1}{\mathrm{e}^{x^2} + 1}\,\mathrm{d}x$$

The corresponding mathematical object is a real number, and the output is an exact representation of this number, which MuPAD was unable to simplify further. As usual, you can obtain a floating-point approximation by applying `float`:

```
float(%)
```

$$0.41946648$$

As already noted, the symbol `%` is an abbreviation for the previously computed expression (Chapter 13.2).

MuPAD knows the most important mathematical functions such as the square

root `sqrt`, the exponential function `exp`, the trigonometric functions `sin`, `cos`, `tan`, the hyperbolic functions `sinh`, `cosh`, `tanh`, the corresponding inverse functions `ln`, `arcsin`, `arccos`, `arctan`, `arcsinh`, `arccosh`, `arctanh`, as well as a variety of other special functions such as, e.g., the gamma function, the error function `erf`, Bessel functions (`besselI`, `besselJ`, ...), etc. (Section "Special Mathematical Functions" of the MuPAD Quick Reference gives a survey.) In particular, MuPAD knows the rules of manipulation for these functions (e.g., the addition theorems for the trigonometric functions) and applies them. It can compute floating-point approximations such as, e.g., `float(exp(1))=2.718...`, and knows special values (e.g., `sin(PI)=0`).

If you call these functions, they often return themselves symbolically, since this is the most simple exact representation of the corresponding value:

```
sqrt(2), exp(1), sin(x + y)
```

$$\sqrt{2},\ \mathrm{e},\ \sin(x+y)$$

For many users, the main feature of the system is to simplify or transform such expressions using the rules for computation. For example, the system function `expand` "expands" functions such as `exp`, `sin`, etc. by means of the addition theorems if their argument is a symbolic sum:

```
expand(exp(x + y)), expand(sin(x + y)),
expand(tan(x + 3*PI/2))
```

$$\mathrm{e}^x\,\mathrm{e}^y,\ \cos(x)\,\sin(y)+\cos(y)\,\sin(x),\ -\frac{1}{\tan(x)}$$

Generally speaking, one of the main tasks of a computer algebra system is to manipulate and to simplify expressions. Besides `expand`, MuPAD provides the functions `collect`, `combine`, `factor`, `normal`, `partfrac`, `radsimp`, `rewrite`, `simplify`, and `Simplify` for manipulation. They are presented in greater detail in Chapter 9. We briefly mention some of them in what follows.

The function `normal` finds a common denominator for rational expressions:

```
f := x/(1 + x) - 2/(1 - x): g := normal(f)
```

$$\frac{x^2+x+2}{x^2-1}$$

Moreover, `normal` automatically cancels common factors in the numerator and the denominator:

```
normal(x^2/(x + y) - y^2/(x + y))
```

$$x - y$$

Conversely, `partfrac` (short for "partial fraction") decomposes a rational expression into a sum of rational terms with simple denominators:

```
partfrac(g, x)
```

$$\frac{2}{x-1} - \frac{1}{x+1} + 1$$

The functions `simplify` and `Simplify` are universal simplifiers and try to find a representation that is as "simple" as possible:

```
simplify((exp(x) - 1)/(exp(x/2) + 1))
```

$$e^{\frac{x}{2}} - 1$$

You may control the simplification by supplying `simplify` or `Simplify` with additional arguments (see the online documentation for details).

The function `radsimp` simplifies arithmetical expressions containing radicals (roots):

```
f := sqrt(4 + 2*sqrt(3)): f = radsimp(f)
```

$$\sqrt{2}\sqrt{\sqrt{3}+2} = \sqrt{3}+1$$

Here, we have generated an equation, which is a genuine MuPAD object.

Another important function is `factor`, which decomposes an expression into a product of simpler ones:

```
factor(x^3 + 3*x^2 + 3*x + 1),
factor(2*x*y - 2*x - 2*y + x^2 + y^2),
factor(x^2/(x + y) - z^2/(x + y))
```

$$(x+1)^3,\ (x+y-2)\cdot(x+y),\ \frac{(x-z)\cdot(x+z)}{(x+y)}$$

The function `limit` does what its name suggests. For example, the function $\sin(x)/x$ has a removable pole at $x = 0$. Its limit for $x \to 0$ is 1:

```
limit(sin(x)/x, x = 0)
```

$$1$$

In a MuPAD session, you can define functions of your own in several ways. A simple and intuitive method is to use the arrow operator `->` (the minus symbol followed by the "greater than" symbol):

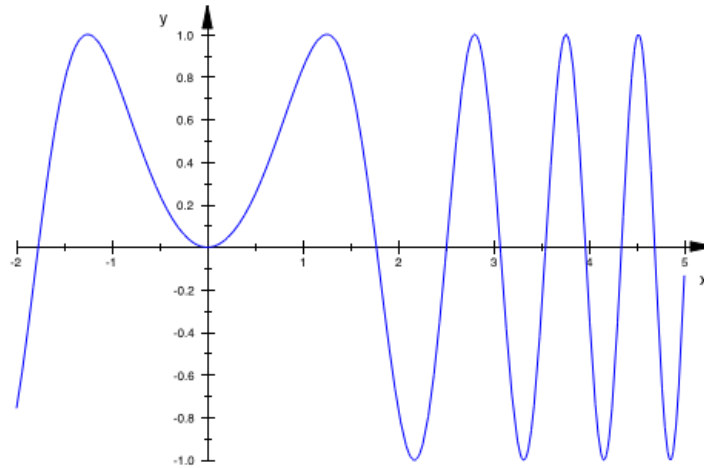
```
F := x -> (x^2):  F(x), F(y), F(a + b), F'(x)
```

$$x^2,\ y^2,\ (a+b)^2,\ 2\,x$$

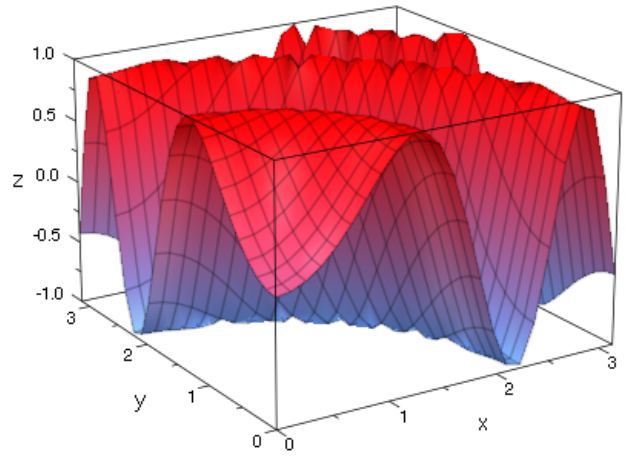
In Chapter 17, we discuss MuPAD programming features and describe how to implement more complex algorithms as MuPAD procedures.

You can also use the graphics facilities to visualize mathematical objects immediately. The relevant MuPAD functions for generating graphics are the `plot` command and the routines from the graphics library `plot`. Here, we present simple calls only, please consult Chapter 11 for in-depth information:

```
plot(sin(x^2), x = -2..5)
```



```
plot(sin(x^2 + y^2), x = 0..PI, y = 0..PI, #3D)
```



Solving equations or systems of equations is certainly an important task for a computer algebra system. This is done via `solve`:

```
equations := {x + y = a, x - a*y = b}:
```

```
unknowns := {x, y}:
```

```
solve(equations, unknowns, IgnoreSpecialCases)
```

$$\left\{\left[x = \frac{a^2+b}{a+1}, y = \frac{a-b}{a+1}\right]\right\}$$

Here, we have generated a set of two equations and a set of unknowns which we wish to solve for. `solve` returns the result in terms of simplified equations, from which you can read off the solution. In the above example, there are two more symbolic parameters a and b. This is why we have told `solve` which of the symbols it should express in terms of the others. The "option" `IgnoreSpecialCases` tells MuPAD to ignore the possibility that `a` could be -1, where the above solution would be incorrect. Without this option, MuPAD returns a complete solution with three branches:

```
solve(equations, unknowns)
```

$$\begin{cases} \left\{\left[x = \frac{a^2+b}{a+1}, y = \frac{a-b}{a+1}\right]\right\} & \text{if} \quad a \neq -1 \\ \{[x = -z-1, y = z]\} & \text{if} \quad a = -1 \wedge b = -1 \\ \emptyset & \text{if} \quad a = -1 \wedge b \neq -1 \end{cases}$$

In the following example, we have only one equation in one unknown. MuPAD automatically recognizes the unknown and solves for it:

```
solve(x^2 - 2*x + 2 = 0)
```

$$\{[x = 1 - \mathrm{i}], [x = 1 + \mathrm{i}]\}$$

If we supply the unknown `x` to solve for, the format of the output changes:

```
solve(x^2 - 2*x + 2 = 0, x)
```

$$\{1 - \mathrm{i}, 1 + \mathrm{i}\}$$

The result is a set containing the two (complex) solutions of the quadratic equation. You find a detailed description of `solve` in Chapter 8.

The functions `sum` and `product` handle symbolic sums and products. For example, the well-known sum $1 + 2 + \cdots + n$ is:

```
sum(i, i = 1..n)
```

$$\frac{n\ (n+1)}{2}$$

The product $1 \cdot 2 \cdots n$ is known as factorial $n!$:

```
product(i^3, i = 1..n)
```

$$n!^3$$

There exist several data structures for vectors and matrices. In principle, you may use arrays (Section 4.9) to represent such objects. However, it is far more intuitive to work with the data type "matrix." You can generate matrices by using the system function `matrix`:

```
A := matrix([[1, 2], [a, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ a & 4 \end{pmatrix}$$

Matrix objects constructed this way have the convenient property that the basic arithmetic operations `+`, `*`, etc. are specialized ("overloaded") according to the appropriate mathematical context. For example, you may use `+` or `*` to add or multiply matrices, respectively (if the dimensions match):

```
B := matrix([[y, 3], [z, 5]]):
```

```
A, B, A + B, A*B
```

$$\begin{pmatrix} 1 & 2 \\ a & 4 \end{pmatrix}, \begin{pmatrix} y & 3 \\ z & 5 \end{pmatrix}, \begin{pmatrix} y+1 & 5 \\ a+z & 9 \end{pmatrix}, \begin{pmatrix} y+2\,z & 13 \\ 4\,z+a\,y & 3\,a+20 \end{pmatrix}$$

The power `A^(-1)`, equivalent to `1/A`, denotes the inverse of the matrix:

```
A^(-1)
```

$$\begin{pmatrix} -\frac{2}{a-2} & \frac{1}{a-2} \\ \frac{a}{2\,(a-2)} & -\frac{1}{2\,(a-2)} \end{pmatrix}$$

The function `linalg::det`, from the MuPAD `linalg` library for linear algebra (Section 4.15), computes the determinant:

```
linalg::det(A)
```

$$4 - 2\,a$$

Column vectors of dimension n can be interpreted as $n \times 1$ matrices:

```
b := matrix([1, x])
```

$$\begin{pmatrix} 1 \\ x \end{pmatrix}$$

You can comfortably determine the solution $A^{-1}\vec{b}$ of the system of linear equations $A\vec{x} = \vec{b}$, with the above coefficient matrix A and the previously defined $\vec{b}$ on the right-hand side:

```
solutionVector := A^(-1)*b
```

$$\begin{pmatrix} \frac{x}{a-2} - \frac{2}{a-2} \\ \frac{a}{2\,(a-2)} - \frac{x}{2\,(a-2)} \end{pmatrix}$$

Now you can apply the function `normal` to each component of the vector by means of the system function `map`, thus simplifying the representation:

```
map(%, normal)
```

$$\begin{pmatrix} \frac{x-2}{a-2} \\ \frac{a-x}{2\,a-4} \end{pmatrix}$$

To verify the computation, you may multiply the solution vector by the matrix `A`:

```
A * %
```

$$\begin{pmatrix} \frac{x-2}{a-2} + \frac{2\,(a-x)}{2\,a-4} \\ \frac{4\,(a-x)}{2\,a-4} + \frac{a\,(x-2)}{a-2} \end{pmatrix}$$

After simplification, you can check that the result equals `b`:

```
map(%, normal)
```

$$\begin{pmatrix} 1 \\ x \end{pmatrix}$$

Section 4.15 provides more information on handling matrices and vectors.

Exercise 2.3: Compute an expanded form of the expression $(x^2 + y)^5$.

Exercise 2.4: Use MuPAD to check that $\dfrac{x^2 - 1}{x + 1} = x - 1$ holds.

Exercise 2.5: Generate a plot of the function $1/\sin(x)$ for $1 \le x \le 10$.

Exercise 2.6: Obtain detailed information about the function `limit`. Use MuPAD to verify the following limits:

$$\lim_{x\to 0}\frac{\sin(x)}{x}=1\,,\quad \lim_{x\to 0}\frac{1-\cos(x)}{x}=0\,,\quad \lim_{x\to 0+}\ln(x)=-\infty\,,$$
$$\lim_{x\to 0}x^{\sin(x)}=1\,,\quad \lim_{x\to\infty}\left(1+\frac{1}{x}\right)^{x}=e\,,\quad \lim_{x\to\infty}\frac{\ln(x)}{e^{x}}=0\,,$$
$$\lim_{x\to 0+}x^{\ln(x)}=\infty\,,\quad \lim_{x\to\infty}\left(1+\frac{\pi}{x}\right)^{x}=e^{\pi}\,,\quad \lim_{x\to 0-}\frac{2}{1+e^{-1/x}}=0\,.$$

The limit $\lim_{x\to 0} e^{\cot(x)}$ does not exist. How does MuPAD react?

Exercise 2.7: Obtain detailed information about the function `sum`. The call `sum(f(k), k=a..b)` computes a *closed form* of a finite or infinite sum, if possible. Use MuPAD to verify the following identity:

$$\sum_{k=1}^{n}(k^2+k+1)=\frac{n\,(n^2+3\,n+5)}{3}\,.$$

Determine the values of the following series:

$$\sum_{k=0}^{\infty}\frac{2\,k-3}{(k+1)\,(k+2)\,(k+3)}\,,\quad \sum_{k=2}^{\infty}\frac{k}{(k-1)^2\,(k+1)^2}\,.$$

Exercise 2.8: Compute $2\cdot(A+B)$, $A\cdot B$, and $(A-B)^{-1}$ for the following matrices:

$$A=\begin{pmatrix}1&2&3\\4&5&6\\7&8&0\end{pmatrix},\quad B=\begin{pmatrix}1&1&0\\0&0&1\\0&1&0\end{pmatrix}.$$

Curve Sketching

In the following sample session, we use some of the system functions from the previous section to sketch and discuss the curve given by the rational function

$$f:\ x\ \mapsto\ \frac{(x-1)^2}{x-2}+a$$

with a parameter a. First, we determine some characteristics of this function: discontinuities, extremal values, and behavior for large x. We assume that a is a real number.

```
assume(a in R_):
f := x -> (x - 1)^2/(x - 2) + a:
```

```
singularities := discont(f(x), x)
```

$$\{2\}$$

The function `discont` determines the discontinuities of the function f with respect to the variable x. It returns a set of such points. Thus, the above f is defined and continuous for all $x \neq 2$. Obviously, $x = 2$ is a pole. Indeed, MuPAD® finds the limit $\mp\infty$ when you approach this point from the left or from the right, respectively:

```
limit(f(x), x = 2, Left), limit(f(x), x = 2, Right)
```

$$-\infty,\ \infty$$

You find the roots of f by solving the equation $f(x) = 0$:

```
roots := solve(f(x) = 0, x)
```

$$\left\{1 - \frac{\sqrt{a^2 + 4\,a}}{2} - \frac{a}{2}, \frac{\sqrt{a^2 + 4\,a}}{2} - \frac{a}{2} + 1\right\}$$

Depending on a, either both or none of the two roots are real. Now, we want to find the local extrema of f. To this end, we determine the roots of the first derivative f':

```
f'(x)
```

$$\frac{2\,x - 2}{x - 2} - \frac{(x - 1)^2}{(x - 2)^2}$$

```
extrema := solve(f'(x) = 0, x)
```

$$\{1, 3\}$$

These are the candidates for local extrema. However, some of them might be saddle points. If the second derivative f'' does not vanish at these points, then both are really extrema. We check:

```
f''(1), f''(3)
```

$$-2,\ 2$$

Our results imply that f has the following properties: for any choice of the parameter a, there is a local maximum at $x = 1$, a pole at $x = 2$, and a local minimum at $x = 3$. The corresponding values of f at these points are

```
maxvalue := f(1); minvalue := f(3)
```

$$a$$

$$a + 4$$

f tends to $\mp\infty$ for $x \to \mp\infty$:

```
limit(f(x), x = -infinity), limit(f(x), x = infinity)
```

$$-\infty,\ \infty$$

We can specify the behavior of f more precisely for large values of x. It asymptotically approaches the linear function $x \mapsto x + a$:

```
series(f(x), x = infinity)
```

$$x + a + \frac{1}{x} + \frac{2}{x^2} + \frac{4}{x^3} + \frac{8}{x^4} + O\left(\frac{1}{x^5}\right)$$

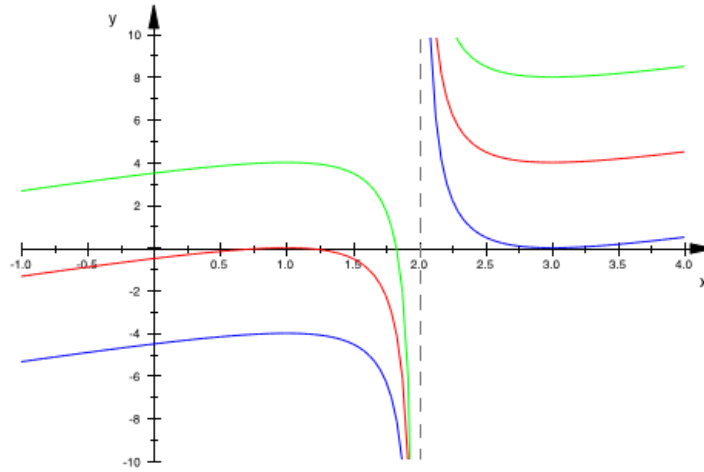
Here we have employed the function `series` to compute an asymptotic expansion of f (Section 4.13). We can easily check our results visually by plotting the graph of f for several values of a:

```
F := f(x) | a = -4: G := f(x) | a = 0: H := f(x) | a = 4:
F, G, H
```

$$\frac{(x-1)^2}{x-2} - 4,\ \frac{(x-1)^2}{x-2},\ \frac{(x-1)^2}{x-2} + 4$$

The operator `|` (Chapter 6) evaluates an expression at some point: in the example, we have substituted the concrete values -4, 0 and 4, respectively, for a. We now can plot the three functions together in one picture:

```
plot(F, G, H, x = -1..4)
```



Elementary Number Theory

MuPAD® provides a lot of elementary number theoretic functions, for example:

- `isprime(n)` tests whether $n \in \mathbb{N}$ is a prime number,
- `ithprime(n)` returns the n-th prime number,
- `nextprime(n)` finds the least prime number $\geq n$,
- `prevprime(n)` finds the largest prime number $\leq n$,
- `ifactor(n)` computes the prime factorization of n.

These routines are quite fast. However, since they employ probabilistic primality tests, they may return wrong results with very small probability.[5] Instead of `isprime`, you can use the (slower) function `numlib::proveprime` as an error-free primality test.

[5] In practice, you need not worry about this because the chances of a wrong answer are negligible: the probability of a hardware failure is much higher than the probability that the randomized test returns the wrong answer on correctly working hardware.

Let us generate a list of all primes up to 10 000. Here is one of many ways to do this:

```
primes := select([$ 1..10000], isprime)
```

```
      [2, 3, 5, 7, 11, 13, 17, ..., 9949, 9967, 9973]
```

First, we have generated the sequence of all positive integers up to 10 000 by means of the sequence generator `$` (Section 4.5). The square brackets `[]` convert this to a MuPAD list. Then `select` (Section 4.6) eliminates all those list elements for which the function `isprime`, supplied as second argument, returns `FALSE`. The number of these primes equals the number of list elements, which we can obtain via `nops` ("number of operands," Section 4.1):

```
nops(primes)
```

```
   1229
```

Alternatively, we may generate the same prime list by

```
primes := [ithprime(i) $ i = 1..1229]:
```

Here we have used the fact that we already know the number of primes up to 10 000. Another possibility is to generate a large list of primes and discard the ones greater than 10 000:

```
primes := select([ithprime(i) $ i=1..5000],
                 x -> (x<=10000)):
```

Here, the object `x -> (x <= 10000)` represents the function that maps each `x` to the inequality `x <= 10000`. The `select` command then keeps only those list elements for which the inequality evaluates to `TRUE`.

In the next example, we use a `repeat` loop (Chapter 15) to generate the list of primes. With the help of the concatenation operator `.` (Section 4.6), we successively append primes i to a list until `nextprime(i+1)`, the next prime greater than i, exceeds 10 000. We start with the empty list and the first prime $i = 2$:

```
primes := []: i := 2:
```

```
repeat
  primes := primes . [i];
  i := nextprime(i + 1)
until i > 10000 end_repeat:
```

Now, we consider Goldbach's famous conjecture:

> "Every even integer greater than 2 is the sum of two primes."

We want to verify this conjecture for all even numbers up to 10 000. First, we generate the list of even integers `[4, 6, …, 10000]`:

```
list := [2*i $ i = 2..5000]:
```

```
nops(list)
```

$$4999$$

Now, we select those numbers from the list that cannot be written in the form "prime + 2." This is done by testing for each i in the list whether $i - 2$ is a prime:

```
list := select(list, i -> (not isprime(i - 2))):
```

```
nops(list)
```

$$4998$$

The only integer that has been eliminated is 4 (since for all other even positive integers $i - 2$ is even and greater than 2, and hence not prime). Now we discard all numbers of the form "prime + 3":

```
list := select(list, i -> (not isprime(i - 3))):
```

```
nops(list)
```

$$3770$$

The remaining 3770 integers are neither of the form "prime + 2" nor of the form "prime + 3." We now continue this procedure by means of a `while` loop (Chapter 15). In the loop, j successively runs through all primes > 3, and the numbers of the form "prime + j" are eliminated. A `print` command (Section 12.1) outputs the number of remaining integers in each step. The loop ends as soon as the list is empty:

```
j := 3:
```

```
while list <> [] do
    j := nextprime(j + 1):
    list := select(list, i -> (not isprime(i - j))):
    print(j, nops(list)):
end_while:
```

```
                          5, 2747

                          7, 1926

                          11, 1400

                             ...

                           163, 1

                           167, 1

                           173, 0
```

Thus we have confirmed that Goldbach's conjecture holds true for all even positive integers up to $10\,000$. We have even shown that all those numbers can be written as a sum of a prime less or equal to 173 and another prime.

In the next example, we generate a list of distances between two successive primes up to 500:

```
primes := select([$ 1..500], isprime):
```

```
distances := [primes[i] - primes[i - 1]
              $ i = 2..nops(primes)]
```

```
  [1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4, 6, 6, 2,

     6, 4, 2, 6, 4, 6, 8, 4, 2, 4, 2, 4, 14, 4, 6, 2,

     10, 2, 6, 6, 4, 6, 6, 2, 10, 2, 4, 2, 12, 12, 4,

     2, 4, 6, 2, 10, 6, 6, 6, 2, 6, 4, 2, 10, 14, 4, 2,

     4, 14, 6, 10, 2, 4, 6, 8, 6, 6, 4, 6, 8, 4, 8, 10,

     2, 10, 2, 6, 4, 6, 8, 4, 2, 4, 12, 8, 4, 8]
```

The indexed call `primes[i]` returns the ith element in the list.

The function `zip` (Section 4.6) provides an alternative method. The call `zip(a, b, f)` combines two lists $a = [a_1, a_2, \ldots]$ and $b = [b_1, b_2, \ldots]$ componentwise by means of the function f: the resulting list is

$$[f(a_1, b_1), f(a_2, b_2), \ldots]$$

and has as many elements as the shorter of the two lists. In our example, we apply this to the prime list $a = [a_1, \ldots, a_n]$, the "shifted" prime list $b = [a_2, \ldots, a_n]$, and the function $(x, y) \mapsto y - x$. We first generate a shifted copy of the prime list by deleting the first element, thus shortening the list:

```
b := primes: delete b[1]:
```

The following command returns the same result as above:

```
distances := zip(primes, b, (x, y) -> (y - x)):
```

We have presented another useful function in Section 2.3, the routine `ifactor` for factoring an integer into primes. The call `ifactor(n)` returns an object of the same type as `factor`: it is a special data type called `Factored`. Objects of this type are printed on the screen in a form that is easily readable:

```
ifactor(-123456789)
```

$$-3^2 \cdot 3607 \cdot 3803$$

Internally, the prime factors and the exponents are stored in form of a list, and you can extract them by using `op` or by an indexed access. Consult the help pages of `ifactor` and `Factored` for details. The internal list has the format

$$[s, p_1, e_1, \ldots, p_k, e_k]$$

with primes $p_1, \ldots, p_k$, their exponents $e_1, \ldots, e_k$, and the sign $s = \pm 1$, such that $n = s \cdot p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$:

```
op(%)
```

$$-1,\ 3,\ 2,\ 3607,\ 1,\ 3803,\ 1$$

We now employ the function `ifactor` to find out how many integers between 2 and $10\,000$ are divisible by exactly two distinct prime numbers. We note that the object returned by `ifactor(n)` has $2\,m + 1$ elements in its list representation, where m is the number of distinct prime divisors of n. Thus, the function

```
m := n -> (nops(ifactor(n)) - 1)/2:
```

returns the number of distinct prime factors. We construct the list of values $m(k)$ for $k = 2, \ldots, 10000$:

```
list := [m(k) $ k = 2..10000]:
```

The following `for` loop (Section 15) displays the number of integers with precisely $i = 1, 2, \ldots, 6$ distinct prime divisors:

```
for i from 1 to 6 do
    print(i, nops(select(list, x -> (x = i))))
end_for:
```

```
    1, 1280

    2, 4097

    3, 3695

    4, 894

    5, 33

    6, 0
```

Thus there are 1280 integers with exactly one prime divisor in the scanned interval,[6] 4097 integers with precisely two distinct prime factors, and so on. It is easy to see why the interval contains no integer with six or more prime divisors: the smallest such number $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 = 30\,030$ exceeds $10\,000$.

The `numlib` library comprises various number theoretic functions. Among others, it contains the routine `numlib::numprimedivisors`, equivalent to the above `m`, for computing the number of prime divisors. We refer to Chapter 3 for a description of the MuPAD libraries.

[6] We have already seen that the interval contains 1229 prime numbers. Can you explain the difference?

Exercise 2.9: Primes of the form $2^n \pm 1$ always have produced particular interest.

a) Primes of the form $2^p - 1$, where p is a prime, are called *Mersenne primes*. Find all Mersenne primes for $1 < p \leq 1000$.

b) For a positive integer n, the *n-th Fermat number* is $2^{(2^n)} + 1$. Refute Fermat's conjecture that all those numbers are primes.

3

The MuPAD® Libraries

Most of MuPAD®'s mathematical knowledge is organized in libraries. Such a library comprises a collection of functions for solving problems in a particular area, such as linear algebra, number theory, numerical analysis, etc. Library functions are written in the MuPAD programming language. You can use them in the same way as kernel functions, without knowing anything about the programming language.

Section "Libraries and Modules" of the MuPAD Quick Reference contains a survey of all libraries. Alternatively, the help viewer contains a list of libraries (and third-party packages installed) in its contents pane on the left.

The libraries are developed continuously, and future versions of MuPAD will provide additional functionality.

In this chapter, we do not address the mathematical functionality of libraries but rather discuss their general use.

Information About a Particular Library

You can obtain information and help for libraries by calling the functions `info` and `help`. The function `info` gives a list of all functions installed in the library. The `numlib` library is a collection of number theoretic functions:

```
info(numlib)
```

```
  Library 'numlib':     the package for elementary
  number theory

  -- Interface:
  numlib::Lambda,        numlib::Omega,
  numlib::contfrac,      numlib::cornacchia,
  numlib::decimal,       numlib::divisors,
  numlib::ecm,           numlib::factorGaussInt,
  numlib::fibonacci,     numlib::fromAscii,
  ...
```

The commands `help` or `?` supply a more detailed description of the library.

The function `numlib::decimal` of the `numlib` library computes the decimal expansion of a rational number:[1]

```
numlib::decimal(123/7)
```

$$17,\ [5, 7, 1, 4, 2, 8]$$

As for other system functions, you will see information when hovering your mouse pointer over the name of a library function and you may request information on the function by means of `help` or `?`:

```
?numlib::decimal
```

[1] The result is to be interpreted as: $123/7 = 17.\overline{571428} = 17.571428\,571428\,\ldots$

You can have a look at the implementation of a library function by using expose:

```
expose(numlib::decimal)
```

```
  proc(a)
    name numlib::decimal;
    local p, q, s, l, i;
  begin
    if not testtype(a, Type::Numeric) then
       ...
  end_proc
```

Exporting Libraries

In the previous section, you have seen that the calling syntax for a library function is `library::function`, where `library` and `function` are the names of the library and the function, respectively. For example, the library `numeric` for numerical computations contains the function `numeric::fsolve`. It implements a modified version of the well-known Newton method for numerical root finding. In the following example, we approximate a root of the sine function in the interval $[2, 4]$:

```
numeric::fsolve(sin(x), x = 2..4)
```

$$[x = 3.141592654]$$

The function `use`[2] makes functions of a library "globally known," so that you can use them without specifying the library name:

```
use(numeric, fsolve): fsolve(sin(x), x = 2..4)
```

$$[x = 3.141592654]$$

If you already have assigned a value to the name of the function to be exported, `use` returns a warning and the function is not exported. In the following, the numerical integrator `numeric::quadrature` cannot be exported to the name `quadrature`, because this identifier has a value:

```
quadrature := 1: use(numeric, quadrature)
  Warning: 'quadrature' already has a value, not exported.
```

After deletion of the value, the name of the function is available and the corresponding library function can be exported successfully. One can export several functions at once:

```
delete quadrature:
```

```
use(numeric, realroots, quadrature):
```

Now you can use `realroots` (to find *all* real roots of an expression in an interval) and `quadrature` (for numerical integration) directly. Please refer to the corresponding help pages for the meaning of the input parameters and the returned output. Please note that neither tooltips nor the `help` command will

[2] This function used to be called `export`, but that name is more appropriately used for the library exporting data in formats for other programs.

react to exported names. You will need to use the full name (or the search functionality of the help browser) in these cases.

```
realroots(x^4 + x^3 - 6*x^2 + 11*x - 6,
          x = -10..10, 0.001)
```

$$[[-3.623046875, -3.62109375], [0.8217773438, 0.822265625]]$$

```
quadrature(exp(x) + 1, x = 0..1)
```

$$2.718281828$$

If you call `use` with only one argument, namely the name of the library, then all functions in that library are exported. If there are name conflicts with already existing identifiers, then `use` issues warnings:

```
eigenvalues := 1: use(numeric)
```

```
  Info: 'numeric::quadrature' already is exported.
  Info: 'numeric::realroots' already is exported.
  Warning: 'indets' already has a value, not exported.
  Info: 'numeric::fsolve' already is exported.
  Warning: 'rationalize' already has a value, not
           exported.
  Warning: 'linsolve' already has a value, not exported.
  Warning: 'sum' already has a value, not exported.
  Warning: 'int' already has a value, not exported.
  Warning: 'solve' already has a value, not exported.
  Warning: 'sort' already has a value, not exported.
  Warning: 'eigenvalues' already has a value, not
           exported.
```

After deleting the value of the identifier `eigenvalues`, the library function with the same name can be exported successfully:

```
delete eigenvalues: use(numeric, eigenvalues):
```

However, the other name conflicts `int`, `solve` etc. cannot be resolved. The important symbolic system functions `int`, `solve` etc. should not be replaced by their numerical counterparts `numeric::int`, `numeric::solve` etc.

The Standard Library

The standard library is the most important MuPAD® library. It contains the most frequently used functions such as `diff`, `simplify`, etc. (For an even more restricted selection of frequently used commands, check the command bar at the right-hand side of the notebook interface and the MuPAD source code editor.) The functions of the standard library do not have a prefix separated with `::`, as other functions do (unless exported). In this respect, there is no notable difference between the MuPAD kernel functions, which are written in the C++ programming language, and the other functions from the standard library, which are implemented in the MuPAD programming language.

You obtain more information about the available functions of the standard library by navigating to the relevant section in the help system, which is the first reference section. The MuPAD Quick Reference lists all functions of the standard library in MuPAD version 5.

Many of these functions are implemented as function environments (Section 17.12). You can view the source code via `expose(name)`:

```
expose(exp)
```

```
  proc(x)
    name exp;
    local y, lny, c;
    option noDebug;
  begin
    if args(0) = 0 then
      error("expecting one argument")
    else
      if x::dom::exp <> FAIL then
        return(x::dom::exp(args()))
      else
        if args(0) <> 1 then
          error("expecting one argument")
        end_if
      end_if
    end_if;
    ...
  end_proc
```

4

MuPAD® Objects

In Chapter 2, we introduced MuPAD® objects such as numbers, symbolic expressions, maps, or matrices. Now, we discuss these objects more systematically.

The objects sent to the kernel for evaluation can be of various forms: arithmetic expressions with numbers such as `1+(1+I)/3`, arithmetic expressions with symbolic objects such as `x+(y+I)/3`, lists, sets, equations, inequalities, maps, arrays, abstract mathematical objects, and more. Every MuPAD object belongs to some data type, called the *domain type*. It corresponds to a certain internal representation of the object. In what follows, we discuss the following fundamental domain types. As a convention, the names of domains provided by the MuPAD kernel consist of capital letters, such as `DOM_INT`, `DOM_RAT` etc., while domains implemented in the MuPAD language such as `Series::Puiseux` or `Dom::Matrix(R)` involve small letters:

domain type	meaning
`DOM_INT`	integers, e.g., `-3`, `10^5`
`DOM_RAT`	rational numbers, e.g., `7/11`
`DOM_FLOAT`	floating-point numbers, e.g., `0.123`
`DOM_COMPLEX`	complex numbers, e.g., `1+2/3*I`
`DOM_INTERVAL`	floating-point intervals, e.g., `2.1...3.2`
`DOM_IDENT`	symbolic identifiers, e.g., `x`, `y`, `f`
`DOM_EXPR`	symbolic expressions, e.g., `x+y`
`Series::Puiseux`	symbolic series expansions, e.g., `1+x+x^2+O(x^3)`
`DOM_LIST`	lists, e.g., `[1, 2, 3]`
`DOM_SET`	sets, e.g., `{1, 2, 3}`
`DOM_ARRAY`	arrays

domain type	meaning
DOM_TABLE	tables
DOM_BOOL	Boolean values: TRUE, FALSE, UNKNOWN
DOM_STRING	strings, e.g., "I am a string"
Dom::Matrix(R)	matrices and vectors over the ring R
DOM_POLY	polynomials, e.g., poly(x^2+x+1, [x])
DOM_PROC	functions and procedures

Moreover, you can define your own data types, but we do not discuss this here. The system function domtype returns the domain type of a MuPAD object.

In the following section, we first present the important operand function op, which enables you to decompose a MuPAD object into its building blocks. The following sections discuss the above data types and some of the main system functions for handling them.

Operands: the Functions op and nops

It is often necessary to decompose a MuPAD® object into its components in order to process them individually. The building blocks of an object are called *operands*. The system functions for accessing them are `op` and `nops` (short for: number of operands):

`nops(object)`	: the number of operands,
`op(object,i)`	: the i-th operand, $0 \le i \le$ `nops(object)`,
`op(object,i..j)`	: the sequence of operands i through j, where $0 \le i \le j \le$ `nops(object)`,
`op(object)`	: the sequence `op(object, 1)`, `op(object, 2)`, ... of all operands.

The meaning of an operand depends on the data type of the object. We discuss this for each data type in detail in the following sections. For example, the operands of a rational number are the numerator and the denominator, the operands of a list or set are the elements, and the operands of a function call are the arguments. However, there are also objects where the decomposition into operands is less intuitive, such as series expansions as generated by the system functions `taylor` or `series` (Section 4.13). Here is an example with a list (Section 4.6):

```
list := [a, b, c, d, sin(x)]: nops(list)
```

$$5$$

```
op(list, 2)
```

$$b$$

```
op(list, 3..5)
```

$$c,\ d,\ \sin(x)$$

```
op(list)
```

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

By repeatedly calling the `op` function, you can decompose arbitrary MuPAD expressions into “atomic” ones. In this model, a MuPAD atom is an expression

that cannot be further decomposed by `op`, such that `op(atom) = atom` holds.[1] This is essentially the case for integers, floating-point numbers, identifiers that have not been assigned a value, and for strings:

```
op(-2), op(0.1234), op(a), op("I am a text")
```

$$-2,\ 0.1234,\ a,\ \text{"I am a text"}$$

In the following example, a nested list is decomposed completely into its atoms `a11`, `a12`, `a21`, `x`, `2`:

```
list := [[a11, a12], [a21, x^2]]
```

The operands and suboperands are:

```
op(list, 1)                 :  [a11, a12]
op(list, 2)                 :  [a21, x^2]
op(op(list, 1), 1)          :  a11
op(op(list, 1), 2)          :  a12
op(op(list, 2), 1)          :  a21
op(op(list, 2), 2)          :  x^2
op(op(op(list, 2), 2), 1)   :  x
op(op(op(list, 2), 2), 2)   :  2
```

[1] This model is a good approximation to MuPAD's internal mode of operation, but there are some exceptions. For example, you can decompose rational numbers via `op`, but the kernel regards them as atoms. On the other hand, although strings are indecomposable with respect to `op`, it is still possible to access the characters of a string individually (Section 4.11).

Instead of the annoying nested calls of `op`, you may also use the following short form to access subexpressions:

```
op(list, [1])       :  [a11, a12]
op(list, [2])       :  [a21, x^2]
op(list, [1, 1])    :  a11
op(list, [1, 2])    :  a12
op(list, [2, 1])    :  a21
op(list, [2, 2])    :  x^2
op(list, [2, 2, 1]) :  x
op(list, [2, 2, 2]) :  2
```

Exercise 4.1: Determine the operands of the power `a^b`, the equation `a=b`, and the symbolic function call `f(a,b)`.

Exercise 4.2: The following call of `solve` (Chapter 8) returns a set:

```
set := solve({x + sin(3)*y = exp(a),
              y - sin(3)*y = exp(-a)}, {x,y})
```

$$\left\{\left[x = \frac{e^{a}\ \sin(3) - e^{a} + \frac{\sin(3)}{e^{a}}}{\sin(3) - 1}, y = -\frac{1}{e^{a}\ (\sin(3) - 1)}\right]\right\}$$

Extract the value of the solution for y and assign it to the identifier `y`.

Numbers

We have demonstrated in Section 2.3 how to work with numbers. There are various data types for numbers:

```
domtype(-10), domtype(2/3), domtype(0.1234),
domtype(0.1 + 2*I)
```

$$\text{DOM\_INT},\ \text{DOM\_RAT},\ \text{DOM\_FLOAT},\ \text{DOM\_COMPLEX}$$

A rational number is a compound object: the building blocks are the numerator and the denominator. Similarly, a complex number consists of the real and the imaginary part. You can use the operand function `op` from the previous section to access these components:

```
op(111/223, 1), op(111/223, 2)
```

$$111,\ 223$$

```
op(100 + 200*I, 1), op(100 + 200*I, 2)
```

$$100,\ 200$$

Alternatively, you can use the system functions `numer`, `denom`, `Re`, and `Im`:

```
numer(111/223), denom(111/223),
Re(100 + 200*I), Im(100 + 200*I)
```

$$111,\ 223,\ 100,\ 200$$

Besides the common arithmetic operations `+`, `-`, `*`, and `/`, there are the arithmetic operators `div` and `mod` for division of an integer x by a non-zero integer p with remainder. If $x = k\,p + r$ holds with integers k and $0 \le r < |p|$, then `x div p` returns the "integral quotient" k and `x mod p` returns the "remainder" r:

```
25 div 4, 25 mod 4
```

$$6,\ 1$$

Table 4.2 gives a compilation of the main MuPAD functions and operators for handling numbers. We refer to the help system (i.e., `?abs`, `?ceil` etc.) for a detailed description of these functions. We stress that while expressions such as $\sqrt{2}$ mathematically represent numbers, MuPAD treats them as symbolic expressions (Section 4.4):

`+, -, *, /, ^`	: basic arithmetic operations
`abs`	: absolute value
`ceil`	: rounding up
`div`	: quotient “modulo”
`fact`	: factorial
`float`	: approximation by floating-point numbers
`floor`	: rounding down
`frac`	: fractional part
`ifactor`, `factor`	: prime factorization
`isprime`	: primality test
`mod`	: remainder “modulo”
`round`	: rounding to nearest
`sign`	: sign
`sqrt`	: square root
`trunc`	: integral part

Table 4.2: MuPAD® functions and operators for numbers

```
domtype(sqrt(2))
    DOM_EXPR
```

Exercise 4.3: What is the difference between $1/3 + 1/3 + 1/3$ and $1.0/3 + 1/3 + 1/3$ in MuPAD?

Exercise 4.4: Compute the decimal expansions of $\pi^{(\pi^\pi)}$ and $e^{\frac{1}{3}\,\pi\,\sqrt{163}}$ with a precision of 10 and 100 digits, respectively. What is the 234-th digit after the decimal point of π?

Exercise 4.5: After you execute `x:=10^50/3.0`, only the first `DIGITS` decimal digits of `x` are guaranteed to be correct.

a) Truncating the fractional part via `trunc` is therefore questionable. What does MuPAD do?

b) What is returned for `x` after increasing `DIGITS`?

Identifiers

Identifiers are names such as `x` or `f` that may represent variables and unknowns. They may consist of arbitrary combinations of letters, digits, the hash mark "#," and the underscore "_," with the only exception that the first symbol must not be a digit. Identifiers starting with a hash mark can never have values or properties. MuPAD® distinguishes uppercase and lowercase letters. Examples of admissible identifiers are `x`, `_x23`, and `the_MuPAD_system`, while MuPAD would not accept `12x`, `p-2`, and `x>y` as identifiers. MuPAD also accepts any sequence of characters starting and ending with a 'backtick' ` as an identifier, so `` `x>y` `` is, in fact, an identifier. We will not use such identifiers in this tutorial.

Identifiers that have not been assigned a value evaluate to their name. In MuPAD, they represent symbolic objects such as unknowns in equations. Their domain type is `DOM_IDENT`:

```
domtype(x)
```

```
    DOM_IDENT
```

Identifiers can contain special characters, such as an α. We recommend using the `Symbol` library for generating these. Using `_`, `^`, `$`, and `{}`, you can also create identifiers with super- and subscripts, as in a_1^1. We recommend using the `Symbol` library for generating these, too.

You can assign an arbitrary object to an identifier by means of the *assignment operator* `:=`. Afterwards, this object is the *value* of the identifier. For example, after the command

```
x := 1 + I:
```

the identifier `x` has the value `1+I`, which is a complex number of domain type `DOM_COMPLEX`. You should be careful to distinguish between an identifier, its value, and its evaluation. We refer to the important Chapter 5, where MuPAD's evaluation strategy is described.

If an identifier already has been assigned a value, another assignment overwrites the previous value. The statement `y:=x` does not assign the identifier `x` to the identifier `y`, but the current value (the evaluation) of `x`:

```
x := 1: y := x: x, y
```
$$1,\ 1$$

If the value of x is changed later on, this does not affect y:

```
x := 2: x, y
```
$$2,\ 1$$

However, if x is a symbolic identifier, which evaluates to itself, the new identifier y refers to this symbol:

```
delete x: y := x: x, y; x := 2: x, y
```
$$x,\ x$$

$$2,\ 2$$

Here we have deleted the value of the identifier x by means of the function `delete`. After deletion, x is again a symbolic identifier without a value.

The assignment operator `:=` is a short form of the system function `_assign`, which may also be called directly:

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

This function returns its second argument, namely the right-hand side of an assignment. This explains the screen output after an assignment:

```
y := 2*x
```
$2\,\mathrm{value}$

You can work with the returned value immediately. For example, the following construction is allowed (the assignment must be put in parentheses):

```
y := cos((x := 0)): x, y
```
$$0,\ 1$$

Here the value 0 is assigned to the identifier x. The return value of the assignment, i.e., 0, is fed directly as argument to the cosine function, and the result $\cos(0) = 1$ is assigned to y. Thus, we have simultaneously assigned values to both x and y.

Another assignment function is `assign`. Its input are sets or lists of equations, which are transformed into assignments:

```
delete x, y: assign({x = 0, y = 1}): x, y
```

$$0,\ 1$$

This function is particularly useful in connection with `solve` (Section 8), which returns solutions as a set containing lists of equations of the form `identifier=value` without assigning these values.

There exist many identifiers with predefined values. They represent mathematical functions (such as `sin`, `exp`, or `sqrt`), mathematical constants (such as `PI`), or MuPAD algorithms (such as `diff`, `int`, or `limit`). If you try to change the value of such a predefined identifier, MuPAD issues a warning or an error message:

```
sin := 1
  Error: Identifier 'sin' is protected [_assign]
```

You can protect your own identifiers against overwriting via the command `protect(identifier)`. The write protection of both your own and of system identifiers can be removed by `unprotect(identifier)`. However, we strongly recommend not to overwrite predefined identifiers, since they are used by many system functions which would return unpredictable results after a redefinition. Identifiers starting with a hash mark (`#`) are always protected and cannot be unprotected. The command `anames(All)` lists all currently defined identifiers.

You can use the concatenation operator "`.`" to generate names of identifiers dynamically. If `x` and `i` are identifiers, then `x.i` generates a new identifier by concatenating the *evaluations* (see Chapter 5) of `x` and `i`:

```
x := z: i := 2:  x.i
```

$$z2$$

```
x.i := value: z2
```

$$value$$

In the following example, we use a `for` loop (Chapter 15) to assign values to the identifiers `x1`, `...`, `x1000`:

```
delete x:
```

```
for i from 1 to 1000 do x.i := i^2 end_for:
```

Due to possible side effects or conflicts with already existing identifiers, we strongly recommend to use this concept only interactively and not within MuPAD procedures.

The function `genident` generates a new identifier that has not been used before in the current MuPAD session:

```
X3 := (X2 := (X1 := 0)): genident()
```

$$X4$$

You may use strings enclosed in quotation marks " (Section 4.11) to generate identifiers dynamically:

```
a := email: b := "4you": a.b
```

$$email4you$$

Even if the string contains blanks or operator symbols, a valid identifier is generated, which MuPAD displays using the backtick notation mentioned above:

```
a := email: b := "4you + x": a.b
```

$$email4you + x$$

Strings are not identifiers and cannot be assigned a value:

```
"string" := 1
  Error: Invalid left-hand side in assignment [line 1, col 10]
```

Exercise 4.6: Which of the following names `x`, `x2`, `2x`, `x_t`, `diff`, `exp`, `caution!-!`, `x-y`, `Jack&Jill`, `a_valid_identifier`, `#1` are valid identifiers?Which of them can be assigned values?

Exercise 4.7: Read the help page for `solve`. Solve the system of equations

$$x_1 + x_2 = 1,\ x_2 + x_3 = 1,\ \ldots,\ x_{19} + x_{20} = 1,\ x_{20} = \pi$$

in the unknowns $x_1, x_2, \ldots, x_{20}$. Read the help page for `assign` and assign the values of the solution to the unknowns.

Symbolic Expressions

We say that an object containing symbolic terms such as the equation

$$0.3 + \sin(3) + \frac{f(x,y)}{5} = 0$$

is an *expression*. Expressions of domain type `DOM_EXPR` are probably the most general data type in MuPAD®. Expressions are built of atomic components, as all MuPAD objects, and are composed by means of *operators*. This comprises binary operators, such as the basic arithmetic operations `+`, `-`, `*`, `/`, `^`, and function calls such as `sin(·)`, `f(·)`, etc.

Operators

MuPAD® throughout uses functions to combine or manipulate objects.[2] It would be little intuitive, however, to use function calls everywhere,[3] say, `_plus(a, b)` for the addition $a + b$. For that reason, a variety of important operations is implemented in such a way that you can use the familiar mathematical notation ("operator notation") for input. Also the output is given in such a form. In the following, we list the operators for building more complex MuPAD expressions from atoms.

The operators `+`, `-`, `*`, `/` for the basic arithmetic operations and `^` for exponentiation are valid for symbolic expressions as well:

```
a + b + c, a - b, -a, a*b*c, a/b, a^b
```

$$a + b + c,\ a - b,\ -a,\ a\,b\,c,\ \frac{a}{b},\ a^b$$

You may input these operators in the familiar mathematical way, but internally they are function calls:

[2] Remarkably, the MuPAD kernel treats not only genuine function calls, such as `sin(0.2)`, assignments, or arithmetical operations in a functional way, but also loops (Chapter 15) and case distinctions (Chapter 16).

[3] LISP programmers may disagree.

```
_plus(a,b,c), _subtract(a,b), _negate(a),
_mult(a,b,c), _divide(a,b), _power(a,b)
```

$$a + b + c,\ a - b,\ -a,\ a\,b\,c,\ \frac{a}{b},\ a^b$$

The same holds for the factorial of a nonnegative integer. You may input it in the mathematical notation `n!`. Internally it is converted to a call of the function `fact`:

```
n! = fact(n), fact(10)
```

$$n! = n!,\ 3628800$$

The arithmetic operators `div` and `mod`[4] were presented in Chapter 4.2. They may also be used in a symbolic context, but then return only symbolic results:

```
x div 4, 25 mod p
```

$$x \operatorname{div} 4,\ 25 \bmod p$$

Several MuPAD objects separated by commas form a sequence:

```
sequence := a, b, c + d
```

$$a,\ b,\ c + d$$

The operator `$` is an important tool to generate such sequences:

```
i^2 $ i = 2..7;  x^i $ i = 1..5
```

$$4,\ 9,\ 16,\ 25,\ 36,\ 49$$

$$x,\ x^2,\ x^3,\ x^4,\ x^5$$

Equations and inequalities are valid MuPAD objects. They are generated by the equality sign `=` and by `<>`, respectively:

```
equation := x + y = 2; inequality := x <> y
```

$$x + y = 2$$

$$x \neq y$$

[4] The object `x mod p` is converted to the function call `_mod(x, p)`. The function `_mod` can be redefined, e.g., `_mod:=modp` or `_mod:=mods`. The behavior of `modp` and `mods` is documented on the corresponding help pages. A redefinition of `_mod` also redefines the operator `mod`.

The operators <, <=, >, and >= compare the magnitudes of their arguments. The corresponding expressions represent conditions:

```
condition := i <= 2
```

$$i \leq 2$$

In a concrete context, they usually can be evaluated to one of the truth ("Boolean") values TRUE or FALSE. Typically, they are used in `if` statements or as termination conditions in loops. You may combine Boolean expressions via the logical operators `and` and `or`, or negate them via `not`:

```
condition3 := condition1 and (not condition2)
```

$$\text{condition1} \wedge \neg \text{condition2}$$

You can define maps (functions) in several ways in MuPAD. The simplest method is to use the *arrow operator* `->` (the minus symbol followed by the "greater than" symbol):

```
f := x -> x^2
```

$$x \to x^2$$

After defining a function in this way, you may call it like a system function:

```
f(4), f(x + 1), f(y)
```

$$16,\ (x+1)^2,\ y^2$$

If you wish to define a function that first treats the right-hand side as a command to be performed, use the *double arrow operator* `-->`:

```
g1 := x -> int(f(x), x);
g2 := x --> int(f(x), x)
```

$$x \to \int f(x)\ \mathrm{d}x$$

$$x \to \frac{x^3}{3}$$

The composition of functions is defined by means of the *composition operator* `@`:

```
c := a@b: c(x)
```

$$a(b(x))$$

The *iteration operator* `@@` denotes iterated composition of a function with itself:

```
f := g@@4: f(x)
```

$$g(g(g(g(x))))$$

Some system functions, such as definite integration via `int` or the `$` operator, require a *range*. You generate a range by means of the operator `..`:

```
range := 0..1;  int(x, x = range)
```

$$0..1$$

$$\frac{1}{2}$$

Ranges should not be confused with floating-point intervals of domain type `DOM_INTERVAL`, which may be created via the operator `...` or the function `hull`:

```
PI ... 20/3, hull(PI, 20/3)
```

$$3.141592653 \ldots 6.666666667,\ 3.141592653 \ldots 6.666666667$$

This data type is explained in Section 4.18.

MuPAD treats any expression of the form `identifier(argument)` as a function call:

```
delete f:
expression := sin(x) + f(x, y) + int(g(x), x = 0..1)
```

$$\sin(x) + f(x, y) + \int_0^1 g(x)\ \mathrm{d}x$$

Table 4.3 lists all operators presented above together with their equivalent functional form. You may use either form to input expressions:

```
2/14 = _divide(2, 14);
```

$$\frac{1}{7} = \frac{1}{7}$$

operator	system function	meaning	example
+	_plus	addition	SUM:= a+b
-	_subtract	subtraction	Difference:= a-b
*	_mult	multiplication	Product:= a*b
/	_divide	division	Quotient:= a/b
^	_power	exponentiation	Power:= a^b
div	_div	quotient modulo p	Quotient:= a div p
mod	_mod	remainder modulo p	Remainder:= a mod p
!	fact	factorial	n!
$	_seqgen	sequence generation	Sequence:= i^2$i=3..5
,	_exprseq	sequence concatenation	Seq:= Seq1, Seq2
union	_union	set union	S:= Set1 union Set2
intersect	_intersect	set intersection	S:= Set1 intersect Set2
minus	_minus	set difference	S:= Set1 minus Set2
=	_equal	equation	Equation:= x+y=2
<>	_unequal	inequality	Condition:= x<>y
<	_less	comparison	Condition:=a<b
>		comparison	Condition:=a>b
<=	_leequal	comparison	Condition:=a<=b
>=		comparison	Condition:=a>=b
not	_not	negation	Condition2:=not Condition1
and	_and	logical 'and'	Condition:=a<b and b<c
or	_or	logical 'or'	Condition:=a<b or b<c
->		mapping	Square:= x -> x^2
'	D	differential operator	f'(x)
@	_fconcat	composition	h:= f@g
@@	_fnest	iteration	g:= f@@3
.	_concat	concatenation	NewName:= Name1.Name2
..	_range	range	Range:= a..b
...	interval	interval	IV:= 2.1...3.5
name()		function call	sin(1), f(x), reset()

Table 4.3: The main operators for generating MuPAD® expressions

```
[i $ i = 3..5] = [_seqgen(i, i, 3..5)];
```
$$[3,4,5] = [3,4,5]$$

```
a < b = _less(a, b);
```
$$(a < b) = (a < b)$$

```
(f@g)(x) = _fconcat(f, g)(x)
```
$$f(g(x)) = f(g(x))$$

We remark that some of the system functions such as `_plus`, `_mult`, `_union`, or `_concat` accept arbitrarily many arguments, while the corresponding operators are only binary:

```
_plus(a, b, u, v), _concat(a, b, u, v), _union()
```
$$a + b + u + v,\ \text{abuv},\ \emptyset$$

It is often useful to know and to use the functional form of the operators. For example, it is very efficient to form longer sums by applying `_plus` to many arguments. You may generate the argument sequence quickly by means of the sequence generator `$`:

```
_plus(1/i! $ i = 0..100): float(%)
```
$$2.718281828$$

The function `map` is a useful tool. It applies a function to all operands of a MuPAD object. For example:

```
map([x1, x2, x3], function, y, z)
```
$$[\text{function}(\text{x1}, y, z)\,, \text{function}(\text{x2}, y, z)\,, \text{function}(\text{x3}, y, z)]$$

If you want to apply operators via `map`, use their functional equivalent:

```
map([x1, x2, x3], _power, 5), map([f, g], _fnest, 5)
```
$$\left[\text{x1}^5, \text{x2}^5, \text{x3}^5\right],\ [f \circ f \circ f \circ f \circ f, g \circ g \circ g \circ g \circ g]$$

For the most common operators, namely `+`, `-`, `*`, `/`, `^`, `=`, `<>`, `<`, `>`, `<=`, `>=`, `==>`, and `<=>`, the corresponding functions can be accessed in the form `` `+` ``, `` `-` ``, `` `*` `` etc.:

```
map([1, 2, 3, 4], `*`, 3), map([1, 2, 3, 4], `^`, 2)
```

$$[3, 6, 9, 12],\ [1, 4, 9, 16]$$

Note that `-` is negation, not subtraction.

Some operations are invalid because they do not make sense mathematically:

```
3 and x
```

```
  Error: Illegal operand [_and]
```

The system function _and recognizes that the argument 3 cannot represent a Boolean value and issues an error message. However, MuPAD accepts a symbolic expression such as a and b with symbolic identifiers a, b. As soon as a and b get Boolean values, the expression can be evaluated to a Boolean value as well:

```
c := a and b: a := TRUE: b := TRUE: c
```

TRUE

The operators have different *priorities*, for example:

```
a.fact(3) means a.(fact(3)) and returns a6,
a.6^2     means (a.6)^2 and returns a6^2,
a*b^c     means a*(b^c),
a+b*c     means a+(b*c),
a+b mod c means (a+b) mod c,
a=b mod c means a=(b mod c),
a,b$3     means a,(b$3) and returns a,b,b,b.
```

If we denote the relation "is of lower priority than" by $\prec$, then we have:

$$\texttt{,} \prec \texttt{\$} \prec \texttt{=} \prec \texttt{mod} \prec \texttt{+} \prec \texttt{*} \prec \texttt{\^{}} \prec \texttt{.} \prec \text{function call.}$$

You find a complete list of the operators and their priorities in Section "Operators" of the MuPAD Quick Reference. Parentheses can always be used to enforce an evaluation order that differs from the standard priority of the operators:

```
1 + 1 mod 2,  1 + (1 mod 2)
```

$$0,\ 2$$

```
i := 2: x.i^2, x.(i^2)
```

$$x2^2,\ x4$$

```
u, v $ 3; (u, v) $ 3
```

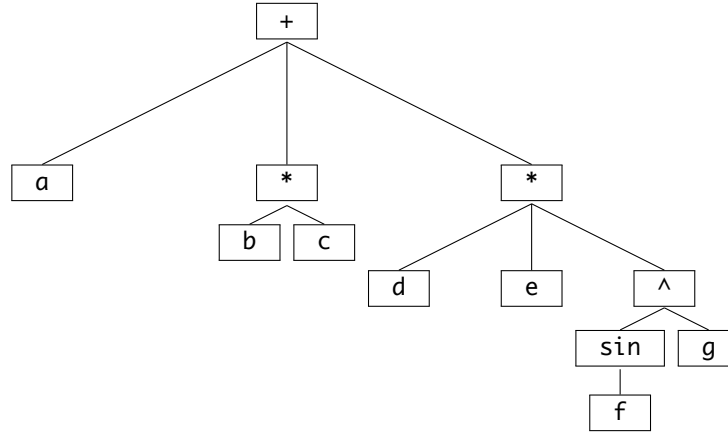
$$u,\ v,\ v,\ v$$

$$u,\ v,\ u,\ v,\ u,\ v$$

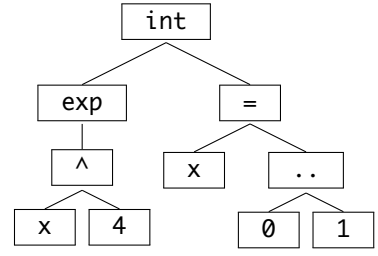
Expression Trees

A useful model for representing a MuPAD® expression is the *expression tree*. It reflects the internal representation. The operators or their corresponding functions, respectively, are the vertices, and the arguments are subtrees. The operator of lowest priority is at the root. Here are some examples:

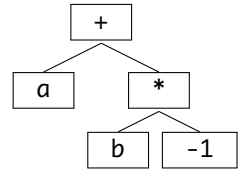
```
a + b * c + d * e * sin(f)^g
```



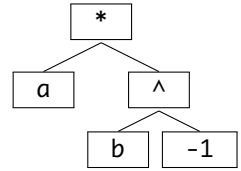

```
int(exp(x^4), x=0..1)
```



The difference `a - b` is internally represented as `a + b * (-1)`:



Similarly, a quotient `a/b` has the internal representation `a * b^(-1)`:



The leaves of an expression tree are MuPAD atoms.

The function `prog::exprtree` allows to display expression trees. Operators are replaced by the names of the corresponding system functions:

```
prog::exprtree(a/b):
    _mult
    |
    +-- a
    |
    `-- _power
        |
        +-- b
        |
        `-- -1
```

Exercise 4.8: Sketch the expression tree of `a^b - sin(a/b)`.

Exercise 4.9: Determine the operands of `2/3`, `x/3`, `1 + 2*I`, and `x + 2*I`. Explain the differences that you observe.

Operands

You can decompose expressions systematically by means of the operand functions op and nops, which were already presented in Section 4.1. The operands of an expression correspond to the subtrees below the root in the expression tree.

```
expression := a + b + c + sin(x): nops(expression)
```

$$4$$

```
op(expression)
```

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

Additionally, expressions of domain type DOM_EXPR have a "0th operand," which is accessible via op(·,0). It corresponds to the root of the expression tree and tells you which function is used to build the expression:

```
op(a + b*c, 0), op(a*b^c, 0), op(a^(b*c), 0),
op(f(sin(x)),0)
```

$$\text{\_plus},\ \text{\_mult},\ \text{\_power},\ f$$

```
sequence := a, b, c: op(sequence, 0)
```

$$\text{\_exprseq}$$

If the expression is a symbolic function call, op(·, 0) returns the identifier of that function:

```
op(sin(1), 0), op(f(x), 0), op(diff(y(x), x), 0)
```

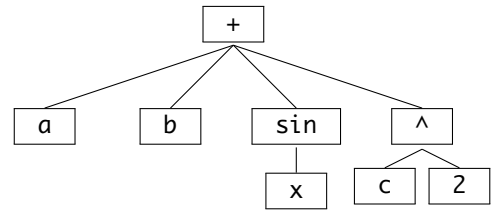
$$\sin,\ f,\ \text{diff}$$

You may regard the 0th operand of an expression as a "mathematical type." For example, an algorithm for differentiation of arbitrary expressions must find out whether the expression is a sum, a product, or a function call. To this end, it may look at the 0th operand to decide whether linearity, the product rule, or the chain rule of differentiation applies.

As an example, we decompose the expression

```
expression := a + b + sin(x) + c^2:
```

with the expression tree



systematically by means of the op function:

```
op(expression, 0..nops(expression))
```

$$\text{\_plus},\ a,\ b,\ \sin(x)\,,\ c^2$$

We can put these expressions together again in the following form:

```
root := op(expression, 0): operands := op(expression):
```

```
root(operands)
```

$$c^2 + a + b + \sin(x)$$

Note that the output order visible on the screen is different from the internal order accessible by op. You can access the building blocks of an expression in output order with the index operator [], as in the following:

```
expression[1], expression[4]
```

$$c^2,\ \sin(x)$$

In the following example, we decompose an expression completely into its atoms (compare with Section 4.1):

```
expression := sin(x + cos(a*b)):
```

The operands and subexpressions are:

```
op(expression, 0)            :  sin
op(expression, 1)            :  x+cos(a*b)
op(expression, [1, 0])       :  _plus
op(expression, [1, 1])       :  x
op(expression, [1, 2])       :  cos(a*b)
op(expression, [1, 2, 0])    :  cos
op(expression, [1, 2, 1])    :  a*b
op(expression, [1, 2, 1, 0]) :  _mult
op(expression, [1, 2, 1, 1]) :  a
op(expression, [1, 2, 1, 2]) :  b
```

Exercise 4.10: Sketch the expression tree of the following Boolean expression:

```
condition := (not a) and (b or c):
```

How can you use `op` to pick the symbolic identifiers `a`, `b`, and `c` out of the object `condition`?

Sequences

Sequences form an important MuPAD® data structure. Lists and sets are built from sequences. As discussed in Section 4.4, a sequence is a series of MuPAD objects separated by commas.

```
sequence1 := a, b, c; sequence2 := c, d, e
```

$$a,\ b,\ c$$

$$c,\ d,\ e$$

You may also use the comma to concatenate sequences:

```
sequence3 := sequence1, sequence2
```

$$a,\ b,\ c,\ c,\ d,\ e$$

Sequences are MuPAD expressions of domain type `DOM_EXPR`.

If m and n are integers, the call `object(i)$i=m..n` generates the sequence

$$object(m),\ object(m+1),\ \ldots,\ object(n):$$

```
i^2 $ i = 2..7, x^i $ i = 1..5
```

$$4,\ 9,\ 16,\ 25,\ 36,\ 49,\ x,\ x^2,\ x^3,\ x^4,\ x^5$$

The operator `$` is called the *sequence generator*. The equivalent functional form is `_seqgen(object(i),i,m..n)`

```
_seqgen(i^2, i, 2..7), _seqgen(x^i, i, 1..5)
```

$$4,\ 9,\ 16,\ 25,\ 36,\ 49,\ x,\ x^2,\ x^3,\ x^4,\ x^5$$

Usually, you will prefer the operator notation. The functional form is useful in connection with `map`, `zip` or similar functions.

You may use `$` in the following way to generate a sequence of successive integers:

```
$ 23..30
```

$$23,\ 24,\ 25,\ 26,\ 27,\ 28,\ 29,\ 30$$

The command `object $ n` returns a sequence of n identical objects:

```
x^2 $ 10
```

$$x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2$$

You can also use the sequence generator in connection with the keyword `in`. (The functional equivalent in this case in `_seqin`.) The loop variable then runs through all operands of the stated object:

```
f(x) $ x in [a, b, c, d]
```

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

```
f(x) $ x in a + b + c + d + sin(sqrt(2))
```

$$f(a),\ f(b),\ f(c),\ f(d),\ f\left(\sin\left(\sqrt{2}\right)\right)$$

It may be tempting to abuse the `$` operator for loops, as in these commands:

```
(x.i := sin(i); y.i := x.i) $ i=1..4:
```

For readability, memory consumption and therefore speed, we recommend against this usage. The assignments above should use a loop instead (cf. Chapter 15):

```
for i from 1 to 4 do
  x.i := sin(i);
  y.i := x.i;
end_for:
```

As a simple application of sequences, we now consider the MuPAD differentiator `diff`. The call `diff(f(x), x)` returns the derivative of f with respect to x. Higher derivatives are given by `diff(f(x), x, x)`, `diff(f(x), x, x, x)` etc. Thus, the 10-th derivative of $f(x) = \sin(x^2)$ can be computed conveniently by means of the sequence generator:

```
diff(sin(x^2), x $ 10)
```

$$30240\cos(x^2) - 403200\,x^4\cos(x^2) + 23040\,x^8\cos(x^2) - 302400\,x^2\sin(x^2) + 161280\,x^6\sin(x^2) - 1024\,x^{10}\sin(x^2)$$

The "void" object (Section 4.19) may be regarded as an empty sequence. You may generate it by calling `null()` or `_exprseq()`. The system automatically eliminates this object from sequences:

```
Seq := null(): Seq := Seq, a, b, null(), c
```

$$a,\ b,\ c$$

Some system functions such as the `print` command for screen output (Section 12.1) return the `null()` object:

```
sequence := a, b, print(Hello), c
```

```
 Hello
```

$$a,\ b,\ c$$

You can access the i-th entry of a sequence by `sequence[i]`. Redefinitions of the form `sequence[i]:=newvalue` are also possible:

```
F := a, b, c: F[2]
```

$$b$$

```
F[2] := newvalue: F
```

$$a,\ \text{newvalue},\ c$$

Alternatively, you may use the operand function `op` (Section 4.1) to access subsequences:[5]

```
F := a, b, c, d, e: op(F, 2); op(F, 2..4)
```

$$b$$

$$b,\ c,\ d$$

[5] Note that, in this example, the identifier `F` of the sequence is provided as argument to `op`. The `op` function regards a direct call of the form `op(a, b, c, d, e, 2)` as an (invalid) call with six arguments and issues an error message. You may use parentheses to avoid this error: `op((a, b, c, d, e), 2)`.

You may use `delete` to delete entries from a sequence, thus shortening the sequence:

```
F; delete F[2]: F; delete F[3]: F
```

$$a,\ b,\ c,\ d,\ e$$

$$a,\ c,\ d,\ e$$

$$a,\ c,\ e$$

The main usage of MuPAD sequences is the generation of lists and sets and supplying arguments to function calls. For example, the functions `max` and `min` can compute the maximum and minimum, respectively, of arbitrarily many arguments:

```
Seq := 1, 2, -1, 3, 0: max(Seq), min(Seq)
```

$$3,\ -1$$

Exercise 4.11: Assign the values $x_1 = 1, x_2 = 2, \ldots, x_{100} = 100$ to the identifiers $x_1, x_2, \ldots, x_{100}$.

Exercise 4.12: Generate the sequence

$$x_1, \underbrace{x_2, x_2}_{2}, \underbrace{x_3, x_3, x_3}_{3}, \ldots, \underbrace{x_{10}, x_{10}, \ldots, x_{10}}_{10}\ .$$

Exercise 4.13: Use a simple command to generate the double sum

$$\sum_{i=1}^{10} \sum_{j=1}^{i} \frac{1}{i+j}\ .$$

Hint: the function `_plus` accepts arbitrarily many arguments. Generate a suitable argument sequence.

Lists

A list is an ordered sequence of arbitrary MuPAD® objects enclosed in square brackets:

```
list := [a, 5, sin(x)^2 + 4, [a, b, c], hello,
         3/4, 3.9087]
```

$$\left[a, 5, \sin(x)^2 + 4, [a, b, c], \text{hello}, \frac{3}{4}, 3.9087\right]$$

A list may contain lists as elements. It may also be empty:

```
list := []
```

$$[]$$

The possibility to generate sequences via `$` is helpful for constructing lists:

```
sequence := i $ i = 1..10: list := [sequence]
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

```
list := [x^i $ i = 0..12]
```

$$\left[1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}, x^{11}, x^{12}\right]$$

A list may occur on the left hand side of an assignment. This may be used to assign values to several identifiers simultaneously:

```
[A, B, C] := [a, b, c]: A + B^C
```

$$a + b^c$$

A useful property of this construction is that all the assignments are performed at the same time, so you can swap values:

```
a := 1: b:= 2: a, b
```

$$1,\ 2$$

```
[a, b] := [b, a]: a, b
```

$$2,\ 1$$

The function nops returns the number of elements of a list. You can access the elements of a list by means of indexed access: `list[i]` returns the i-th list element, and `list[i..j]` returns the sub-list starting at the i-th and extending up to and including the j-th element. Negative values for i and j are counted from the end:

```
list := [a, b, c, d, e, f]: list[1], list[3..-2]
```

$$2,\ [c, d, e]$$

As with almost any MuPAD object, you can also access the elements of a list with the op function: `op(list)` returns the sequence of elements, i.e., the sequence that has been used to construct the list by enclosing it in square brackets `[]`. The call `op(list, i)` returns the i-th list element, and `op(list, i..j)` extracts the sequence of the i-th up to the j-th list element:

```
delete a, b, c: list := [a, b, sin(x), c]: op(list)
```

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

```
op(list, 2..3)
```

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

You may change a list element by an indexed assignment:

```
list := [a, b, c]: list[1] := newvalue: list
```

$$[\mathrm{newvalue}, b, c]$$

Writing to a sublist may change the length of the list:

```
list[2..3] := [d,e,f,g]: list
```

$$[\mathrm{newvalue}, d, e, f, g]$$

Alternatively, the command `subsop(list, i=newvalue)` (Chapter 6) returns a copy with the i-th operand redefined:

```
list := [a, b, c]: list2 := subsop(list, 1 = newvalue)
```

$$[\mathrm{newvalue}, b, c]$$

Caution: If `L` is an identifier without a value, then the indexed assignment

```
L[index] := value:
```

generates a table (Section 4.8) and not a list:

```
delete L: L[1] := a: L
```

1	a

You can remove elements from a list by using `delete`. This shortens the list:

```
list := [a, b, c]: delete list[1]: list
```

$$[b, c]$$

The function `contains` checks whether a MuPAD object belongs to a list. It returns the index of (the first occurrence of) the element in the list. If the list does not contain the element, then `contains` returns the integer 0:

```
contains([x + 1, a, x + 1], x + 1)
```

$$1$$

```
contains([sin(a), b, c], a)
```

$$0$$

The function `append` adds elements to the tail of a list:

```
list := [a, b, c]: append(list, 3, 4, 5)
```

$$[a, b, c, 3, 4, 5]$$

The dot operator `.` concatenates lists:

```
list1 := [1, 2, 3]: list2 := [4, 5, 6]:
```

```
list1.list2, list2.list1
```

$$[1, 2, 3, 4, 5, 6]\,,\ [4, 5, 6, 1, 2, 3]$$

The corresponding system function is `_concat` and accepts arbitrarily many arguments. You can use it to combine many lists:

```
_concat(list1 $ 5)
```

$$[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]$$

A list can be sorted by means of the function `sort`. This arranges numerical values according to their magnitude, strings (Section 4.11) are sorted lexicographically:

```
sort([-1.23, 4, 3, 2, 1/2])
```

$$\left[-1.23, \frac{1}{2}, 2, 3, 4\right]$$

```
sort(["A", "b", "a", "c", "C", "c", "B", "a1", "abc"])
```

["A", "B", "C", "a", "a1", "abc", "b", "c", "c"]

```
sort(["x10002", "x10011", "x10003"])
```

["x10002", "x10003", "x10011"]

Note that the lexicographical order only applies to strings generated with `"`. Names of identifiers are sorted according to different (internal) rules, which only reduce to lexicographic order for short identifiers (and may change in future releases):

```
delete A, B, C, a, b, c, a1, abc:
sort([A, b, a, c, C, c, B, a1, abc])
```

$[A, B, C, a, \mathrm{a1}, \mathrm{abc}, b, c, c]$

```
sort([x10002, x10011, x10003])
```

$[\mathrm{x10002}, \mathrm{x10011}, \mathrm{x10003}]$

MuPAD regards lists of function names as list-valued functions:

```
[sin, cos, f](x)
```

$[\sin(x), \cos(x), f(x)]$

The function `map` applies a function to all elements of a list:

```
map([x, 1, 0, PI, 0.3], sin)
```

$[\sin(x), \sin(1), 0, 0, 0.2955202067]$

If the function has more than one argument, then `map` substitutes the list elements for the first argument and takes the remaining arguments from its own argument list:

```
map([a, b, c], f, y, z)
```

$$[f(a, y, z), f(b, y, z), f(c, y, z)]$$

This `map` construction is a powerful tool for handling lists as well as other MuPAD objects. In the following example, we have a nested list L. We want to extract the first elements of the sublists using `op(·, 1)`. This is easily done using `map`:

```
L := [[a1, b1], [a2, b2], [a3, b3]]: map(L, op, 1)
```

$$[a1, a2, a3]$$

The MuPAD function `select` enables you to extract elements with a certain property from a list. To this end, you need a function that checks whether an object has this property and returns `TRUE` or `FALSE`. For example, the call `has(object1, object2)` returns `TRUE` if `object2` is an operand or suboperand of `object1`, and `FALSE` otherwise:

```
has(1 + sin(1 + x), x), has(1 + sin(1 + x), y)
```

$$\text{TRUE},\ \text{FALSE}$$

Now,

```
select([a + 2, x, y, z, sin(a)], has, a)
```

$$[a + 2, \sin(a)]$$

extracts all those list elements for which `has(·,a)` returns `TRUE`, i.e., those which contain the identifier `a`.

The function `split` divides a list into three lists, as follows. The first list contains all elements with a certain property, the second list collects all those elements without the property. If the test for the property returns the value `UNKNOWN` for some elements, then these are put into the third list. Otherwise, the third list is empty:

```
split([sin(x), x^2, y, 11], has, x)
```

$$[[\sin(x), x^2], [y, 11], []]$$

The MuPAD function `zip` combines elements of two lists pairwise into a new list:

```
L1 := [a, b, c]: L2 := [d, e, f]:
```

```
zip(L1, L2, _plus), zip(L1, L2, _mult),
zip(L1, L2, _power)
```

$$[a+d, b+e, c+f],\ [a\,d, b\,e, c\,f],\ \left[a^d, b^e, c^f\right]$$

The third argument of `zip` must be a function that takes two arguments. This function is then applied to the pairs of list elements. In the above example, we have used the MuPAD functions `_plus`, `_mult`, and `_power` for addition, multiplication, and exponentiation, respectively. If the two input lists have different lengths, then the behavior of `zip` depends on the optional fourth argument. If this is not present, then the length of the resulting list is the minimum of the lengths of the two input lists. Otherwise, if you supply an additional fourth argument, then `zip` replaces the "missing" list entries by this argument:

```
L1 := [a, b, c, 1, 2]: L2 := [d, e, f]:
```

```
zip(L1, L2, _plus)
```

$$[a+d, b+e, c+f]$$

```
zip(L1, L2, _plus, hello)
```

$$[a+d, b+e, c+f, \text{hello}+1, \text{hello}+2]$$

Table 4.4 gives a summary of all list operations that we have discussed.

Exercise 4.14: Generate two lists with the elements a, b, c, d and $1, 2, 3, 4$, respectively. Concatenate the lists. Multiply the lists pairwise.

Exercise 4.15: Multiply all entries of the list `[1, x, 2]` by 2. Suppose you are given a list, whose elements are lists of numbers or expressions, such as `[[1, x, 2], [PI], [2/3, 1]]`, how can you multiply all entries by 2?

`.` or `_concat`	:	concatenating lists
`append`	:	appending elements
`contains(list,x)`	:	does `list` contain the element `x`?
`list[i]`	:	accessing the i-th element
`map`	:	applying a function
`nops`	:	length
`op`	:	accessing elements
`select`	:	select according to properties
`sort`	:	sorting
`split`	:	split according to properties
`subsop`	:	replacing elements
`delete`	:	deleting elements
`zip`	:	combining two lists

Table 4.4: MuPAD® functions and operators for lists

Exercise 4.16: Let $X = [x_1, \ldots, x_n]$ and $Y = [y_1, \ldots, y_n]$ be two lists of the same length. Find a simple method to compute

- their "inner product" (X as row vector and Y as column vector)

$$x_1\, y_1 + \cdots + x_n\, y_n,$$

- their "matrix product" (X as column vector and Y as row vector)

$$[\,[x_1\, y_1,\ x_1\, y_2, \ldots, x_1\, y_n],\ [x_2\, y_1,\ x_2\, y_2, \ldots, x_2\, y_n],\\ \ldots,\ [x_n\, y_1,\ x_n\, y_2, \ldots, x_n\, y_n]\,]\,.$$

You can achieve this by using `zip`, `_plus`, `map` and appropriate functions (Section 4.12) within a single command line in each case. Loops (Chapter 15) are not required.

Exercise 4.17: In number theory, one is often interested in the density of prime numbers in sequences of the form $f(1), f(2), \ldots$, where f is a polynomial. For each value of $m = 0, 1, \ldots, 41$, find out how many of the integers $n^2 + n + m$ with $n = 1, 2, \ldots, 100$ are primes.

Exercise 4.18: In which ordering will n children be selected for "removal" by a counting-out rhyme composed of m words? For example, using the rhyme

"eenie–meenie–miney–moe–catch a–tiger–by the–toe"

with 8 "words," 12 children are counted out in the order 8–4–1–11–10–12–3–7–6–2–9–5. Hint: represent the children by a list `[1, 2, ...]` and remove an element from this list after it is counted out.

Sets

A *set* is an unordered sequence of arbitrary objects enclosed in curly braces. Sets are of domain type DOM_SET:

```
{34, 1, 89, x, -9, 8}
```
$$\{-9, 1, 8, 34, 89, x\}$$

The order of the elements in a MuPAD® list seems to be random. The MuPAD kernel sorts the elements according to internal principles. You should use sets only if the order of the elements does not matter. If you want to process a sequence of expressions in a certain order, use lists as discussed in the previous section.

Sets may be empty:

```
emptyset := {}
```
$$\emptyset$$

A set contains each element only once, i.e., duplicate elements are removed automatically:

```
set := {a, 1, 2, 3, 4, a, b, 1, 2, a}
```
$$\{1, 2, 3, 4, a, b\}$$

The function nops determines the number of elements in a set. As for sequences and lists, op extracts elements from a set:

```
op(set)
```
$$a,\ 1,\ 2,\ 3,\ 4,\ b$$

```
op(set, 2..4)
```
$$1,\ 2,\ 3$$

Warning: Since elements of a set may be reordered internally, you should check carefully whether it makes sense to access the i-th element. For example, subsop(set, i = newvalue) (Section 6) replaces the i-th element by a new value. However, you should check in advance (using op) that the element that you want to replace really is the i-th element. Especially note that replacing an element of a set often reorders other entries.

The command `op(set, i)` returns the i-th element of `set` in the internal order, which usually is different from the i-th element of `set` that you see on the screen. However, you can access elements by using `set[i]`, where the returned elements is the i-th element as printed on the screen.

The functions `union`, `intersect`, and `minus` form the union, the intersection, and the set-theoretic difference, respectively, of sets:

```
M1 := {1, 2, 3, a, b}: M2 := {a, b, c, 4, 5}:
```

```
M1 union M2, M1 intersect M2, M1 minus M2, M2 minus M1
```

$$\{1, 2, 3, 4, 5, a, b, c\},\ \{a, b\},\ \{1, 2, 3\},\ \{4, 5, c\}$$

In particular, you can use `minus` to remove elements from a set:

```
{1, 2, 3, a, b} minus {3, a}
```

$$\{1, 2, b\}$$

You can also replace an element by a new value without caring about the order of the elements:

```
set := {a, b, oldvalue, c, d}
```

$$\{a, b, c, d, \text{oldvalue}\}$$

```
set minus {oldvalue} union {newvalue}
```

$$\{a, b, c, d, \text{newvalue}\}$$

The function `contains` checks whether an element belongs to a set, and returns either `TRUE` or `FALSE`:[6]

```
contains({a, b, c}, a), contains({a, b, c + d}, c)
```

$$\text{TRUE},\ \text{FALSE}$$

MuPAD regards sets of function names as set-valued functions:

```
{sin, cos, f}(x)
```

$$\{\cos(x), f(x), \sin(x)\}$$

[6] Note the difference to the behavior of `contains` for lists: there the ordering of the elements is determined when you generate the list, and `contains` returns the position of the element in the list.

`contains(M, x)`	:	does `M` contain the element `x`?
`intersect`	:	intersection
`map`	:	applying a function
`minus`	:	set-theoretic difference
`nops`	:	number of elements
`op`	:	accessing elements
`select`	:	select according to properties
`split`	:	split according to properties
`subsop`	:	replacing elements
`union`	:	set-theoretic union

Table 4.5: MuPAD® functions and operators for sets

You can apply a function to all elements of a set by means of `map`:

```
map({x, 1, 0,  PI, 0.3}, sin)
```

$$\{0, 0.2955202067, \sin(1), \sin(x)\}$$

You can use the function `select` to extract elements with a certain property from a set. This works as for lists, but the returned object is a set:

```
select({{a, x, b}, {a}, {x, 1}}, contains, x)
```

$$\{\{1, x\}, \{a, b, x\}\}$$

Similarly, you can use the function `split` to divide a set into three subsets of elements with a certain property, elements without that property, and elements for which the system cannot decide this and returns `UNKNOWN`. The result is a list comprising these three sets:

```
split({{a, x, b}, {a}, {x, 1}}, contains, x)
```

$$[\{\{1, x\}, \{a, b, x\}\}, \{\{a\}\}, \emptyset]$$

Table 4.5 contains a summary of the set operations discussed so far.

MuPAD also provides the data structure `Dom::ImageSet` for handling infinite sets; see page 8-9 in Chapter 8.

Exercise 4.19: How can you convert a list to a set and vice versa?

Exercise 4.20: Generate the sets $A = \{a, b, c\}$, $B = \{b, c, d\}$, and $C = \{b, c, e\}$. Compute the union and the intersection of the three sets, as well as the difference $A \setminus (B \cup C)$.

Exercise 4.21: Instead of the binary operators `intersect` and `union`, you can also use the corresponding MuPAD functions `_intersect` and `_union` to compute unions and intersections of sets. These functions accept arbitrarily many arguments. Use simple commands to compute the union and the intersection of all sets belonging to `M`:

```
M := {{2, 3}, {3, 4}, {3, 7}, {5, 3}, {1, 2, 3, 4}}:
```

Tables

A table is a MuPAD® object of domain type `DOM_TABLE`. It is a collection of equations of the form `index=value`. Both indices and values may be arbitrary MuPAD objects. You can generate a table by using the system function `table` ("explicit generation"):

```
T := table(a = b, c = d)
```

a	b
c	d

You can generate more entries or change existing ones by "indexed assignments" of the form `Table[index]:=value`:

```
T[f(x)] := sin(x): T[1, 2] := 5:
```

```
T[1, 2, 3] := {a, b, c}: T[a] := B:
```

```
T
```

a	B
c	d
$f(x)$	$\sin(x)$
$1, 2$	5
$1, 2, 3$	$\{a, b, c\}$

It is not necessary to initialize a table via `table`. If `T` is an identifier that does not have a value, or has a value which does not allow indexed access (such as an integer or a function), then an indexed assignment of the form `T[index]:=value` automatically turns `T` into a table ("implicit generation"):

```
delete T: T[a] := b: T[b] := c: T
```

a	b
b	c

A table may be empty:

```
T := table()
```

You may delete table entries via `delete Table[index]`:

```
T := table(a = b, c = d, d = a*c):
```

```
delete T[a], T[c]: T
```

$$\begin{array}{c|c} d & a\,c \end{array}$$

You can access table entries in the form `Table[index]`; this returns the element corresponding to the index. If there is no entry for the index, then MuPAD returns `Table[index]` symbolically:

```
T := table(a = b, c = d, d = a*c):
```

```
T[a], T[b], T[c], T[d]
```

$$b,\ T_b,\ d,\ a\,c$$

The call `op(Table)` returns all entries of a table, i.e., the sequence of all equations `index=value`:

```
op(table(a = A, b = B, c = C, d = D))
```

$$a = A,\ b = B,\ c = C,\ d = D$$

Note that the internal order of the table entries may differ both from the order in which you have generated the table and from the order of entries printed on the screen. It may look quite random:

```
op(table(a.i = i^2 $ i = 1..17))
  a16 = 256, a9 = 81, a12 = 144, a6 = 36, a3 = 9,

     a15 = 225, a11 = 121, a8 = 64, a5 = 25, a2 = 4,

     a14 = 196, a10 = 100, a17 = 289, a7 = 49, a4 = 16,

     a13 = 169, a1 = 1
```

The function `map` applies a given function to the *values* (not the *indices*) of all table entries:

```
T := table(1 = PI, 2 = 4, 3 = exp(1)): map(T, float)
```

1	3.141592654
2	4.0
3	2.718281828

The function `contains` checks whether a table contains a particular *index*. It ignores the *values*:

```
T := table(a = b): contains(T, a), contains(T, b)
```

TRUE, FALSE

You may employ the functions `select` and `split` to inspect *both indices and values* of a table and to extract them according to certain properties. This works similarly as for lists (Section 4.6) and for sets (Section 4.7):

```
T := table(1 = "number", 1.0 = "number", x = "symbol"):
```

```
select(T, has, "symbol")
```

x	"symbol"

```
select(T, has, 1.0)
```

1.0	"number"

```
split(T, has, "number")
```

1	"number"
1.0	"number"

, | x | "symbol" |

Tables are particularly well suited for storing large amounts of data. Indexed accesses to *individual* elements are implemented efficiently also for big tables: a write or read does not file through the whole data structure.

Exercise 4.22: Generate a table `telephoneDirectory` with the following entries:

$$\text{Ford } 1815,\ \text{Reagan } 4711,\ \text{Bush } 1234,\ \text{Clinton } 5678.$$

Look up Ford's number. How can you find out whose number is 5678?

Exercise 4.23: Given a table, how can you generate a list of all indices and a list of all values, respectively?

Exercise 4.24: Generate the table `table(1=1, 2=2, …, n=n)` and the list `[1, 2, …, n]` of length $n = 100000$. Add a new entry to the table and to the list. How long does this take? Hint: the call `time((a:=b))` returns the execution time for an assignment.

Arrays

Arrays, of domain type DOM_ARRAY, may be regarded as special tables. You may think of them as a collection of equations of the form index = value, but in contrast to tables, the indices must be integers. A one-dimensional array consists of equations of the form i = value. Mathematically, it represents a vector whose i-th component is value. A two-dimensional array represents a matrix, whose (i, j)-th component is stored in the form (i, j) = value. You may generate arrays of arbitrary dimension, with entries of the form (i, j, k, …) = value.

The system function *array* generates arrays. In its simplest form, you only specify a sequence of ranges that determine the dimension and the size of the array:

```
A := array(0..1, 1..3)
```

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

You can see here that the first range 0..1 and the second range 1..3 determine the array's number of rows and columns, respectively. The output ?[0, 1] signals that the corresponding index has not been assigned a value, yet. Thus, the above command has generated an empty array. Now, you can assign values to the indices:

```
A[0, 1] := 1: A[0, 2] := 2: A[0, 3] := 3:
```

```
A[1, 3] := HELLO: A
```

$$\begin{pmatrix} 1 & 2 & 3 \\ \text{NIL} & \text{NIL} & \text{HELLO} \end{pmatrix}$$

You can also initialize the complete array directly when generating it by *array*. Just supply the values as a (nested) list:

```
A := array(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

You can access and modify array elements in the same way as table elements:

```
A[2, 3] := A[2, 3] + 10: A
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 16 \end{pmatrix}$$

Again, you may delete an array element by using `delete`:

```
delete A[1, 1], A[2, 3]:  A, A[2, 3]
```

$$\begin{pmatrix} \text{NIL} & 2 & 3 \\ 4 & 5 & \text{NIL} \end{pmatrix},\ A_{2,3}$$

Arrays possess a "0-th operand" `op(Array, 0)` providing information on the dimension and the size of the array. The call `op(Array, 0)` returns a sequence d, $a_1..b_1, \ldots, a_d..b_d$, where d is the dimension (i.e., the number of indices) and $a_i..b_i$ is the valid range for the i-th index:

```
Vec := array(1..3, [x, y, z]): op(Vec, 0)
```

$$1,\ 1..3$$

```
Mat := array(1..2, 1..3, [[a, b, c], [d, e, f]]):
```

```
op(Mat, 0)
```

$$2,\ 1..2,\ 1..3$$

Thus, the dimension of an $m \times n$ matrix `array(1..m, 1..n)` may be obtained as

`m=op(Mat, [0, 2, 2])`, `n=op(Mat, [0, 3, 2])`.

The internal structure of arrays differs from the structure of tables. The entries are not stored in the form of equations:

```
op(Mat)
```

$$a,\ b,\ c,\ d,\ e,\ f$$

The table type is more flexible than the array type: tables admit arbitrary indices, and their size may grow dynamically. Arrays are intended for storing vectors and matrices of a fixed size. When you enter an indexed call, the system checks whether the indices are within the specified ranges. For example:

```
Mat[4, 7]
  Error: Illegal argument [array]
```

You may apply a function to all array components via `map`. For example, here is the simplest way to convert all array entries to floating-point numbers:

```
A := array(1..2, [PI, 1/7]): map(A, float)
```

$$\begin{pmatrix} 3.141592654 & 0.1428571429 \end{pmatrix}$$

Warning: If `M` is an identifier without a value, then an indexed assignment of the form `M[index, index, ...]:=value` generates a table and not an array of type `DOM_ARRAY` (Section 4.8):

```
M[1, 1] := a: M
```

$1, 1$	a

Additionally, MuPAD provides the more powerful data structures of domain type `Dom::Matrix` for handling vectors and matrices. These are discussed in Section 4.15. Such objects are very convenient to use: you can multiply two matrices or a matrix and a vector by means of the usual multiplication symbol `*`. Similarly, you can add matrices of equal dimension via `+`. To achieve the same functionality with arrays, you have to write your own procedures. We refer to the examples `MatrixProduct` and `MatrixMult` in Sections 17.4 and 17.5, respectively.

Exercise 4.25: Generate a so-called Hilbert matrix H of dimension 20×20 with entries $H_{ij} = 1/(i + j - 1)$.

Boolean Expressions

MuPAD® implements three logical ("Boolean") values: TRUE, FALSE, and UNKNOWN:

```
domtype(TRUE), domtype(FALSE), domtype(UNKNOWN)
```

$$\text{DOM\_BOOL},\ \text{DOM\_BOOL},\ \text{DOM\_BOOL}$$

The operators *and*, or, and not operate on Boolean values:

```
TRUE and FALSE, not (TRUE or FALSE), TRUE and UNKNOWN,
TRUE or UNKNOWN
```

$$\text{FALSE},\ \text{FALSE},\ \text{UNKNOWN},\ \text{TRUE}$$

The functions corresponding to these operators are _and, _or, and _not, respectively.

The function bool evaluates equations, inequalities, or comparisons via >, >=, < , <=, to TRUE or FALSE:

```
bool(1 = 2), bool(1 <> 2),
bool(1 <= 2) or not bool(1 > 2)
```

$$\text{FALSE},\ \text{TRUE},\ \text{TRUE}$$

Note that bool can only compare real numbers sufficiently distinct from one another. Especially exact symbolic representations of the same number cannot be compared with bool:

```
bool(3 <= PI)
```

$$\text{TRUE}$$

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

```
  Error: Can't evaluate to boolean [_leequal]
```

```
bool(14885392687/4738167652 <= PI)
```

```
  Error: Can't evaluate to boolean [_leequal]
```

Typically, you will use Boolean constructs in branching conditions of `if` instructions (Chapter 16) or in termination conditions of `repeat` loops (Chapter 15). In the following example, we test the integers $1, 2, 3$ for primality. The system function `isprime` ("is the argument a prime number?") returns TRUE or FALSE. The `repeat` loop stops as soon as the termination condition $i = 3$ evaluates to TRUE:

```
i := 0:
repeat
  i := i + 1;
  if isprime(i)
     then print(i, "is a prime")
     else print(i, "is no prime")
  end_if
until i = 3 end_repeat
```

```
    1, "is no prime"

    2, "is a prime"

    3, "is a prime"
```

Here we have used strings enclosed in " for the screen output. They are discussed in detail in Section 4.11. Note that it is not necessary to use the function `bool` in branching or termination conditions in order to evaluate the condition to `TRUE` or `FALSE`.

Exercise 4.26: Let $\wedge$ denote the logical "and," let $\vee$ denote the logical "or," let $\neg$ denote logical negation. To which Boolean value does

$$\text{TRUE} \wedge (\text{FALSE} \vee \neg (\text{FALSE} \vee \neg \text{FALSE}))$$

evaluate?

Exercise 4.27: Let `L1`, `L2` be two MuPAD lists of equal length. How can you find out whether `L1[i]<L2[i]` holds true for all list elements?

Strings

Strings are pieces of text, which may be used for formatted screen output. A string is a sequence of arbitrary symbols enclosed in “string delimiters” `"`. Its domain type is `DOM_STRING`.

```
string1 := "Use * for multiplication";
string2 := ", ";
string3 := "use ^ for exponentiation."
```

”Use * for multiplication”

”, ”

”use ^ for exponentiation.”

The concatenation operator `.` combines strings:

```
string4 := string1.string2.string3
```

”Use * for multiplication, use ^ for exponentiation.”

The dot operator is a short form of the MuPAD® function `_concat`, which concatenates (arbitrarily many) strings:

```
_concat("This is ", "a string", ".")
```

”This is a string.”

The index operator `[]` extracts the characters from a string:

```
string4[1], string4[2], string4[3],
string4[4], string4[5]
```

”U”, ”s”, ”e”, ” ”, ”*”

When using a range as index, substrings are returned; negative indices count from the end of the string:

```
string4[19..21], string4[-15..-8]
```

”cat”, ”exponent”

You may use the command `print` to output intermediate results in loops or procedures on the screen (Section 12.1). By default, this function prints strings

with the enclosing double quotes. You may change this behavior by using the option Unquoted:

```
print(string4)
    "Use * for multiplication, use ^ for exponentiation."
print(Unquoted, string4)
    Use * for multiplication, use ^ for exponentiation.
```

Strings are not valid identifiers in MuPAD, so you cannot assign values to them:

```
"name" := sin(x)
  Error: Invalid left-hand side in assignment [line 1, col 8]
```

Also, arithmetic with strings is not allowed:

```
1 + "x"
  Error: Illegal operand [_plus]
```

However, you may use strings in equations:

```
"derivative of sin(x)" = cos(x)
```

$$\text{"derivative of sin(x)"} = \cos(x)$$

The function expr2text converts a MuPAD object to a string. You can employ this function to customize print commands:

```
i := 7:
print(Unquoted, expr2text(i)." is a prime.")
    7 is a prime.
a := sin(x):
print(Unquoted, "The derivative of " . expr2text(a) .
                " is " . expr2text(diff(a, x)). ".")
    The derivative of sin(x) is cos(x).
```

The documentation of print contains more advanced examples of user-defined output, see ?print.

You find numerous other useful functions for handling strings in the standard library (Section “Manipulation of Strings” of the MuPAD Quick Reference) and in the string library (?stringlib).

Exercise 4.28: The command `anames(All)` returns a set of all identifiers that have a value in the current session. Generate a *lexicographically* ordered list of these identifiers.

Exercise 4.29: How can you obtain the "mirror image" of a string? Hint: the function `length` returns the number of symbols in a string.

Functions

The arrow operators `->` (a minus sign followed by a "greater than" sign) and `-->` (two minus signs followed by a "greater than" sign) generate simple objects that represent mathematical functions:

```
f := (x, y) -> x^2 + y^2
```

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

The function `f` can now be called like any system function. It takes two arbitrary input parameters (or "arguments") and returns the sum of their squares:

```
f(a, b + 1)
```

$$(b + 1)^2 + a^2$$

In the following example, the return value of the function is generated by an `if` statement:

```
absValue := x -> if x >= 0 then x else -x end_if:
```

```
absValue(-2.3)
```

$$2.3$$

As discussed in Section 4.4, the operator `@` generates the composition $h : x \to f(g(x))$ of two functions f and g:

```
f := x -> 1/(1 + x): g := x -> sin(x^2):
```

```
h := f@g: h(a)
```

$$\frac{1}{\sin(a^2) + 1}$$

You can define a repeated composition $f(f(f(\cdot)))$ of a function with itself by using the iteration operator `@@`:

```
fff := f@@3: fff(a)
```

$$\frac{1}{\frac{1}{\frac{1}{a+1}+1}+1}$$

Of course, these constructions also work for system functions. For example, the function `abs@Re` computes the absolute value of the real part of a complex number:

```
f := abs@Re: f(-2 + 3*I)
```

$$2$$

In symbolic computations, you often have the choice to represent a mathematical function either as a *map* `arguments` $\mapsto$ `value` or as an *expression*:

```
Map := x -> 2*x*cos(x^2):
```

```
Expression := 2*x*cos(x^2):
```

```
int(Map(x), x), int(Expression, x)
```

$$\sin\left(x^2\right),\ \sin\left(x^2\right)$$

If you try to convert an expression into a function by means of the `->` operator, you will find that it does not quite work:

```
h := x -> Expression:
h(1)
```

$$2\,x\,\cos\left(x^2\right)$$

Instead, use the long arrow operator `-->`, which evaluates the right hand side expression, as explained in Chapter 5 and returns a function, which you can manipulate further:

```
h := x --> Expression;
```

$$x \to 2\,x\,\cos\left(x^2\right)$$

```
h'
```

$$x \to 2\,\cos\left(x^2\right) - 4\,x^2\,\sin\left(x^2\right)$$

Indeed, `h'` is the functional equivalent of `diff(Expression, x)`:

```
h'(x) = diff(Expression, x)
```

$$2\,\cos\left(x^2\right) - 4\,x^2\,\sin\left(x^2\right) = 2\,\cos\left(x^2\right) - 4\,x^2\,\sin\left(x^2\right)$$

MuPAD can represent maps by means of *functional expressions*: functions are constructed from simple functions (such as `sin`, `cos`, `exp`, `ln`, `id`) by means of operators (such as the composition operator `@` or the arithmetic operators `+`, `*`, etc.). Note that the arithmetic operators generate functions that are defined *pointwise*, which is mathematically sound. For example, $f + g$ represents the map $x \to f(x) + g(x)$, $f \cdot g$ represents the map $x \to f(x) \cdot g(x)$, etc.:

```
delete f, g:
```

```
a := f + g: b := f*g: c := f/g:
a(x), b(x), c(x)
```

$$f(x) + g(x),\ f(x)\ g(x),\ \frac{f(x)}{g(x)}$$

You are allowed to have numerical values in functional expressions. MuPAD regards them as constant functions which always return the particular value:

```
1(x), 0.1(x, y, z), PI(x)
```

$$1,\ 0.1,\ \pi$$

```
a := f + 1: b := f*3/4: c := f + 0.1: d := f + sqrt(2):
```

```
a(x), b(x), c(x), d(x)
```

$$f(x) + 1,\ \frac{3\,f(x)}{4},\ f(x) + 0.1,\ f(x) + \sqrt{2}$$

It is also possible to use functions without assigning them to an identifier and to pass them as arguments to other functions. We have used this possibility before, so for recapitulation, we just give another simple example:

```
map([1, 2, 3, 4], i -> i^2)
```

$$[1, 4, 9, 16]$$

The operator `->` is useful for defining functions whose return value can be obtained by simple operations. Functions implementing more complex algorithms usually require many commands and auxiliary variables to store intermediate results. In principle, you can define such functions via `->` as well. However, this has the drawback that you often use *global variables*. Instead, we recommend to define a procedure via `proc() begin ... end_proc`. This concept of the MuPAD programming language is much more flexible and is discussed in

more detail in Chapter 17.

Exercise 4.30: Define the functions $f(x) = x^2$ and $g(x) = \sqrt{x}$. Compute $f(f(g(2))$ and $\underbrace{f(f(\ldots f}_{\text{100 times}}(x)\ldots))$.

Exercise 4.31: Define a function that reverses the order of the elements in a list.

Exercise 4.32: The *Chebyshev polynomials* are defined recursively by the following formulae:

$$T_0(x) = 1\,, \quad T_1(x) = x\,, \quad T_k(x) = 2\,x\,T_{k-1}(x) \;-\; T_{k-2}(x)\,.$$

Compute the values of $T_2(x), \ldots, T_5(x)$ for $x = 1/3$, $x = 0.33$, and for a symbolical value x.

Series Expansions

Expressions such as `1/(1 - x)` admit series expansions for symbolic parameters. This particularly simple example is the sum of the geometric series:

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots$$

The function `taylor` computes the leading terms of such series:

```
t := taylor(1/(1 - x), x = 0, 9)
```

$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + O\left(x^9\right)$$

This is the Taylor series expansion of the expression around the point $x = 0$, as requested by the second argument. MuPAD® has truncated the infinite series before the term x^9 and has collected the tail in the "big Oh" term $O(x^9)$ (also called the "Landau symbol"). The (optional) third argument of `taylor` controls the truncation. If it is not present, then MuPAD uses the value of the environment variable `ORDER` instead, whose default value is 6:

```
t := taylor(1/(1 - x), x = 0)
```

$$1 + x + x^2 + x^3 + x^4 + x^5 + O\left(x^6\right)$$

The resulting series looks like an ordinary sum with an additional `O(·)` term. Internally, however, it is represented by a special data structure of domain type `Series::Puiseux`

```
domtype(t)
```

Series::Puiseux

The big-Oh term itself is a data structure on its own, of domain type `O` and with special rules of manipulation:

```
2*O(x^2) + O(x^3), x^2*O(x^10), O(x^5)*O(x^20),
diff(O(x^3), x)
```

$$O\left(x^2\right),\ O\left(x^{12}\right),\ O\left(x^{25}\right),\ O\left(x^2\right)$$

The ordering of the terms in a Taylor series is fixed: powers with smaller exponents precede those with higher exponents. This is in contrast to the ordering in ordinary sums, where high exponents precede small exponents:

```
S := expr(t)
```

$$x^5 + x^4 + x^3 + x^2 + x + 1$$

Here we have used the system function `expr` to convert the series to an expression of domain type `DOM_EXPR`. As you can see in the output, the `O(·)` term has been cut off.

The `op` command acts on series in a non-obvious way and should not be used:

```
op(t)
```

$$0,\ 1,\ 0,\ 6,\ [1,1,1,1,1,1],\ x = 0,\ \text{Undirected}$$

Therefore, the function `coeff` is provided to extract the coefficients. This is more intuitive than `op`. The call `coeff(t, i)` returns the coefficient of x^i:

```
t := taylor(cos(x^2), x, 20)
```

$$1 - \frac{x^4}{2} + \frac{x^8}{24} - \frac{x^{12}}{720} + \frac{x^{16}}{40320} + O\left(x^{20}\right)$$

```
coeff(t, 0), coeff(t, 1), coeff(t, 12), coeff(t, 25)
```

$$1,\ 0,\ -\frac{1}{720},\ \text{FAIL}$$

In the previous example, we have supplied `x` as second argument to specify the point of expansion. This is equivalent to `x=0`.

The usual arithmetic operations also work for series:

```
a := taylor(cos(x), x, 3): b := taylor(sin(x), x, 4):
```

```
a, b
```

$$1 - \frac{x^2}{2} + O\left(x^4\right),\ x - \frac{x^3}{6} + O\left(x^5\right)$$

```
a + b, 2*a*b, a^2
```

$$1 + x - \frac{x^2}{2} - \frac{x^3}{6} + O\left(x^4\right),\ 2\,x - \frac{4\,x^3}{3} + O\left(x^5\right),\ 1 - x^2 + O\left(x^4\right)$$

Both the composition operator `@` and the iteration operator `@@` apply to series as well:

```
a := taylor(sin(x), x, 20):
```

```
b := taylor(arcsin(x), x, 20): a@b
```

$$x + O\left(x^{21}\right)$$

If you try to compute the Taylor series of a function that does not have one, then `taylor` aborts with an error. The function `series` can compute more general expansions (Laurent series, Puiseux series):

```
taylor(cos(x)/x, x = 0, 10)
```

```
  Error: 1/x*cos(x) does not have a Taylor series \
  expansion, try 'series' [taylor]
```

```
series(cos(x)/x, x = 0, 10)
```

$$\frac{1}{x} - \frac{x}{2} + \frac{x^3}{24} - \frac{x^5}{720} + \frac{x^7}{40320} + O\left(x^9\right)$$

You can generate series expansions in terms of negative powers by expanding around the point `infinity`:

```
series((x^2 + 1)/(x + 1), x = infinity)
```

$$x - 1 + \frac{2}{x} - \frac{2}{x^2} + \frac{2}{x^3} - \frac{2}{x^4} + O\left(\frac{1}{x^5}\right)$$

This is an example for an “asymptotic” expansion, which approximates the behavior of a function for large values of the argument. In simple cases, `series` returns an expansion in terms of negative powers of x, but other functions may turn up as well:

```
series((exp(x) - exp(-x))/(exp(x) + exp(-x)),
       x = infinity)
```

$$1 - \frac{2}{\mathrm{e}^{2\,x}} + \frac{2}{\mathrm{e}^{4\,x}} - \frac{2}{\mathrm{e}^{6\,x}} + \frac{2}{\mathrm{e}^{8\,x}} - \frac{2}{\mathrm{e}^{10\,x}} + O\left(\mathrm{e}^{-12\,x}\right)$$

Exercise 4.33: The order p of a root x of a function f is the maximal number of derivatives that vanish at the point x:

$$f(x) = f'(x) = \cdots = f^{(p-1)}(x) = 0\,, \quad f^{(p)}(x) \neq 0\,.$$

What is the order of the root $x = 0$ of $f(x) = \tan(\sin(x)) - \sin(\tan(x))$?

Exercise 4.34: Besides the arithmetical operators, some other system functions such as `diff` or `int` work directly for series. Compare the result of `taylor(diff(1/(1-x),x),x)` and the derivative of `taylor(1/(1-x),x)`. Mathematically, both series are identical. Can you explain the difference in MuPAD?

Exercise 4.35: The function $f(x) = \sqrt{x+1} - \sqrt{x-1}$ tends to zero for large x, i.e., $\lim_{x\to\infty} f(x) = 0$. Show that the approximation $f(x) \approx 1/\sqrt{x}$ is valid for large values of x. Find better asymptotic approximations of f.

Exercise 4.36: Compute the first three terms in the series expansion of the function `f:=sin(x+x^3)` around `x=0`. Read the help page for the MuPAD function `revert`. Use this function to compute the leading terms of the series expansion of the inverse function f^{-1} (which is well-defined in a certain neighborhood of $x = 0$).

Algebraic Structures: Fields, Rings, etc.

The MuPAD® kernel provides domain types for the basic data structures such as numbers, sets, tables, etc. In addition, you can define your own data structures in the MuPAD language and work with them symbolically. We do not discuss the construction of such new "domains" in this elementary introduction, but demonstrate some special "library" domains provided by the system.

Besides the kernel domains, the `Dom` library contains a variety of predefined domains that were implemented by the MuPAD developers. For an overview, see the reference documentation. In this section, we present some particularly useful domains representing complex mathematical objects such as fields, rings, etc. Section 4.15 discusses a data type for matrices, which is well-suited for problems in linear algebra.

The main part of a domain is its *constructor*, which generates objects of the domain. Each such object "knows" its domain, which has *methods* attached to it. They represent the mathematical operations for these objects.

Here is a list of some of the well-known mathematical structures implemented in the `Dom` library:

- the ring of integers $\mathbb{Z}$: `Dom::Integer`,
- the field of rational numbers $\mathbb{Q}$: `Dom::Rational`,
- the field of real numbers $\mathbb{R}$: `Dom::Real` or `Dom::Float`,[7]
- the field of complex numbers $\mathbb{C}$: `Dom::Complex`,
- the ring of integers modulo n: `Dom::IntegerMod(n)`.

We consider the residue class ring of integers modulo n. Its elements are the integers $0, 1, \ldots, n-1$, and addition and multiplication are defined "modulo n." This works by adding or multiplying in $\mathbb{Z}$, dividing the result by n, and taking the remainder of this division in $\{0, 1, \ldots, n-1\}$:

[7] `Dom::Real` is for symbolic representations of real numbers, while `Dom::Float` represents them as floating-point numbers.

```
3*5 mod 7
```

$$1$$

In this example, we have used the data types of the MuPAD kernel: the operator `*` multiplies the integers 3 and 5 in the usual way to get 15, and the operator `mod` computes the decomposition $15 = 2 \cdot 7 + 1$ and returns 1 as remainder modulo 7.

Alternatively, you may tell MuPAD that you want to compute in $\mathbb{Z}_7 = \mathbb{Z}/7\,\mathbb{Z}$ by using the input syntax `Dom::IntegerMod(7)`. The latter object acts as a constructor for elements of the residue class ring[8] modulo 7:

```
constructor := Dom::IntegerMod(7):
```

```
x := constructor(3); y := constructor(5)
```

$$3 \bmod 7$$

$$5 \bmod 7$$

As you can see from the screen output, the identifiers `x` and `y` do not have the integers 3 and 5, respectively, as values. Instead, the numbers are elements of the residue class ring of integers modulo 7:

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

$$\mathbb{Z}_7,\ \mathbb{Z}_7$$

Now, you can use the usual arithmetic operations, and MuPAD automatically uses the computation rules of the residue class ring:

```
x*y, x^123*y^17 - x + y
```

$$1 \bmod 7,\ 6 \bmod 7$$

The ring `Dom::IntegerMod(7)` even has a field structure, so that you can divide by all ring elements except `0 mod 7`:

[8] If you want to execute only a small number of modulo operations, it is often preferable to use the operator `mod`, which is implemented in the MuPAD kernel and quite fast. This approach may require some additional understanding how the system functions work. For example, the computation of `17^29999 mod 7` takes quite a long time, since MuPAD first computes the very big number 17^{29999} and then reduces the result modulo 7. In this case, the computation `x^29999`, where `x:=Dom::IntegerMod(7)(17)`, is much faster since the internal modular arithmetic avoids such big numbers. Alternatively, the call `powermod(17, 29999, 7)` uses the system function `powermod` to compute the result quickly without employing `Dom::IntegerMod(7)`.

```
x/y
```

$$2 \bmod 7$$

A more abstract example is the field extension

$$K = \mathbb{Q}[\sqrt{2}] = \{p + q\sqrt{2}\,;\ p, q \in \mathbb{Q}\}\,.$$

You may define this field via

```
K := Dom::AlgebraicExtension(Dom::Rational,
                             Sqrt2^2 = 2, Sqrt2):
```

Here the identifier `Sqrt2` ($\widehat{=}\sqrt{2}$), defined by its algebraic property `Sqrt2^2=2`, is used to extend the rational numbers `Dom::Rational`. Now, you can compute in this field:

```
x := K(1/2 + 2*Sqrt2): y := K(1 + 2/3*Sqrt2):
```

```
x^2*y + y^4
```

$$\frac{677\,\mathrm{Sqrt2}}{54} + \frac{5845}{324}$$

The domain `Dom::ExpressionField(normalizer, zerotest)` represents the "field" of (symbolic) MuPAD expressions. The constructor is parametrized by two functions `normalizer` and `zerotest`, which may be chosen by the user.

The function `zerotest` is called internally by all algorithms that want to decide whether a domain object is mathematically 0. Typically, you will use the system function `iszero`, which recognizes not only the integer `0` as zero, but also other objects such as the floating-point number `0.0` or the polynomial `poly(0, [x])` (Section 4.16).

The task of the function `normalizer` is to generate a normal form for MuPAD objects of type `Dom::ExpressionField(·, ·)`. Operations on such objects will use this function to simplify the result before returning it. For example, if you supply the identity function `id` for the `normalizer` argument, then operations on objects of this domain work like for the usual MuPAD expressions without additional normalization:

```
constructor := Dom::ExpressionField(id, iszero):
```

```
x := constructor(a/(a + b)^2):
y := constructor(b/(a + b)^2):
```

```
x + y
```

$$\frac{a}{(a+b)^2} + \frac{b}{(a+b)^2}$$

If you supply the system function `normal` instead, the result is simplified automatically (Section 9.1):

```
constructor := Dom::ExpressionField(normal, iszero):
```

```
x := constructor(a/(a + b)^2):
y := constructor(b/(a + b)^2):
```

```
x + y
```

$$\frac{1}{a+b}$$

We note that the purpose of such MuPAD domains is not necessarily the direct generation of data structures or the computation with the corresponding objects. Indeed, some constructors simply return objects of the underlying kernel domains, if such domains exist:

```
domtype(Dom::Integer(2)),
domtype(Dom::Rational(2/3)),
domtype(Dom::Float(PI)),
domtype(Dom::ExpressionField(id, iszero)(a + b))
```

$$\mathrm{DOM\_INT},\ \mathrm{DOM\_RAT},\ \mathrm{DOM\_FLOAT},\ \mathrm{DOM\_EXPR}$$

In these cases, there is no immediate benefit in using such a constructor; you may as well compute directly with the corresponding kernel objects. The main application of such special data structures is the construction of more complex mathematical structures. A simple example is the generation of matrices (Section 4.15) or polynomials (Section 4.16) with entries in a particular ring, such that matrix or polynomial arithmetic, respectively, is performed according to the computation rules of the coefficient ring.

Vectors and Matrices

In Section 4.14, we have given examples of special data types ("domains") for defining algebraic structures such as rings, fields, etc. in MuPAD®. In this section, we discuss two further domains for generation and convenient computation with vectors and matrices: `Dom::Matrix` and `Dom::SquareMatrix`. In principle, you may use arrays for working with vectors or matrices (Section 4.9). However, then you have to define you own routines for addition, multiplication, inversion, or determinant computation, using the MuPAD programming language (Chapter 17). For the special matrix type that we present in what follows, such routines exist and are "attached" to the matrices as methods. Moreover, you may use the functions of the `linalg` library (linear algebra, Section 4.15), which can handle matrices of this type.

Definition of Matrices and Vectors

Vectors in MuPAD® are regarded as special matrices of dimension $1 \times n$ or $n \times 1$, respectively. The command `matrix` may be used for creating matrices and vectors of arbitrary dimension:

```
matrix([[  1,      2,      3,     4   ],
        [  a,      b,      c,     d   ],
        [sin(x), cos(x), exp(x), ln(x)]]),
matrix([x1, x2, x3, x4])
```

$$\left(\begin{array}{cccc} 1 & 2 & 3 & 4 \\ a & b & c & d \\ \sin(x) & \cos(x) & \mathrm{e}^{x} & \ln(x) \end{array}\right), \left(\begin{array}{c} \mathrm{x1} \\ \mathrm{x2} \\ \mathrm{x3} \\ \mathrm{x4} \end{array}\right)$$

Arbitrary arithmetical expressions may be used as elements. Although matrices created by `matrix` are suitable for most applications and will be the most widely used matrix objects, MuPAD's concept for matrices is more general. Typically, a specific coefficient ring for the elements is attached to a MuPAD matrix. In fact, the function `matrix` is just a constructor with a short name for special matrices of domain type `Dom::Matrix(R)`, where `R = Dom::ExpressionField()` represents arbitrary MuPAD expressions.

We explain the general concept. MuPAD provides the data type `Dom::Matrix` for matrices of arbitrary dimension $m \times n$. The type `Dom::SquareMatrix` represents square matrices of dimension $n \times n$. They belong to the library `Dom`, which also comprises data types for mathematical structures such as rings and fields (Section 4.14). Matrices may have entries from a set that must be equipped with a ring structure in the mathematical sense. For example, you may use the predefined rings and fields such as `Dom::Integer`, `Dom::IntegerMod(n)`, etc. from the `Dom` library.

The call `Dom::Matrix(R)` creates the constructor for matrices of arbitrary dimension $m \times n$ with coefficients in the ring `R`. When you construct such a matrix, you are required to ensure that its entries belong to (or may be converted to) this ring. You should keep this in mind when trying to generate matrices with entries outside the coefficient ring in a computation (for example, the inverse of an integer matrix in general has non-integral rational entries).

The following example yields the constructor for matrices with rational number entries:

```
constructor := Dom::Matrix(Dom::Rational)
```

$$\mathrm{Dom}::\mathrm{Matrix}(\mathbb{Q})$$

Now, you may generate matrices of arbitrary dimensions. In the following example, we generate a 2×3 matrix with all entries initialized to 0:

```
A := constructor(2, 3)
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

When generating a matrix, you may supply a function f that takes two arguments. Then the entry in row i and column j is initialized with $f(i, j)$:

```
f := (i, j) -> (i*j):  A := constructor(2, 3, f)
```

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

Alternatively, you can initialize a matrix by specifying a (nested) list. Each list element is itself a list and corresponds to one row of the matrix. The following command generates the same matrix as in the previous example:

```
constructor(2, 3, [[1, 2, 3], [2, 4, 6]]):
```

The parameters for the dimension are optional here, since they are also given by the structure of the list. Thus,

```
constructor([[1, 2, 3], [2, 4, 6]]):
```

also returns the same matrix. An array of domain type `DOM_ARRAY` (Section 4.9) is also valid for initializing a matrix:

```
Array := array(1..2, 1..3, [[1, 2, 3], [2, 4, 6]]):
Mat := constructor(Array):
```

You may define column and row vectors as $m \times 1$ and $1 \times n$ matrices, respectively. Plain lists can be used for defining column or row vectors:

```
column := constructor(3, 1, [1, 2, 3])
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

```
row := constructor(1, 3, [1, 2, 3])
```

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

If no dimensions are specified, a plain list generates a column vector:

```
column := constructor([1, 2, 3])
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

The entries of a matrix or a vector may be accessed in the forms `matrix[i, j]`, `row[i]`, or `column[j]`. Since vectors are special matrices, you may access the components of a vector also in the form `row[1, i]` or `column[j, 1]`, respectively:

```
A[2, 3], row[3], row[1, 3],
column[2], column[2, 1]
```

$$6,\ 3,\ 3,\ 2,\ 2$$

Submatrices are generated as follows:

```
Mat[1..2, 1..2], row[1..1, 1..2],
column[1..2, 1..1]
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix},\ \begin{pmatrix} 1 & 2 \end{pmatrix},\ \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

You may change a matrix entry by an indexed assignment:

```
Mat[2, 3] := 23: row[2] := 5: column[2, 1] := 5:
```

```
Mat, row, column
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 23 \end{pmatrix},\ \begin{pmatrix} 1 & 5 & 3 \end{pmatrix},\ \begin{pmatrix} 1 \\ 5 \\ 3 \end{pmatrix}$$

You can use loops (Chapter 15) to change all components of a matrix:

```
m := 2: n := 3: Mat := constructor(m, n):
```

```
for i from 1 to m do
  for j from 1 to n do
    Mat[i, j] := i*j
  end_for
end_for:
```

You can generate diagonal matrices by supplying the option `Diagonal`. In this case, the third argument to the constructor may either be a list of the diagonal elements or a function f such that the i-th diagonal element is $f(i)$:

```
constructor(2, 2, [11, 12], Diagonal)
```

$$\begin{pmatrix} 11 & 0 \\ 0 & 12 \end{pmatrix}$$

In the next example, we generate an identity matrix by supplying 1 as a function[9] defining the diagonal elements:

```
constructor(2, 2, 1, Diagonal)
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Alternatively, identity matrices can be generated by the "identity" method of the constructor:

```
constructor::identity(2)
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The constructor considered so far returns matrices with rational (i.e., real) number entries. Thus, the following attempt to generate a matrix with complex coefficients does not work:

```
constructor([[1, 2, 3], [2, 4, 1 + I]])
  Error: unable to define matrix over Dom::Rational \
   [(Dom::Matrix(Dom::Rational))::new]
```

You have to choose a suitable coefficient ring to generate a matrix with the above entries. In the following example, we define a new constructor for matrices with complex number entries:

```
constructor := Dom::Matrix(Dom::Complex):
```

```
constructor([[1, 2, 3], [2, 4, 1 + I]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 1+\mathrm{i} \end{pmatrix}$$

You may generate matrices whose entries are arbitrary MuPAD expressions by means of the field Dom::ExpressionField(id, iszero) (see Section 4.14). This is the standard coefficient ring for matrices. You may always use this ring when the coefficients and their properties are irrelevant.

[9] Cf. page 4-54.

`Dom::Matrix()` with no argument is a constructor for such matrices. As a shorthand notation, this matrix ring is attached to the identifier `matrix`:

```
matrix
```

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

```
matrix([[1, x + y, 1/x^2], [sin(x), 0, cos(x)],
        [x*PI, 1 + I, -x*PI]])
```

$$\begin{pmatrix} 1 & x+y & \frac{1}{x^2} \\ \sin(x) & 0 & \cos(x) \\ \pi\,x & 1+\mathrm{i} & -\pi\,x \end{pmatrix}$$

If you use `Dom::ExpressionField(normal, iszero)` as the coefficient ring, then all matrix entries are simplified via the function `normal`, as described in Section 4.14. Arithmetical operations with such matrices are comparatively slow, since a call to `normal` may be quite time consuming. However, the results are in general simpler than the (equivalent) results of computations with the standard coefficient ring `Dom::ExpressionField(id, iszero)` used by `Dom::Matrix()`.

The constructor `Dom::SquareMatrix(n,R)` corresponds to the ring of n-dimensional square matrices with coefficient ring `R`. If the argument `R` is missing, then MuPAD automatically uses the coefficient ring of all MuPAD expressions. The following statement yields the constructor for 2×2 matrices, whose entries may be arbitrary MuPAD expressions:

```
constructor := Dom::SquareMatrix(2)
```

$$\text{Dom::SquareMatrix}(2)$$

```
constructor([[0, y], [x^2, 1]])
```

$$\begin{pmatrix} 0 & y \\ x^2 & 1 \end{pmatrix}$$

Computing with Matrices

You can use the standard arithmetical operators for doing basic arithmetic with matrices:

```
A := matrix([[1, 2], [3, 4]]):
```

```
B := matrix([[a, b], [c, d]]):
```

```
A + B, A*B;
A*B - B*A, A^2 + B
```

$$\begin{pmatrix} a+1 & b+2 \\ c+3 & d+4 \end{pmatrix}, \begin{pmatrix} a+2c & b+2d \\ 3a+4c & 3b+4d \end{pmatrix}$$

$$\begin{pmatrix} 2c-3b & 2d-3b-2a \\ 3a+3c-3d & 3b-2c \end{pmatrix}, \begin{pmatrix} a+7 & b+10 \\ c+15 & d+22 \end{pmatrix}$$

Multiplication of a matrix and a number works componentwise (scalar multiplication):

```
2*B
```

$$\begin{pmatrix} 2a & 2b \\ 2c & 2d \end{pmatrix}$$

The inverse of a matrix is represented by `1/A` or `A^(-1)`:

```
C := A^(-1)
```

$$\begin{pmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{pmatrix}$$

A simple test shows that the computed inverse is correct:

```
A*C, C*A
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

An inversion returns `FAIL` when MuPAD is unable to compute the result. The following matrix is not invertible:

```
C := matrix([[1, 1], [1, 1]]):
C^(-1)
```

$$\text{FAIL}$$

The concatenation operator `.`, which combines lists (Section 4.6) or strings (Section 4.11), is "overloaded" for matrices. You can use it to combine matrices with the same number of rows:

```
A, B, A.B
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \begin{pmatrix} 1 & 2 & a & b \\ 3 & 4 & c & d \end{pmatrix}$$

Besides the arithmetic operators, other system functions are applicable to matrices. Here is a list of examples:

- `conjugate(A)` replaces all components by their complex conjugates,
- `diff(A, x)` differentiates componentwise with respect to `x`,
- `exp(A)` computes $e^A = \sum_{i=0}^{\infty} \frac{1}{i!} A^i$,
- `expand(A)` applies `expand` to all components of `A`,
- `expr(A)` converts `A` to an array of domain type `DOM_ARRAY`,
- `float(A)` applies `float` to all components of `A`,
- `has(A, expression)` checks whether an expression is contained in at least one entry of `A`,
- `int(A, x)` integrates componentwise with respect to `x`,
- `iszero(A)` checks whether all components of `A` vanish,
- `map(A, function)` applies the function to all components,

- `norm(A)` (identical with `norm(A, Infinity)`) computes the infinity norm,[10]
- `A | equation` applies $\cdot$ `| equation` to all entries of `A`,
- `subs(A, equation)` applies `subs(`$\cdot$`, equation)` to all entries of `A`,
- `C :=zip(A, B, f)` returns the matrix defined by $C_{ij} = f(A_{ij}, B_{ij})$.

The linear algebra library `linalg` and the numerics library `numeric` (Section 4.15) comprise many other functions for handling matrices.

Exercise 4.37: Generate the 15×15 Hilbert matrix $H = (H_{ij})$ with $H_{ij} = 1/(i + j - 1)$. Generate the vector $\vec{b} = H\,\vec{e}$, where $\vec{e} = (1, \ldots, 1)^t$. Compute the exact solution vector $\vec{x}$ of the system of equations $H\,\vec{x} = \vec{b}$ (of course, this should yield $\vec{x} = \vec{e}$). Convert all entries of H to floating-point values and solve the system of equations again. Compare the result to the exact solution. You will note a dramatic difference, which is caused by numerical rounding errors. Larger Hilbert matrices cannot be inverted with the standard precision of common numerical software!

Special Methods for Matrices

A constructor that has been generated by means of either `Dom::Matrix(`$\cdot$`)` or `Dom::SquareMatrix(`$\cdot$`)` contains many special functions for the corresponding data type. If `M:=Dom::Matrix(ring)` is a constructor and `A:=M(`$\cdot$`)` is a matrix generated with this constructor, as described in Section 4.15, then, among may others, the following methods are available:

- `M::col(A, i)` returns the i-th column of `A`,
- `M::delCol(A, i)` removes the i-th column from `A`,
- `M::delRow(A, i)` removes the i-th row from `A`,
- `M::matdim(A)` returns the dimension `[m, n]` of the $m \times n$ matrix `A`,

[10] `norm(A, 1)` returns the one-norm, `norm(A, Frobenius)` yields the Frobenius norm $\left(\sum_{i,j} |A_{ij}|^2\right)^{1/2}$.

- `M::random()` returns a matrix with random entries,
- `M::row(A, i)` returns the i-th row of `A`,
- `M::swapCol(A, i, j)` exchanges columns i and j,
- `M::swapRow(A, i, j)` exchanges rows i and j,
- `M::tr(A)` returns the trace $\sum_i A_{ii}$ of A,
- `M::transpose(A)` returns the transpose (A_{ji}) of $A = (A_{ij})$.

```
M := Dom::Matrix(): A := M([[x, 1], [2, y]])
```

$$\begin{pmatrix} x & 1 \\ 2 & y \end{pmatrix}$$

```
M::col(A, 1), M::delCol(A, 1), M::matdim(A)
```

$$\begin{pmatrix} x \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ y \end{pmatrix}, [2, 2]$$

```
M::swapCol(A, 1, 2), M::tr(A), M::transpose(A)
```

$$\begin{pmatrix} 1 & x \\ y & 2 \end{pmatrix}, x + y, \begin{pmatrix} x & 2 \\ 1 & y \end{pmatrix}$$

Since the domain is attached to the object `A` as `A::dom`, one can also call the domain methods via `A::dom::method`. For example:

```
A::dom::tr(A)
```

$$x + y$$

The documentation of `Dom::Matrix` contains a survey of these methods.

The Libraries `linalg` and `numeric`

Besides system functions operating on matrices, the library[11] `linalg` contains a variety of other linear algebra functions:

```
info(linalg)
  Library 'linalg': the linear algebra package

  -- Interface:
  linalg::addCol,           linalg::addRow,
  linalg::adjoint,          linalg::angle,
  linalg::basis,            linalg::charmat,
  linalg::charpoly,         linalg::col,
  linalg::companion,        linalg::concatMatrix,
  linalg::crossProduct,     linalg::curl,
  linalg::delCol,           linalg::delRow,
  linalg::det,              linalg::divergence,
  linalg::eigenvalues,      linalg::eigenvectors,
  linalg::expr2Matrix,      linalg::factorCholesky,
  linalg::factorLU,         linalg::factorQR,
  linalg::frobeniusForm,    linalg::gaussElim,
  linalg::gaussJordan,      linalg::grad,
  linalg::hermiteForm,      linalg::hessenberg,
  linalg::hessian,          linalg::hilbert,
  linalg::intBasis,         linalg::inverseLU,
  linalg::invhilbert,       linalg::invpascal,
  linalg::invvandermonde,   linalg::isHermitean,
  linalg::isPosDef,         linalg::isUnitary,
  linalg::jacobian,         linalg::jordanForm,
  linalg::kroneckerProduct, linalg::laplacian,
  linalg::matdim,           linalg::matlinsolve,
  linalg::matlinsolveLU,    linalg::minpoly,
  linalg::multCol,          linalg::multRow,
  linalg::ncols,            linalg::nonZeros,
  linalg::normalize,        linalg::nrows,
  linalg::nullspace,        linalg::ogCoordTab,
  linalg::orthog,           linalg::pascal,
  linalg::permanent,        linalg::potential,
  linalg::pseudoInverse,    linalg::randomMatrix,
  linalg::rank,             linalg::row,
```

[11] We refer to Chapter 3 for a description of general library organization, exporting, etc.

```
linalg::scalarProduct,    linalg::setCol,
linalg::setRow,           linalg::smithForm,
linalg::stackMatrix,      linalg::submatrix,
linalg::substitute,       linalg::sumBasis,
linalg::swapCol,          linalg::swapRow,
linalg::sylvester,        linalg::toeplitz,
linalg::toeplitzSolve,    linalg::tr,
linalg::transpose,        linalg::vandermonde,
linalg::vandermondeSolve, linalg::vecdim,
linalg::vectorOf,         linalg::vectorPotential,
linalg::wiedemann
```

Some of these functions, such as `linalg::col` or `linalg::delCol`, simply call the internal methods for matrices that we have described in Section 4.15, and hence do not add new functionality. However, `linalg` also contains many additional algorithms. The command `?linalg` yields a short description of all functions. You can find a detailed description of a function on the corresponding help page.

You may use the full name `library::function` to call a function:

```
A := matrix([[a, b], [c, d]]): linalg::det(A)
```

$$a\,d - b\,c$$

The characteristic polynomial $\det(x\,I_n - A)$ of this matrix is

```
linalg::charpoly(A, x)
```

$$x^2 + (-a - d)\,x + a\,d - b\,c$$

The eigenvalues are

```
linalg::eigenvalues(A)
```

$$\left\{\frac{a}{2} + \frac{d}{2} - \frac{\sqrt{a^2 - 2\,a\,d + d^2 + 4\,b\,c}}{2}, \frac{a}{2} + \frac{d}{2} + \frac{\sqrt{a^2 - 2\,a\,d + d^2 + 4\,b\,c}}{2}\right\}$$

The numerics library `numeric` (see `?numeric`) contains many functions for numerical computations with matrices:

`numeric::det`	: determinant
`numeric::expMatrix`	: exp(Matrix)
`numeric::factorCholesky`	: Cholesky factorization
`numeric::factorLU`	: LU factorization
`numeric::factorQR`	: QR factorization
`numeric::fMatrix`	: functional calculus
`numeric::inverse`	: inversion
`numeric::eigenvalues`	: eigenvalues
`numeric::eigenvectors`	: eigenvalues and –vectors
`numeric::singularvalues`	: singular values
`numeric::singularvectors`	: singular values and vectors

Partially, these routines work for matrices with symbolic entries of type `Dom::ExpressionField` and then are more efficient for large matrices than the `linalg` functions. However, the latter can handle arbitrary coefficient rings.

Exercise 4.38: Find the values of a, b, c for which the matrix $\begin{pmatrix} 1 & a & b \\ 1 & 1 & c \\ 1 & 1 & 1 \end{pmatrix}$ is not invertible.

Exercise 4.39: Consider the following matrices:

$$A = \begin{pmatrix} 1 & 3 & 0 \\ -1 & 2 & 7 \\ 0 & 8 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 7 & -1 \\ 2 & 3 \\ 0 & 1 \end{pmatrix}.$$

Let B^T be the transpose of B. Compute the inverse of $2\,A + B\,B^T$, both over the rational numbers and over the residue class ring modulo 7.

Exercise 4.40: Generate the 3×3 matrix

$$A_{ij} = \begin{cases} 0 & \text{for } i = j\,, \\ 1 & \text{for } i \neq j\,. \end{cases}$$

Compute its determinant, its characteristic polynomial, and its eigenvalues. For each eigenvalue, compute a basis of the corresponding eigenspace.

Sparse Matrices

The internal storage of matrices is optimized for sparse data, i.e., for matrices consisting largely of zeroes, since such matrices appear often in practice. Here are some remarks concerning efficiency when your matrices are large and sparse:

- Use the standard coefficient ring `Dom::ExpressionField()` of arbitrary MuPAD® expressions whenever possible. As discussed in the previous sections, the constructor `Dom::Matrix()` creates matrices of this type. For convenience, this constructor is also available as the function `matrix`.
- Indexed reading and writing to large matrices is somewhat expensive. If possible, one should avoid creating large empty matrices of the desired dimension by `matrix(m, n)` and filling in the non-zero entries by indexed assignments. One should set the entries directly when creating the matrix.

Lists of equations can be used to specify the entries when creating a matrix. The following matrix A of dimension 1000×1000 consists of a diagonal band and two additional entries in the upper right and the lower left corner. We display some entries of its 10-th power:

```
n := 1000:
A := matrix(n, n, [(i, i) = i $ i = 1..n,
                   (n, 1) = 1, (1, n) = 1]):
B := A^10:
B[1, 1], B[1, n]
```

$$1002010022050086122130089,\ 100100901504006610111910505 5$$

In the following example, we generate a 100×100 tri-diagonal Toeplitz matrix `A` and solve the equation $A\,x = b$, where b is the column vector $(1, \ldots, 1)$:

```
A := matrix(100, 100, [-1, 2, -1], Banded):
b := matrix(100, 1, [1 $ 100]):
x := (1/A)*b
```

$$\begin{pmatrix} 50 \\ 99 \\ \ldots \\ 50 \end{pmatrix}$$

Here we have applied the inverse `1/A` of `A` to the right-hand side of the equation. Note, however, that inverting a sparse matrix is not a good idea because the inverse of a sparse matrix is, in general, not sparse. It is much more efficient to use sparse matrix factorization to compute the solution of a sparse system of linear equations. We use the linear solver `numeric::matlinsolve` with the option `Symbolic` to compute the solution of a sparse system representing 1000 equations for 1000 unknowns. The `numeric` routine employs sparsity in an optimal way:

```
A := matrix(1000, 1000, [-1, 2, -1], Banded):
b := matrix(1000, 1, [1 $ 1000]):
[x, kernel] := numeric::matlinsolve(A, b, Symbolic):
```

We display only a few components of the solution vector:

```
x[1], x[2], x[3], x[4], x[5], x[999], x[1000]
```

$$500,\ 999,\ 1497,\ 1994,\ 2490,\ 999,\ 500$$

An Application

We want to compute the symbolic solution $a(t), b(t)$ of the system of second order differential equations

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}\,a(t) = 2\,c\,\frac{\mathrm{d}}{\mathrm{d}t}\,b(t)\,, \quad \frac{\mathrm{d}^2}{\mathrm{d}t^2}\,b(t) = -2\,c\,\frac{\mathrm{d}}{\mathrm{d}t}\,a(t) + 3\,c^2\,b(t)$$

with an arbitrary constant c. Writing $a'(t) = \frac{\mathrm{d}}{\mathrm{d}t}a(t)$, $b'(t) = \frac{\mathrm{d}}{\mathrm{d}t}b(t)$, these equations may be equivalently written as a system of first order differential equations in the

variables $x(t) = (a(t), a'(t), b(t), b'(t))$:

$$\frac{\mathrm{d}}{\mathrm{d}t}\, x(t) = A\, x(t), \quad A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2\,c \\ 0 & 0 & 0 & 1 \\ 0 & -2\,c & 3\,c^2 & 0 \end{pmatrix}.$$

The solution of this system is given by applying the exponential matrix e^{tA} to the initial condition $x(0)$:

$$x(t) = e^{tA} x(0)\,.$$

```
delete c, t:
A := matrix([[0,  1,    0,    0 ],
             [0,  0,    0,   2*c],
             [0,  0,    0,    1 ],
             [0, -2*c, 3*c^2, 0 ]]):
```

We use the function `exp` to compute $B = e^{tA}$:

```
B := exp(t*A)
```

$$\begin{pmatrix} 1 & -3\,t + \frac{2\,\mathrm{e}^{-c\,t\,\mathrm{i}}}{c} - \frac{2\,\mathrm{e}^{c\,t\,\mathrm{i}}}{c} & 3\,\mathrm{e}^{c\,t\,\mathrm{i}} + 6\,c\,t - 3\,\mathrm{e}^{-c\,t\,\mathrm{i}} & \frac{2}{c} - \frac{\mathrm{e}^{c\,t\,\mathrm{i}}}{c} - \frac{\mathrm{e}^{-c\,t\,\mathrm{i}}}{c} \\ 0 & 2\,\mathrm{e}^{-c\,t\,\mathrm{i}} + 2\,\mathrm{e}^{c\,t\,\mathrm{i}} - 3 & 6\,c - 3\,c\,\mathrm{e}^{-c\,t\,\mathrm{i}} - 3\,c\,\mathrm{e}^{c\,t\,\mathrm{i}} & \mathrm{e}^{-c\,t\,\mathrm{i}}\,\mathrm{i} - \mathrm{e}^{c\,t\,\mathrm{i}}\,\mathrm{i} \\ 0 & \frac{\mathrm{e}^{-c\,t\,\mathrm{i}}}{c} + \frac{\mathrm{e}^{c\,t\,\mathrm{i}}}{c} - \frac{2}{c} & 4 - \frac{3\,\mathrm{e}^{c\,t\,\mathrm{i}}}{2} - \frac{3\,\mathrm{e}^{-c\,t\,\mathrm{i}}}{2} & \frac{\mathrm{e}^{-c\,t\,\mathrm{i}}\,\mathrm{i}}{2\,c} - \frac{\mathrm{e}^{c\,t\,\mathrm{i}}\,\mathrm{i}}{2\,c} \\ 0 & -\mathrm{e}^{-c\,t\,\mathrm{i}}\,\mathrm{i} + \mathrm{e}^{c\,t\,\mathrm{i}}\,\mathrm{i} & \frac{3\,c\,\mathrm{e}^{-c\,t\,\mathrm{i}}\,\mathrm{i}}{2} - \frac{3\,c\,\mathrm{e}^{c\,t\,\mathrm{i}}\,\mathrm{i}}{2} & \frac{\mathrm{e}^{-c\,t\,\mathrm{i}}}{2} + \frac{\mathrm{e}^{c\,t\,\mathrm{i}}}{2} \end{pmatrix}$$

To simplify this matrix, we use the function `Simplify`, which for matrices simply maps itself to the entries:

```
B := Simplify(B)
```

$$\begin{pmatrix} 1 & \frac{4\,\sin(c\,t)}{c} - 3\,t & 6\,c\,t - 6\,\sin(c\,t) & -\frac{2\,(\cos(c\,t)-1)}{c} \\ 0 & 4\,\cos(c\,t) - 3 & -6\,c\,(\cos(c\,t) - 1) & 2\,\sin(c\,t) \\ 0 & \frac{2\,(\cos(c\,t)-1)}{c} & 4 - 3\,\cos(c\,t) & \frac{\sin(c\,t)}{c} \\ 0 & -2\,\sin(c\,t) & 3\,c\,\sin(c\,t) & \cos(c\,t) \end{pmatrix}$$

We assign a symbolic initial condition to the vector x_0. The following call creates a 4×1 matrix to be interpreted as a column vector:

```
x_0 := matrix([a_0, a_0', b_0, b_0'])
```

$$\begin{pmatrix} a_0 \\ a_0' \\ b_0 \\ b_0' \end{pmatrix}$$

Thus, the desired symbolic solution of the system of differential equations is

```
x := t --> B*x_0:
```

The solution functions $a(t)$ and $b(t)$ with the symbolic initial conditions $a(0) = a_0$, $a'(0) = a_0'$, $b(0) = b_0$, $b'(0) = b_0'$ are:

```
a := t --> expand(x(t)[1]): a(t)
```

$$a_0 - 6\,b_0\,\sin(c\,t) - 3\,t\,a_0' + \frac{2\,b_0'}{c} - \frac{2\,\cos(c\,t)\;b_0'}{c} + \frac{4\,a_0'\,\sin(c\,t)}{c} + 6\,b_0\,c\,t$$

```
b := t --> expand(x(t)[3]): b(t)
```

$$4\,b_0 - 3\,b_0\,\cos(c\,t) - \frac{2\,a_0'}{c} + \frac{2\,\cos(c\,t)\;a_0'}{c} + \frac{b_0'\,\sin(c\,t)}{c}$$

Finally, we check that the above expressions really solve the system of differential equations:

```
expand(diff(a(t),t,t) - 2*c*diff(b(t),t)),
expand(diff(b(t),t,t) + 2*c*diff(a(t),t) - 3*c^2*b(t))
```

$$0,\ 0$$

Polynomials

Computation with polynomials is an important task for a computer algebra system. Of course, you may realize a polynomial in MuPAD® as an expression in the sense of Section 4.4 and use the standard arithmetic:

```
polynomialExpression := 1 + x + x^2:
```

```
expand(polynomialExpression^2)
```

$$x^4 + 2\,x^3 + 3\,x^2 + 2\,x + 1$$

However, there exists a special data type `DOM_POLY` together with some kernel and library functions, which simplifies such computations and makes them more efficient.

Definition of Polynomials

The system function `poly` generates polynomials:

```
poly(1 + 2*x + 3*x^2)
```

$$\text{poly}\left(3\,x^2 + 2\,x + 1, [x]\right)$$

Here we have supplied the expression $1 + 2\,x + 3\,x^2$ (of domain type `DOM_EXPR`) to `poly`, which converts this expression to a new object of domain type `DOM_POLY`. The indeterminate `[x]` is a fixed part of this data type. This is relevant for distinguishing between indeterminates and (symbolic) coefficients or parameters. For example, if you want to regard the expression $a_0 + a_1\,x + a_2\,x^2$ as a polynomial in x with coefficients a_0, a_1, a_2, then the above form of the call to `poly` does not yield the desired result:

```
poly(a_0 + a_1*x + a_2*x^2)
```

$$\text{poly}\left(a_0 + a_1\,x + a_2\,x^2, [a_0, a_1, a_2, x]\right)$$

This does not represent a polynomial in x but a "multivariate" polynomial in four indeterminates x, a_0, a_1, a_2. You can specify the indeterminates of a polynomial in form of a list as argument to `poly`. The system then regards all other symbolic identifiers as symbolic coefficients:

```
poly(a_0 + a_1*x + a_2*x^2, [x])
```

$$\text{poly}\left(a_2\,x^2 + a_1\,x + a_0, [x]\right)$$

If you do not specify a list of indeterminates, then `poly` calls the function `indets` to determine all symbolic identifiers in the expression and interprets them as indeterminates of the polynomial:

```
indets(a_0 + a_1*x + a_2*x^2, PolyExpr)
```

$$\{a_0, a_1, a_2, x\}$$

The distinction between indeterminates and coefficients is relevant for the representation of the polynomial:

```
expression := 1 + x + x^2 + a*x + PI*x^2 - b
```

$$x - b + a\,x + \pi\,x^2 + x^2 + 1$$

```
poly(expression, [a, x])
```

$$\text{poly}\left(a\,x + (\pi + 1)\,x^2 + x + (1 - b), [a, x]\right)$$

```
poly(expression, [x])
```

$$\text{poly}\left((\pi + 1)\,x^2 + (a + 1)\,x + (1 - b), [x]\right)$$

You can see that MuPAD collects the coefficients of equal powers of the indeterminate. The terms are sorted according to falling exponents.

Instead of using an expression, you may also generate a polynomial by specifying a list of the non-zero coefficients together with the respective exponents. The command `poly(list, [x])` generates the polynomial $\sum_{i=0}^{k} a_i\,x^{n_i}$ from the list $[[a_0, n_0], [a_1, n_1], \ldots, [a_k, n_k]]$:

```
list := [[1, 0], [a, 3], [b, 5]]: poly(list, [x])
```

$$\text{poly}\left(b\,x^5 + a\,x^3 + 1, [x]\right)$$

If you want to construct a multivariate polynomial in this way, specify lists of exponents for all variables:

```
poly([[3, [2, 1]], [2, [3, 4]]], [x, y])
```

$$\text{poly}\left(2\,x^3\,y^4 + 3\,x^2\,y, [x, y]\right)$$

Conversely, the function `poly2list` converts a polynomial to a list of coefficients and exponents:

```
poly2list(poly(b*x^5 + a*x^3 + 1, [x]) )
```

$$[[b, 5], [a, 3], [1, 0]]$$

For more abstract computations, you may want to restrict the coefficients of a polynomial to a certain set (mathematically: a ring) which is represented by a special data structure. We have already seen typical examples of rings and their corresponding MuPAD domains in Section 4.14: the integers `Dom::Integer`, the rational numbers `Dom::Rational`, or the residue class ring `Dom::IntegerMod(n)` of integers modulo n. You may specify the coefficient ring as argument to `poly`:

```
poly(x + 1, [x], Dom::Integer)
```

$$\text{poly}\,(x + 1, [x], \mathbb{Z})$$

```
poly(2*x - 1/2, [x], Dom::Rational)
```

$$\text{poly}\left(2\,x - \left(\frac{1}{2}\right), [x], \mathbb{Q}\right)$$

```
poly(4*x + 11, [x], Dom::IntegerMod(3))
```

$$\text{poly}\,(x + 2, [x], \mathbb{Z}_3)$$

Note that in the last example, the system has automatically simplified the coefficients according to the rules for computing with integers modulo 3:[12]

```
4 mod 3, 11 mod 3
```

$$1,\ 2$$

In the following example, `poly` converts the coefficients to floating-point numbers, as specified by the third argument:

```
poly(PI*x - 1/2, [x], Dom::Float)
```

$$\text{poly}\,(3.141592654\,x - 0.5, [x], \text{Dom::Float})$$

[12] For polynomials, you may also use `IntMod(3)` instead of `Dom::IntegerMod(3)`, in the form `poly(4*x+11, [x], IntMod(3))`. Then the integers modulo 3 are represented by $-1, 0, 1$ and not, as for `Dom::IntegerMod(3)`, by $0, 1, 2$. Polynomial arithmetic is much faster when you use `IntMod(3)`.

If no coefficient ring is specified, then MuPAD by default uses the ring `Expr` which symbolizes arbitrary MuPAD expressions. In this case, you may use symbolic identifiers as coefficients:

```
polynomial := poly(a + x + b*y, [x, y]);
op(polynomial)
```

$$\text{poly}\left(x + b\,y + a, [x, y]\right)$$

$$a + x + b\,y,\ [x, y]\,,\ \text{Expr}$$

We summarize that a MuPAD polynomial comprises three parts:

1. a polynomial expression of the form $\sum\ a_{i_1 i_2 \ldots}\ x_1^{i_1} x_2^{i_2} \ldots$,
2. a list of indeterminates $[x_1, x_2, \ldots]$,
3. the coefficient ring.

These are the three operands of a MuPAD polynomial `p`, which can be accessed via `op(p, 1)`, `op(p, 2)`, and `op(p, 3)`, respectively. Thus, you may convert a polynomial to a mathematically equivalent expression[13] of domain type `DOM_EXPR` by

```
expression := op(polynomial, 1):
```

However, you should preferably use the system function `expr`, which can convert various domain types such as polynomials to expressions:

```
polynomial := poly(x^3 + 5*x + 3)
```

$$\text{poly}\left(x^3 + 5\,x + 3, [x]\right)$$

```
op(polynomial, 1) = expr(polynomial)
```

$$x^3 + 5\,x + 3 = x^3 + 5\,x + 3$$

[13] If the polynomial is defined over a ring other than `Dom::ExpressionField` or `Expr`, the result may not be equivalent.

Computing with Polynomials

The function `degree` determines the degree of a polynomial:

```
p := poly(1 + x + a*x^2*y, [x, y]):
```

```
degree(p, x), degree(p, y)
```

$$2,\ 1$$

If you do not specify the name of an indeterminate as second argument, then `degree` returns the "total degree":

```
degree(p), degree(poly(x^27 + x + 1))
```

$$3,\ 27$$

The function `coeff` extracts coefficients from a polynomial:

```
p := poly(1 + a*x + 7*x^7, [x]):
```

```
coeff(p, 1), coeff(p, 2), coeff(p, 8)
```

$$a,\ 0,\ 0$$

For multivariate polynomials, the coefficient of a power of one particular indeterminate is again a polynomial in the remaining indeterminates:

```
p := poly(1 + x + a*x^2*y, [x, y]):
```

```
coeff(p, y, 0), coeff(p, y, 1)
```

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

The standard operators `+`, `-`, `*` and `^` work for polynomial arithmetic as well:

```
p := poly(1 + a*x^2, [x]): q := poly(b + c*x, [x]):
```

```
p + q, p - q;
p*q, p^2
```

$$\text{poly}\left(a\,x^2+c\,x+(b+1),[x]\right),\ \text{poly}\left(a\,x^2+(-c)\ x+(1-b),[x]\right)$$

$$\text{poly}\left((a\,c)\ x^3+(a\,b)\ x^2+c\,x+b,[x]\right),\ \text{poly}\left(a^2\,x^4+(2\,a)\ x^2+1,[x]\right)$$

The function `divide` performs a “division with remainder”:

```
p := poly(x^3 + 1): q := poly(x^2 - 1): divide(p, q)
```

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

The result is a sequence with two operands: the quotient and the remainder of the division:

```
[quotient, remainder] := [divide(p, q)]:
```

We check:

```
p = quotient*q + remainder
```

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

The polynomial denoted by `remainder` is of lower degree than `q`, which makes the decomposition `p = quotient * q + remainder` unique. Dividing two polynomials by means of the usual division operator `/` is only allowed in the special case when the remainder that `divide` would return vanishes:

```
p := poly(x^2 - 1): q := poly(x - 1): p/q
```

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

```
p := poly(x^2 + 1): q := poly(x - 1): p/q
```

FAIL

Note that the arithmetic operators process only polynomials of exactly identical types:

```
poly(x + y, [x, y]) + poly(x^2, [x, y]),
poly(x) + poly(x, [x], Expr)
```

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

Both the list of indeterminates and the coefficient ring must coincide. Otherwise the system errors.

The polynomial arithmetic performs coefficient additions and multiplications according to the rules of the coefficient ring:

```
p := poly(4*x + 11, [x], Dom::IntegerMod(3)):
```

```
p; p + p; p*p
```

$$\text{poly}\left(x+2,[x],\mathbb{Z}_3\right)$$

$$\text{poly}\left(2\,x+1,[x],\mathbb{Z}_3\right)$$

$$\text{poly}\left(x^2+x+1,[x],\mathbb{Z}_3\right)$$

The standard operator * works for multiplying a polynomial by a scalar:

```
p := poly(x^2 + y):
```

```
scalar*p
```

$$\text{poly}\left(\text{scalar}\,x^2+\text{scalar}\,y,[x,y]\right)$$

Another important operation is the evaluation of a polynomial at a point (computing the image value). The function `evalp` achieves this:

```
p := poly(x^2 + 1, [x]):
evalp(p, x = 2), evalp(p, x = x + y)
```

$$5,\ (x+y)^2+1$$

This computation is also valid for multivariate polynomials and yields a polynomial in the remaining indeterminates or, for a univariate polynomial, an element of the coefficient ring:

```
p := poly(x^2 + y):
```

```
q := evalp(p, x = 0); evalp(q, y = 2)
```

$$\text{poly}\left(y,[y]\right)$$

$$2$$

You can also use the more general evaluation operator | for the same effect:

```
p | x = 0; p | [x = 0, y = 2]
```

$$\text{poly}\left(y,[y]\right)$$

$$2$$

You may also regard a polynomial as a function of the indeterminates and call this function with arguments:

```
p(2, z)
```
$$z + 4$$

A variety of MuPAD functions accept polynomials as input. An important operation is factorization, which is performed according to the rules of the coefficient ring. You can do factorization with the MuPAD function `factor`:

```
factor(poly(x^3 - 1))
```
$$\text{poly}\left(x - 1, [x]\right) \cdot \text{poly}\left(x^2 + x + 1, [x]\right)$$

```
factor(poly(x^2 + 1, Dom::IntegerMod(2)))
```
$$\text{poly}\left(x + 1, [x], \mathbb{Z}_2\right)^2$$

The function `D` differentiates polynomials:

```
D(poly(x^7 + x + 1))
```
$$\text{poly}\left(7\,x^6 + 1, [x]\right)$$

Equivalently, you may also use `diff(polynomial, x)`.

Integration also works for polynomials:

```
p := poly(x^7 + x + 1): int(p, x)
```
$$\text{poly}\left(\left(\frac{1}{8}\right) x^8 + \left(\frac{1}{2}\right) x^2 + x, [x]\right)$$

The function `gcd` computes the greatest common divisor of polynomials:

```
p := poly((x + 1)^2*(x + 2)):
```
```
q := poly((x + 1)*(x + 2)^2):
```
```
factor(gcd(p, q))
```
$$\text{poly}\left(x + 2, [x]\right) \cdot \text{poly}\left(x + 1, [x]\right)$$

The internal representation of a polynomial stores only those powers of the indeterminates with non-zero coefficients. This is particularly advantageous for "sparse" polynomials of high degree and makes arithmetic with such polynomials efficient. The function `nterms` returns the number of non-zero terms of a polynomial. The function `nthmonomial` extracts individual monomials (coefficient

times powers of the indeterminates), `nthcoeff` and `nthterm` return the appropriate coefficient and product of powers of the indeterminates, respectively:

```
p := poly(a*x^100 + b*x^10 + c, [x]):
```

```
nterms(p), nthmonomial(p, 2),
nthcoeff(p, 2), nthterm(p, 2)
```

$$3,\ \text{poly}\left(b\,x^{10}, [x]\right),\ b,\ \text{poly}\left(x^{10}, [x]\right)$$

Table 4.6 is a summary of the operations for polynomials discussed above. Section "Functions for Polynomials" of the MuPAD Quick Reference lists further functions for polynomials in the standard library. The `groebner` library comprises functions for handling multivariate polynomial ideals (see `?groebner`).

Exercise 4.41: Consider the polynomials $p = x^7 - x^4 + x^3 - 1$ and $q = x^3 - 1$. Compute $p - q^2$. Does q divide p? Factor p and q.

Exercise 4.42: A polynomial is called irreducible (over a coefficient field) if it cannot be factored into a product of more than one nonconstant polynomials. The function `irreducible` tests a polynomial for irreducibility. Find all irreducible quadratic polynomials $a\,x^2 + b\,x + c\,,\ a \neq 0$ over the field of integers modulo 3.

`+, -, *, ^`	: arithmetic
`coeff`	: extract coefficients
`degree`	: polynomial degree
`diff, D`	: differentiation
`divide`	: division with remainder
`evalp`	: evaluation
`expr`	: conversion to expression
`factor`	: factorization
`gcd`	: greatest common divisor
`mapcoeffs`	: apply a function
`multcoeffs`	: multiplication by a scalar
`nterms`	: number of non-zero coefficients
`nthcoeff`	: n-th coefficient
`nthmonomial`	: n-th monomial
`nthterm`	: n-th term
`poly`	: construct a polynomial
`poly2list`	: conversion to list

Table 4.6: MuPAD® functions operating on polynomials

Hardware Float Arrays

Usually, floating-point computations in computer algebra systems are performed in "software floats," providing user-controllable precision (in MuPAD®, via the environment variable `DIGITS`) at the expense of memory and speed. There are cases, however, where especially the memory requirements of software floats are prohibitive, especially when importing data from an external source, such as image files. For such occasions, MuPAD version 5 introduced a new data type, arrays of hardware floating point numbers, `DOM_HFARRAY`.

A hardware floating point array is created by the function `hfarray`, which mimics `array` (cf. Chapter 4.9) closely. It can contain only floating point numbers (real and/or complex), and completely ignores the values of `DIGITS` for its computations, which are performed on hardware floats with whatever precision the underlying hardware and/or operating system provide. (Usually, this means something around 15 decimal digits.) Accessing elements converts them to and from software floats:

```
A := hfarray(1..10, 1..10, [frandom() $ i = 1..100]):
```

```
domtype(A), domtype(A[1, 1])
```

DOM_HFARRAY, DOM_FLOAT

Most of the `numeric` library accepts hardware floating point arrays as inputs and returns objects of the same type:

```
B := numeric::fft(A): domtype(B)
```

DOM_HFARRAY

```
numeric::eigenvalues(A)
```

$[5.173121022, 0.8239615051, 0.3962245484 + 0.3259514491\,i, \ldots]$

In addition to standard array operations, hf-arrays allow addition of arrays of the same size as well as matrix multiplication for two-dimensional hf-arrays of compatible sizes:

```
A := hfarray(1..3, 1..4, [$ 1..12]);
B := hfarray(1..4, 1..2, [1/(i+j) $ j = 1..2 $ i= 1..4])
```

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 5.0 & 6.0 & 7.0 & 8.0 \\ 9.0 & 10.0 & 11.0 & 12.0 \end{pmatrix}$$

$$\begin{pmatrix} 0.5 & 0.3333333333 \\ 0.3333333333 & 0.25 \\ 0.25 & 0.2 \\ 0.2 & 0.1666666667 \end{pmatrix}$$

```
A + A; A*B
```

$$\begin{pmatrix} 2.0 & 4.0 & 6.0 & 8.0 \\ 10.0 & 12.0 & 14.0 & 16.0 \\ 18.0 & 20.0 & 22.0 & 24.0 \end{pmatrix}$$

$$\begin{pmatrix} 2.716666667 & 2.1 \\ 7.85 & 5.9 \\ 12.98333333 & 9.7 \end{pmatrix}$$

Interval Arithmetic

There is another kernel data type, called `DOM_INTERVAL`. Objects of this type represent real or complex intervals of floating-point numbers. They provide, among other possibilities, a means of controlling one of the most fundamental problems of floating-point arithmetic: round-off errors.

The basic idea is as follows: Instead of floating-point numbers x_1, x_2 etc., where almost each operation leads to (usually small) errors, consider intervals X_1, X_2 etc. which are known to contain the precise numbers coming from the application. One would like to have a verified statement that the value $y = f(x_1, x_2, \dots)$ of a function f lies in some interval Y. Mathematically, the image set

$$Y = f(X_1, X_2, \dots) = \{f(\xi_1, \xi_2, \dots);\ \xi_1 \in X_1;\ \xi_2 \in X_2;\ \dots\}$$

is wanted. Computing this image set *exactly* is a formidable task. In fact, it is too ambitious to ask for an exact representation when using fast and memory efficient floating-point arithmetic. Instead, the interval version of a function f is an algorithm $\hat{f}$ that produces a larger set $\hat{f}(X_1, X_2, \dots)$ which is *guaranteed* to contain the exact image set of f:

$$f(X_1, X_2, \dots) \subset \hat{f}(X_1, X_2, \dots).$$

If we think about the intervals X_1, X_2 etc. as incorporating the inaccuracies in x_1, x_2 etc. caused by round-off, the interval version $\hat{f}$ of the function f provides verified upper and lower bounds for the result $y = f(x_1, x_2, \dots)$ in which the round-off errors of x_1, x_2 etc. have propagated. Alternatively, you can also regard $X_1, \dots$ as imprecise measurements or in some other way underdetermined quantity ("this parameter is somewhere between 0 and 1," "this resistor has a tolerance of 10%"). In such a setting, the result $\hat{f}(X_1, X_2, \dots)$ is a set *guaranteed* to contain *all* possible true results.

Floating-point intervals are created from exact numbers or floating-point numbers using the function `hull` or its operator equivalent `...`:

```
X1 := hull(PI)
```

$$3.141592653 \dots 3.141592654$$

```
X2 := cos(7*PI/9 ... 13*PI/9)
```

$$-1.0 \dots -0.1736481776$$

Complex intervals are rectangular areas in the complex plain consisting of an interval for the real part and an interval for the imaginary part. Using `hull` or `...`, you may specify the rectangle by its "lower left" and "upper right" corners:

```
X3 := (2 - 3*I) ... (4 + 5*I)
```

$$(2.0 \ldots 4.0) + \mathrm{i}\,(-3.0 \ldots 5.0)$$

The MuPAD® functions that support interval arithmetic include the basic arithmetical operations `+`, `-`, `*`, `/`, `^` as well as most of the special functions such as `sin`, `cos`, `exp`, `ln`, `abs` etc.:

```
X1^2 + X2
```

$$8.869604401 \ldots 9.695956224$$

```
X1 - I*X2 + X3
```

$$(5.141592653 \ldots 7.141592654) + \mathrm{i}\,(-2.826351823 \ldots 6.0)$$

```
sin(X1) + exp(abs(X3))
```

$$7.389056098 \ldots 603.728285$$

When dividing by an interval containing 0, infinite intervals are produced. The objects `RD_INF` and `RD_NINF` ("rounded infinity" and "rounded negative infinity," respectively) represent the values $\pm\infty$ in an interval context:

```
sin(X2^2 - 1/2)
```

$$-0.4527492553 \ldots 0.4794255387$$

```
1/%
```

$$\mathrm{RD\_NINF} \ldots -2.208728094 \cup 2.085829642 \ldots \mathrm{RD\_INF}$$

The last example shows that the arithmetic may produce "symbolic" unions of floating-point intervals. Actually, the union is still of type `DOM_INTERVAL`:

```
domtype(%)
```

DOM_INTERVAL

In fact, the functions `union` and `intersect` for creating unions and intersections of sets can be used for creating intervals (which are, after all, a special kind of sets):

```
X1 union X2^2
```

$$0.0301536896 \ldots 1.0 \cup 3.141592653 \ldots 3.141592654$$

```
cos(X1*X2) intersect X2
```

$$-1.0 \ldots -0.1736481776$$

Symbolic objects such as identifiers (Chapter 4.3) and floating point intervals can be mixed. The function `interval` replaces all numerical subexpressions of an expression (Chapter 4.4) by floating-point intervals:

```
interval(2*x^2 + PI)
```

$$(2.0 \ldots 2.0)\ x^2 + (3.141592653 \ldots 3.141592654)$$

In MuPAD, identifiers are implicitly assumed to represent arbitrary complex values. Consequently, the function `hull` replaces `x` by the interval representing the whole complex plane:

```
hull(%)
```

$$(\text{RD\_NINF} \ldots \text{RD\_INF}) + \mathrm{i}\ (\text{RD\_NINF} \ldots \text{RD\_INF})$$

There is a number of specialized functions for floating-point intervals attached as methods to the kernel domain `DOM_INTERVAL`. Here, we only mention `DOM_INTERVAL::center` (the center of an interval or a union of intervals) and `DOM_INTERVAL::width` (the width of an interval or a union of intervals):

```
DOM_INTERVAL::center(2 ... 5 union 7 ... 9)
```

$$5.5$$

```
DOM_INTERVAL::width(2 ... 5 union 7 ... 9)
```

$$7.0$$

See `?DOM_INTERVAL` for a survey of the available methods.

Further, a library domain `Dom::FloatIV` exists which is just a façade for the floating-point intervals of the kernel type `DOM_INTERVAL`. It is useful for embedding floating-point intervals in "containers" such as matrices (Chapter 4.15) or polynomials (Chapter 4.16). These containers require the specificiation of a coefficient domain that has the mathematical properties of a ring. In order to

make floating-point intervals embeddable in such containers, the domain `Dom::FloatIV` was given a variety of mathematical attributes ("categories") required for such purposes:

```
Dom::FloatIV::allCategories()
  [Cat::Field, ...,  Cat::Ring, ...]
```

In particular, the set of all floating point intervals is not only regarded as a ring, but even as a field. Strictly speaking, however, these mathematical categories are not really adequate for interval objects. For example, subtracting an interval from itself does not yield the zero element (the neutral element with respect to addition):

```
(2 ... 3) - (2 ... 3)
```

$$-1.0 \ldots 1.0$$

Pragmatically, however, the mathematical categories were set, anyway, to enable embedding. After all, the operations implied by these categories do work. For example, we consider the inverse of a Hilbert matrix (also see Exercise 4.37). These matrices are notoriously ill-conditioned. For the 8×8 Hilbert matrix, the condition number (the ratio of the largest and the smallest eigenvalue) is

```
A := linalg::hilbert(8):
ev := numeric::eigenvalues(A):
max(op(ev))/min(op(ev))
```

$$15257575299.0$$

Roughly speaking, when inverting this matrix by a floating-point algorithm, one should expect to loose about 10 decimal digits of accuracy:

```
log(10, %)
```

$$10.18348552$$

Consequently, with low values of `DIGITS`, the inverse should be substantially marred by round-off. Indeed, after conversion of `A` to a matrix of floating-point intervals, we can use the generic algorithm for matrix inversion:

```
B := 1/Dom::Matrix(Dom::FloatIV)(A)
    array(1..8, 1..8,
      (1, 1) = -73.29144677 ... 201.2914468,
             ...
      (3, 2) = -955198.1290 ... -949921.8709,
             ...
      (8, 8) = 176679046.2 ... 176679673.8
    )
```

The entries of the inverse are *guaranteed* to lie in the indicated intervals. The component $(8, 8)$ is determined to a precision of 6 leading decimal digits, whereas the component $(1, 1)$ is only known to lie somewhere between -73.29 and 201.3. Note, however, that the generic inversion algorithm tends to overestimate the intervals drastically. The results returned by a numerical inversion with "standard" floating-point numbers *might* actually be more accurate than predicted by this interval computation.

The exact components of the inverse are available, too. All entries of inverse Hilbert matrices are integers:

```
C := linalg::invhilbert(8)
    array(1..8, 1..8,
      (1, 1) = 64,
             ...
      (3, 2) = -952560,
             ...
      (8, 8) = 176679360
    )
```

Using the operator `in`, it is possible to determine whether a number or expression is inside an interval. To combine the matrices, we use `zip` (Chapter 4.15) and check for each entry of the exact inverse `C` if it is inside the interval in the matrix `B`. To this end, the following input converts both matrices into lists of their entries and then performs the check:

```
zip([op(C)], [op(B)], (c, b) -> bool(c in b))
    [TRUE, TRUE, TRUE, TRUE, ..., TRUE]
```

Null Objects: null(), NIL, FAIL, undefined

There are several objects representing the "void" in MuPAD®. First, we have the "empty sequence" generated by `null()`. It is of domain type `DOM_NULL` and generates no output on the screen. System functions such as `reset` (Section 13.4), `print` (Section 12.1), or `if` (Chapter 16) without an `else` branch, which cannot return mathematically useful values, return this MuPAD object instead:

```
a := reset(): b := print("hello"):
```

```
    "hello"
```

```
domtype(a), domtype(b)
```

$$\text{DOM\_NULL},\ \text{DOM\_NULL}$$

The object `null()` is particularly useful in connection with sequences (Section 4.5). The system automatically removes this object from sequences, and you can use it, for example, to remove sequence entries selectively:

```
delete a, b, c:
Seq := a, b, c: Seq := subs(Seq, b = null())
```

$$a,\ c$$

Here we have used the substitution command `subs` (Chapter 6) to replace `b` by `null()`.

The MuPAD object `NIL`, which is distinct from `null()`, intuitively means "no value." Some system functions return the `NIL` object when you call them with arguments for which they need not compute anything. A typical example are uninitialized operands of arrays:

```
A := array(1..2): op(A, 1)
```

```
    NIL
```

Uninitialized local variables and parameters of MuPAD procedures also have the value `NIL` (Section 17.4).

The MuPAD object `FAIL` intuitively means "I could not find a value." System functions return this object when there is no meaningful result for the given input parameters.

In the following example, we try to compute the inverse of a singular matrix:

```
A := matrix([[1, 2], [2, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

```
A^(-1)
```

FAIL

Another object with a similar meaning is `undefined`. For example, the MuPAD function `limit` returns this object when the requested limit does not exist:

```
limit(1/x, x = 0)
```

undefined

Arithmetical operations with `undefined` return again `undefined`:

```
undefined + 1, 2^undefined
```

undefined, undefined

5

Evaluation and Simplification

Identifiers and Their Values

Consider:

```
delete x, a: y := a + x
```

$$a + x$$

Since the identifiers `a` and `x` only represent themselves, the "value" of `y` is the symbolic expression $a + x$. We have to distinguish carefully between the identifier `y` and its value. More precisely, the *value* of an identifier denotes the MuPAD® object that the system computes by evaluation and simplification of the right-hand side of the assignment `identifier:=value` *at the time of assignment*.

Note that in the example above, the value of `y` is composed of the symbolic identifiers `a` and `x`, which may be assigned values at a later time. For example, if we assign the value 1 to the identifier `a`, then `a` is replaced by its value 1 in the expression `a+x`, and the call `y` returns $x + 1$:

```
a := 1: y
```

$$x + 1$$

We say that the *evaluation* of the identifier `y` returns the result $x + 1$, but its *value* is still `a+x`:

> We distinguish between an identifier, its value, and its evaluation: the *value* denotes the evaluation *at the time of assignment*, a later *evaluation* may return a different *"current value"*.

If we now assign the value 2 to x, then both a and x are replaced by their values at the next evaluation of y. Hence we obtain the sum $2+1$ as a result, which MuPAD automatically simplifies to 3:

```
x := 2: y
```
$$3$$

The *evaluation* of y now returns the number 3; its *value* is still $a+x$.

It is reasonable to say that the value of y is the result at the time of assignment. Namely, if we delete the values of the identifiers a and x in the above example, then the evaluation of y yields its original value immediately after the assignment:

```
delete a, x: y
```
$$a+x$$

If a or x already have a value *before* we assign the expression a+x to y, then the following happens:

```
x := 1: y := a + x: y
```
$$a+1$$

At the time of assignment, y is assigned the evaluation of a+x, i.e., $a+1$. Indeed, this is now the *value* of y, which contains no reference to x:

```
delete x: y
```
$$a+1$$

Here are some further examples for this mechanism. We first assign the rational number 1/3 to x, then we assign the object [x, x^2, x^3] to the identifier list. In the assignment, the system evaluates the right-hand side and automatically replaces the identifier x by its value. Thus, at the time of assignment, the identifier list gets the value $[1/3, 1/9, 1/27]$ and not $[x, x^2, x^3]$:

```
x := 1/3: list := [x, x^2, x^3]
```
$$\left[\frac{1}{3}, \frac{1}{9}, \frac{1}{27}\right]$$

```
delete x: list
```

$$\left[\frac{1}{3}, \frac{1}{9}, \frac{1}{27}\right]$$

MuPAD applies the same evaluation scheme to symbolic function calls:

```
delete f: y := f(PI)
```

$$f(\pi)$$

After the assignment

```
f := sin:
```

we obtain the evaluation

```
y
```

$$0$$

When evaluating `y`, the system replaced the identifier `f` by its value, which is the value of the identifier `sin`. This is a procedure which is executed when `y` is evaluated and returns $\sin(\pi)$ as 0.

Complete, Incomplete, and Enforced Evaluation

We consider once again the first example from the previous section. There we have assigned the expression `a+x` to the identifier `y`, and `a` and `x` did not have a value:

```
delete a, x: y := a + x: a := 1: y
```

$$x + 1$$

We now explain in greater detail how MuPAD® performs the final evaluation.

First ("level 1") the evaluator considers the value $a + x$ of `y`. Since this value contains identifiers `x` and `a`, a second evaluation step ("level 2") is necessary to determine the value of these identifiers. The system recognizes that `a` has the value 1, while `x` has no value (and thus mathematically represents an unknown). Now the system's arithmetic combines these results to $x + 1$, and this is the evaluation of `y`. Figures 5.1–5.3 illustrate this process. A box represents an identifier and its value (or ·, respectively, if it has no value). An arrow represents one evaluation step.

Figure 5.1: The identifier `y` without a value.

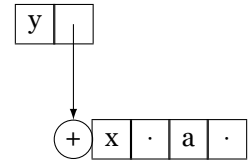


Figure 5.2: After the assignment `y:=a+x`.

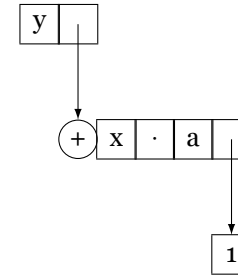


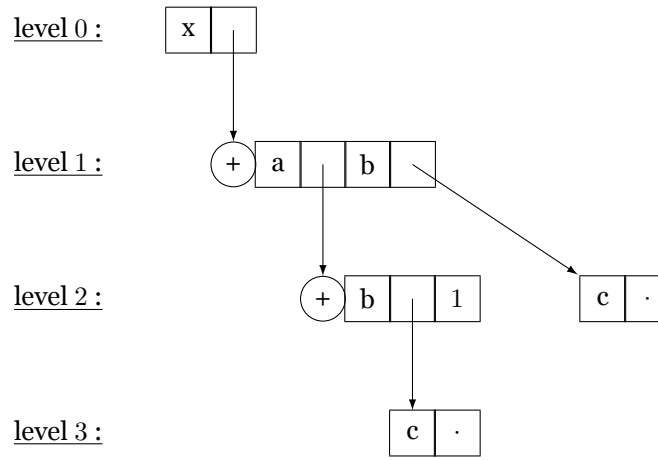
Figure 5.3: After the assignment `a:=1`, we finally obtain `x+1` as the evaluation of `y`.

In analogy to the expression trees for representing symbolic expressions (Section 4.4), you may imagine the process of evaluation as an *evaluation tree*, whose vertices are expressions with symbolic identifiers, with branches pointing to the corresponding values of these identifiers. The system traverses this tree until there are either no identifiers left or all remaining identifiers have no value.

The user may control the levels of this tree via the system function `level`. We look at an example:

```
delete a, b, c: x := a + b: a := b + 1: b := c:
```

The evaluation tree for x is:



The identifier x forms the top level (the root, level 0) of its own evaluation tree:

```
level(x, 0)
```

$$x$$

The next level 1 determines the value of x:

```
level(x, 1)
```

$$a + b$$

At the following level 2, a and b are replaced by their values $b + 1$ and c, respectively:

```
level(x, 2)
```

$$b + c + 1$$

The remaining `b` is replaced by its value c only in the next level 3:

```
level(x, 3)
```
$$2\,c+1$$

We call the type of evaluation described here a *complete evaluation*. This means that identifiers are replaced by their values recursively until no further evaluations are possible. The environment variable `LEVEL`, which has the default value 100, determines how far MuPAD descends at most in an evaluation tree.

In interactive mode, MuPAD always evaluates completely!

More precisely, this means that MuPAD evaluates up to depth `LEVEL` in interactive mode.[1]

```
delete a0, a1, a2: LEVEL := 2:
```
```
a0 := a1: a0
```
$$a1$$
```
a1 := a2: a0
```
$$a2$$

Up to now, the evaluation tree for `a0` has depth 2, and the `LEVEL` value of 2 achieves a complete evaluation. However, in the next step, the value of `a2` is not taken into account:

```
a2 := a3: a0
```
$$a2$$

To restore the default value of `LEVEL`, we use `delete`:

```
delete LEVEL:
```

As soon as MuPAD realizes that the current evaluation level exceeds the value of the environment variable `MAXLEVEL` (whose default value is 100), then it assumes to be in an infinite loop and aborts the evaluation with an error message:

[1] One should not confuse this with the effect of a system function call, which may return a *not completely evaluated object*, such as `subs` (Chapter 6). The call `subs(sin(x), x=0)`, for example, returns $\sin(0)$ and not 0! The functionality of `subs` is to perform a substitution and to return the resulting object without further evaluation.

```
MAXLEVEL := 2: a0
  Error: Recursive definition [See ?MAXLEVEL]
delete MAXLEVEL:
```

We now present some important exceptions to the rule of complete evaluation. The calls `last(i)`, `%i`, or `%` (Chapter 13.2) do not lead to an evaluation! We consider the example

```
delete x: [sin(x), cos(x)]: x := 0:
```

Now, `%2` accesses the list without evaluating it:

```
%2
```

$$[\sin(x), \cos(x)]$$

However, you can enforce evaluation by means of `eval`:

```
eval(%)
```

$$[0, 1]$$

Compare this to the following statements, where requesting the identifier `list` causes the usual complete evaluation:

```
delete x: list := [sin(x), cos(x)]: x := 0: list
```

$$[0, 1]$$

Arrays of domain type `DOM_ARRAY` are always evaluated with level 1:

```
delete a, b: A := array(1..2, [a, b]):
b := a: a := 1: A
```

$$\begin{pmatrix} a & b \end{pmatrix}$$

As you can see, the call of `A` returns the value (the array), but does not replace `a`, `b` by their values. You can evaluate the entries via `map(A, eval)`:

```
map(A, eval)
```

$$\begin{pmatrix} 1 & 1 \end{pmatrix}$$

Note that in contrast to the above behavior, the indexed access of an individual entry is evaluated completely:

```
A[1], A[2]
```

$$1,\ 1$$

Matrices (of type `Dom::Matrix(·)`), tables (of domain type `DOM_TABLE`), and polynomials (`DOM_POLY`) are treated in the same way as arrays. Moreover, within procedures, MuPAD always evaluates only up to level 1 (Section 17.11). If this is not sufficient, you may control this behavior explicitly by means of `level`.

The command `hold(object)` is similar to `level(object, 0)` and prevents the evaluation of `object`.[2] This may be desirable in many situations. The following function, which cannot be executed for symbolic arguments, yields an example where the (premature) evaluation is undesirable:

```
absValue := X -> (if X >= 0 then X else -X end_if):
```

```
absValue(X)
```

```
  Error: Can't evaluate to boolean [_leequal];
  during evaluation of 'absValue'
```

If you want to numerically integrate this function using `numeric::int`, the obvious input returns the same error:

```
numeric::int(absValue(X), X = -1..1)
```

```
  Error: Can't evaluate to boolean [_leequal];
  during evaluation of 'absValue'
```

If you delay the evaluation of `absValue(X)` by `hold`, the value is computed:

```
numeric::int(hold(absValue)(X), X = -1..1)
```

$$1.0$$

The reason is that `numeric::int` internally substitutes numerical values for `X`, for which `absValue` can be evaluated without problems.

[2] See `?hold` for the exact difference between `hold(object)` and `level(object, 0)`.

Here is another example: like most MuPAD functions, the function `domtype` first evaluates its argument, so that the command `domtype(object)` returns the domain type of the *evaluation* of `object`:

```
x := 1: y := 1: x, x + y, sin(0), sin(0.1)
```

$$1,\ 2,\ 0,\ 0.09983341665$$

```
domtype(x), domtype(x + y), domtype(sin(0)),
domtype(sin(0.1))
```

$$\text{DOM\_INT},\ \text{DOM\_INT},\ \text{DOM\_INT},\ \text{DOM\_FLOAT}$$

Using `hold`, you obtain the domain types of the objects themselves: `x` is an identifier, `x+y` is an expression, and `sin(0)` and `sin(0.1)` are function calls and hence expressions as well:

```
domtype(hold(x)), domtype(hold(x + y)),
domtype(hold(sin(0))), domtype(hold(sin(0.1)))
```

$$\text{DOM\_IDENT},\ \text{DOM\_EXPR},\ \text{DOM\_EXPR},\ \text{DOM\_EXPR}$$

The commands `?level` and `?hold` provide further information from the corresponding help pages.

Exercise 5.1: What are the *values* of the identifiers `x`, `y`, and `z` after the following statements? What is the *evaluation* of the last statement in each case?

```
delete a1, b1, c1, x:
x := a1: a1 := b1: a1 := c1: x
```

```
delete a2, b2, c2, y:
a2 := b2: y := a2: b2 := c2: y
```

```
delete a3, b3, z:
b3 := a3: z := b3: a3 := 10: z
```

Predict the results of the following statement sequences:

```
delete u1, v1, w1:
u1 := v1: v1 := w1: w1 := u1: u1
```

```
delete u2, v2:
u2 := v2: u2 := u2^2 - 1: u2
```

Automatic Simplification

MuPAD® automatically simplifies many objects such as certain function calls or arithmetical expressions with numbers:

```
sin(15*PI), exp(0), (1 + I)*(1 - I)
```

$$0,\ 1,\ 2$$

The same holds true for arithmetic expressions containing `infinity`:

```
2*infinity - 5
```

$$\infty$$

Such automatic simplifications serve for reducing the complexity of expressions without an explicit request by the user:

```
cos(1 + exp((-1)^(1/2)*PI))
```

$$1$$

The user can neither control nor extend the automatic simplifier.

In most cases, however, MuPAD does *not* automatically simplify expressions. The reason is that the system generally cannot decide which is the most reasonable way of simplification. For example, consider the following expression which is not simplified:

```
y := (-4*x + x^2 + x^3 - 4)*(7*x - 5*x^2 + x^3 - 3)
```

$$-\left(x^3 - 5\,x^2 + 7\,x - 3\right)\,\left(-x^3 - x^2 + 4\,x + 4\right)$$

Naturally, you can expand this expression, which may be reasonable, for example, before computing its symbolic integral:

```
expand(y)
```

$$x^6 - 4\,x^5 - 2\,x^4 + 20\,x^3 - 11\,x^2 - 16\,x + 12$$

However, if you are interested in the roots of the polynomial, it makes more sense to compute its linear factors:

```
factor(y)
```

$$(x-3)\cdot(x-1)^2\cdot(x-2)\cdot(x+2)\cdot(x+1)$$

No universal answer is possible to the question which of the two representations is "simpler." Depending on the intended application, you can selectively apply appropriate system functions (such as `expand` or `factor`) to simplify an expression.

There is another argument against automatic simplification. The symbol f, for example, might represent a bounded function, for which the limit $\lim_{x\to 0} x\,f(x)$ is 0. However, simplifying this expression to 0 can be wrong for functions f with a singularity at the origin such as $f(x) = 1/x$! Thus, automatic simplifications such as $0\cdot f(0) = 0$ are questionable as long as the system has no additional knowledge about the symbols involved. In general, MuPAD cannot know which rule can be applied and which must not. Now you might object that MuPAD should perform no simplifications at all instead of wrong ones. Unfortunately, this is not reasonable either, since in symbolic computations, expressions tend to grow very quickly, and this seriously affects the performance of the system. In fact, in most cases MuPAD simplifies an expression of the form `0*y` to `0`, except, e.g., when the value of `y` is `infinity`, `FAIL`, or `undefined`. You should always keep in mind that such a simplified result may be wrong in extreme cases.

For that reason, MuPAD performs only some simplifications automatically, and you must explicitly request other simplifications yourself. For this purpose MuPAD provides a variety of functions, some of which are described in Section 9.2.

While the automatic simplifications have been carefully chosen, there are still examples where they lead to unexpected results, usually because of ambiguities in mathematical notation. An example is the solution of the equation $x/x = 1$ for $x \neq 0$:

```
solve(x/x = 1, x)
```

$$\mathbb{C}$$

The MuPAD equation solver `solve` (Chapter 8) returns the set $\mathbb{C}$ of all complex numbers.[3] Thus, `solve` claims that *arbitrary* complex values of x yield a solution of the equation $x/x = 1$. The reason is that the system first automatically simplifies the expression `x/x` to `1`, and then in fact solves the equation $1 = 1$. The exceptional case $x = 0$, for which the original problem makes no sense, is completely ignored in the simplified output!

[3] To enter this set, type "`C_`."

Evaluation at a Point

There is also another concept commonly known as “evaluation,” namely replacing some placeholder (free variable) by a specific value and simplifying the resulting expression. This operation is implemented by the function `evalAt`, which is also accessible as the `|` operator. (The `|` operator mimics the notation $f(x)|_{x=1}$ commonly used in math texts.)

```
f := sin(x):
f | x=0, f | x=1
```

$$0,\ \sin(1)$$

Note that, unlike `subs`, `evalAt` respects the mathematical meaning and only substitutes free variables, not bound ones:

```
delete f:
F := int(f(x), x=0..infinity) + sin(x):
subs(F, x = 1), F | x=1
```

$$\sin(1) + \int_0^\infty f(1)\ \mathrm{d}1,\ \sin(1) + \int_0^\infty f(x)\ \mathrm{d}x$$

6

Substitution: subs, subsex, and subsop

All MuPAD® objects consist of operands (Section 4.1). An important feature of a computer algebra system is that it can replace these building blocks by new values. For that purpose, MuPAD provides the functions `subs`, `subsex` (short for: substitute expression), and `subsop` (short for: substitute operand).

The command `subs(object, Old=New)` replaces all occurrences of the subexpression `Old` in `object` by the value `New`:

```
f := a + b*c^b: g := subs(f, b = 2):  f, g
```

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

You see that `subs` returns the result of the substitution, but the identifier `f` remains unchanged. If you represent a map F by the expression $f = F(x)$, then you may use `subs` to evaluate the function at some point:

```
f := 1 + x + x^2:
```

```
subs(f, x = 0), subs(f, x = 1),
subs(f, x = 2), subs(f, x = 3)
```

$$1,\ 3,\ 7,\ 13$$

Note, however, that `subs` (as well as the commands `subsex` and `subsop` to be handled shortly) performs a purely syntactical replacement, which may lead to incorrect results when evaluating a function in this way:

```
f := x/sin(x): subs(f, x = 0)
```

$$0$$

```
delete g: G := int(g(x), x=0..infinity):
subs(G, x=0)
```

$$\int_0^\infty g(0)\,\mathrm{d}0$$

To get substitutions respecting mathematical meaning, use the | operator or its equivalent function evalAt instead, which has already been described at the end of Chapter 5:

```
f | x=0
```

```
  Error: Division by zero [_power];
  during evaluation of 'evalAt'
```

```
G | x=0
```

$$\int_0^\infty g(x)\,\mathrm{d}x$$

The output of the subs command is subjected to the usual simplifications of the internal simplifier. In the above example, the call subs(f, x=0) produces the object 1+0+0^2, which is automatically simplified to 1. You must not confuse this with *evaluation* (Chapter 5), where in addition all identifiers in an expression are replaced by their values.

> The function subs performs a substitution. The system only simplifies the resulting object, but does not evaluate it upon return!

In the following example

```
f := x + sin(x): g := subs(f, x = 0)
```

$$\sin(0)$$

the identifier sin for the sine function is not replaced by the corresponding MuPAD function, which would return sin(0)=0. Only the next call to g performs a complete evaluation:

```
g
```

$$0$$

You can enforce evaluation by using `eval`:

```
eval(subs(f, x = 0))
```

$$0$$

You may replace arbitrary MuPAD objects by substitution. In particular, you can substitute functions or procedures as new values:

```
eval(subs(h(a + b), h = (x -> 1 + x^2)))
```

$$(a+b)^2+1$$

If you want to replace a system function, enclose its name in a `hold` command:

```
eval(subs(sin(a + b), hold(sin) = (x -> x - x^3/3)))
```

$$a+b-\frac{(a+b)^3}{3}$$

You can also replace more complex subexpressions:

```
subs(sin(x)/(sin(x) + cos(x)), sin(x) + cos(x) = 1)
```

$$\sin(x)$$

You should be careful with such substitutions: the command `subs(object,Old=New)` replaces all those occurrences of the expression `Old` *that can be found by means of* `op`. This explains why nothing happens in the following example:

```
subs(a + b + c, a + b = 1), subs(a * b * c, a * b = 1)
```

$$a+b+c,\ a\,b\,c$$

Here the sum `a+b` and the product `a*b` are *not* operands of the corresponding expressions. Even worse, we find:

```
f := a + b + sin(a + b): subs(f, a + b = 1)
```

$$a+b+\sin(1)$$

Again, you cannot obtain the subexpression `a+b` of the outer sum by means of `op`. However, the argument of the sine is the sub-operand `op(f, [3, 1])` (see Sections 4.1 and 4.4), and hence it is replaced by `1`.

In contrast to subs, the function subsex also replaces subexpressions in sums and products:

```
subsex(f, a + b = x + y), subsex(a * b * c, a * b = x + y)
```

$$x + y + \sin(x + y),\ c\ (x + y)$$

This kind of substitution requires a closer analysis of the expression tree, and hence subsex is much slower than subs for large objects. When replacing more complex subexpressions, you should not be misled by the screen output of expressions:

```
f := a/(b*c)
```

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

```
subs(f, b*c = New), subsex(f, b*c = New)
```

$$\frac{a}{b\,c},\ \frac{a}{\mathrm{New}}$$

If you look at the operands of f, you see that the expression tree does not contain the product b*c. This explains why no substitution took place in the subs call:

```
op(f)
```

$$a,\ \frac{1}{b},\ \frac{1}{c}$$

You can perform several substitutions with a single call of subs:

```
subs(a + b + c, a = A, b = B, c = C)
```

$$A + B + C$$

This is equivalent to the nested call

```
subs(subs(subs(a + b + c, a = A), b = B), c = C):
```

Thus we obtain:

```
subs(a + b^2, a = b, b = a)
```

$$a^2 + a$$

First, MuPAD replaces a by b, yielding b+b^2. Then it substitutes a for b in this new expression and returns the above result. In contrast, you may achieve a

simultaneous substitution by specifying the substitution equations in form of a list or a set:

```
subs(a + b^2, [a = b, b = a]),
subs(a + b^2, {a = b, b = a})
```

$$a^2 + b,\ a^2 + b$$

The output of the equation solver `solve` (Chapter 8) supports the functionality of `subs`. In general, `solve` returns lists of equations, which may be used in `subs`:

```
equations := {x + y = 2, x - y = 1}:
```

```
solution := solve(equations, {x, y})
```

$$\left\{\left[x = \frac{3}{2}, y = \frac{1}{2}\right]\right\}$$

```
subs(equations, op(solution, 1))
```

$$\{1 = 1, 2 = 2\}$$

The function `subsop` provides another variant of substitution: `subsop(object, i=New)` selectively replaces the i-th operand of the object by the value `New`:

```
subsop(2*c + a^2, 2 = d^5)
```

$$d^5 + 2\,c$$

Here, we have replaced the second operand `a^2` of the sum by `d^5`. In the following example, we first replace the exponent of the second term (this is the operand `[2, 2]` of the sum), and then the first term:

```
subsop(2*c + a^2, [2, 2] = 4, 1 = x*y)
```

$$a^4 + x\,y$$

In the following expression, we first replace the first term, yielding the expression `x*y+c^2`. Then we substitute `z` for the second factor of the first term (which now is `y`):

```
subsop(a*b + c^2, 1 = x*y, [1, 2] = z)
```

$$c^2 + x\,z$$

The expression `a+2` is a symbolic sum, which has a 0-th operand, namely, the system function `_plus` for generating sums:

```
op(a + 2, 0)
```

_plus

You can replace this operand by any other function (for example, by the system function `_mult` which multiplies its arguments):

```
subsop(a + 2, 0 = _mult)
```

$$2\,a$$

When using `subsop`, you need to know the position of the operand that you want to replace. Nonetheless, you should be cautious, since the system may change the order of the operands when this is mathematically valid (for example, in sums, products, or sets):

```
set := {sin(1 + a), a, b, c^2}
```

$$\left\{a, b, \sin(a+1), c^2\right\}$$

If you use `subs`, you need not know the position of the subexpression. Another difference between `subs` and `subsop` is that `subs` traverses the expression tree of the object *recursively*, and thus also replaces suboperands:

```
subs(set, a = a^2)
```

$$\left\{b, \sin\left(a^2+1\right), a^2, c^2\right\}$$

Exercise 6.1: Does the command `subsop(b+a, 1=c)` replace the identifier `b` by `c`?

Exercise 6.2: The commands

```
delete f: g := diff(f(x)/diff(f(x), x), x $ 5)
  25 diff(f(x), x, x) diff(f(x), x, x, x, x)
  ------------------------------------------ -
                                2
                 diff(f(x), x)

     4 diff(f(x), x, x, x, x, x)
     --------------------------- - ...
            diff(f(x), x)
```

generate a lengthy expression containing symbolic derivatives. Make this expression more readable by replacing these derivatives by simpler names $f_0 = f(x)$, $f_1 = f'(x)$, etc.

7

Differentiation and Integration

We have already used the MuPAD® commands for differentiation and integration. Since they are important, we recapitulate the usage of these routines here.

Differentiation

The call `diff(expression, x)` computes the derivative of the expression with respect to the unknown `x`:

```
diff(sin(x^2), x)
```

$$2\,x\,\cos\left(x^2\right)$$

If the expression contains symbolic calls to functions whose derivative is not known, then `diff` returns itself symbolically:

```
diff(x*f(x), x)
```

$$x\,\frac{\partial}{\partial x}f(x)+f(x)$$

You may compute higher derivatives via `diff(expression, x, x, ...)`. The sequence `x, x, ...` of identifiers may be generated conveniently via the sequence operator `$` (Section 4.5):

```
diff(sin(x^2), x, x, x) = diff(sin(x^2), x $ 3)
```

$$-12\,x\,\sin\left(x^2\right)-8\,x^3\,\cos\left(x^2\right)=-12\,x\,\sin\left(x^2\right)-8\,x^3\,\cos\left(x^2\right)$$

You can compute partial derivatives in the same way. MuPAD® assumes that mixed partial derivatives of symbolic expressions are symmetric:

```
diff(f(x,y), x, y) - diff(f(x,y), y, x)
```

$$0$$

If a mathematical map is represented by a function instead of an expression, then the differential operator `D` computes the derivative as a function:

```
D(sin), D(exp), D(ln), D(sin*cos), D(sin@ln), D(f+g)
```

$$\cos,\ \exp,\ \frac{1}{\mathrm{id}},\ \cos^2-\sin^2,\ \frac{\cos\circ\ln}{\mathrm{id}},\ f'+g'$$

```
f := x -> (sin(ln(x))): D(f)
```

$$x\to\frac{\cos(\ln(x))}{x}$$

Here, `id` denotes the identity map $x \to x$. The expression `D(f)(x)` returns the value of the derivative at a point:

```
D(f)(1), D(f)(y^2), D(g)(0)
```

$$1,\ \frac{\cos(\ln(y^2))}{y^2},\ g'(0)$$

The system converts the prime `'` for the derivative to a call of `D`:

```
f'(1), f'(y^2), g'(0)
```

$$1,\ \frac{\cos(\ln(y^2))}{y^2},\ g'(0)$$

For a function with more than one argument, `D([i], f)` is the partial derivative with respect to the i-th argument, and `D([i, j, ...], f)` is equivalent to `D([i], D([j], ...))`, for higher partial derivatives.

Exercise 7.1: Consider the function $f : x \to \sin(x)/x$. Compute first the value of f at the point $x = 1.23$, and then the derivative $f'(x)$. Why does the following input not yield the desired result?

```
f := sin(x)/x: x := 1.23: diff(f, x)
```

Exercise 7.2: De l'Hospital's rule states that

$$\lim_{x\to x_0} \frac{f(x)}{g(x)} = \lim_{x\to x_0} \frac{f'(x)}{g'(x)} = \cdots = \lim_{x\to x_0} \frac{f^{(k-1)}(x)}{g^{(k-1)}(x)} = \frac{f^{(k)}(x_0)}{g^{(k)}(x_0)},$$

if $f(x_0) = g(x_0) = \cdots = f^{(k-1)}(x_0) = g^{(k-1)}(x_0) = 0$ and $g^{(k)}(x_0) \neq 0$. Compute $\lim_{x\to 0} \frac{x^3 \sin(x)}{(1-\cos(x))^2}$ by applying this rule interactively. Use the function `limit` to check your result.

Exercise 7.3: Determine the first and second order partial derivatives of $f_1(x_1, x_2) = \sin(x_1\, x_2)$. Let $x = x(t) = \sin(t)$, $y = y(t) = \cos(t)$, and $f_2(x, y) = x^2\, y^2$. Compute the derivative of $f_2(x(t), y(t))$ with respect to t.

Integration

The function `int` features both definite and indefinite integration:

```
int(sin(x), x), int(sin(x), x = 0..PI/2)
```

$$-\cos(x),\ 1$$

If `int` is unable to compute a result, it returns itself symbolically. In the following example, the integrand is split into two terms. Only one of these has an integral that can be represented by elementary functions:

```
int((x - 1)/(x*sqrt(x^5 + 1)), x)
```

$$\int \frac{x-1}{x\sqrt{x^5+1}}\,\mathrm{d}x$$

```
expand(%)
```

$$\int \frac{1}{\sqrt{x^5+1}}\,\mathrm{d}x - \frac{2\,\arctan\left(\sqrt{x^5+1}\,\mathrm{i}\right)\,\mathrm{i}}{5}$$

The function $\mathrm{Si}(x) = \int_0^x \sin(t)/t\,\mathrm{d}t$ is implemented as a special function in MuPAD:

```
int(sin(a*x)/x, x)
```

$$\mathrm{Si}(a\,x)$$

While integrating, the integration variable itself is assumed to be real. Other than this, all computations take place over the complex numbers, and the system regards every symbolic parameter in the integrand as a complex number unless told otherwise. In the following example, the definite integral exists only for specific ranges of the parameter a, and the system returns a symbolic `limit` call:

```
int(sin(a*x)/x, x = 0..infinity)
```

$$\lim_{x\to\infty} \mathrm{Si}(x\,a)$$

You can tell the system that an identifier has certain properties by using the operator `assuming` (Section 9.3). The following call stipulates that a be a positive real number:

```
int(sin(a*x)/x, x = 0..infinity) assuming a > 0
```

$$\frac{\pi}{2}$$

Besides the exact computation of definite integrals, MuPAD also provides several numerical methods:

```
float(int(sin(x)/x, x = 0..2))
```

$$1.605412977$$

In the previous computation, `int` first returns a symbolic result (the error function `erf`), which is then approximated by `float`. If you want to compute numerically from the beginning, then you can suppress the symbolic computation via `int` by using `hold` (Section 5.2):

```
float(hold(int)(sin(x)/x, x = 0..2))
```

$$1.605412977$$

Alternatively, the function `numeric::int` from the `numeric` library can be used:

```
numeric::int(sin(x)/x, x = 0..2)
```

$$1.605412977$$

This function allows you to choose different numerical methods for computing the integral. You find more detailed information using `?numeric::int`. It works in a purely numerical fashion without any symbolic preprocessing of the integrand. For smooth integrands without singularities, `numeric::int` is quite efficient.

Exercise 7.4: Compute the following integrals:

$$\int_0^{\pi/2} \sin(x)\,\cos(x)\,\mathrm{d}x\,, \quad \int_0^1 \frac{\mathrm{d}x}{\sqrt{1-x^2}}\,, \quad \int_0^1 x\,\arctan(x)\,\mathrm{d}x\,.$$

Use MuPAD to verify the following equality: $\displaystyle\int_{-2}^{-1} \frac{\mathrm{d}x}{x} = -\ln(2)\,.$

Exercise 7.5: Use MuPAD to determine the following indefinite integrals:

$$\int^x \frac{t\,\mathrm{d}t}{\sqrt{(2\,a\,t-t^2)^3}}\,, \quad \int^x \sqrt{t^2-a^2}\,\mathrm{d}t\,, \quad \int^x \frac{\mathrm{d}t}{t\,\sqrt{1+t^2}}\,.$$

Exercise 7.6: The function `intlib::changevar` performs a change of variable in a symbolic integral. Read the corresponding help page. MuPAD cannot compute the integral

$$\int_{-\pi/2}^{\pi/2} \sin(x)\,\sqrt{1+\sin(x)}\,\mathrm{d}x.$$

Assist the system by using the substitution `t=sin(x)`. Compare the value that you get to the numerical result returned by the function `numeric::int`.

8

Solving Equations: `solve`

The function `solve` solves equations, inequalities, systems of these, differential equations, recurrence equations etc. This routine can handle various different types of equations. Besides "algebraic" equations, also certain classes of differential and recurrence equations can be solved. Further, there is a variety of specialized solvers that only handle special classes of equations. Many of these algorithms are actually called by the "universal" `solve` once it has identified the type of the equations. The user can also call the specialized solvers directly. You can find a survey of all available MuPAD® solvers in in the "Quick Reference" in the online documentation.

Polynomial Equations

You can supply an individual equation as first argument to `solve`. The unknown for which you want to solve is the second argument:

```
solve(x^2 + x = y/4, x), solve(x^2 + x - y/4 = 0, y)
```

$$\left\{-\frac{\sqrt{y+1}}{2}-\frac{1}{2}, \frac{\sqrt{y+1}}{2}-\frac{1}{2}\right\}, \left\{4x^2+4x\right\}$$

In this case, the system returns a set of solutions. If you specify an expression instead of an equation, then `solve` assumes the equation `expression=0`:

```
solve(x^2 + x - y/4, y)
```

$$\left\{4x^2+4x\right\}$$

For polynomials of higher degree, it is provably impossible to always find a closed form for the solutions by means of radicals etc. In such cases, MuPAD® uses the `RootOf` object:

```
solve(x^7 + x^2 + x, x)
```

$$\{0\} \cup \text{RootOf}(z^6+z+1, z)$$

The above `RootOf` object represents all solutions of the equation `x^6+x+1=0`. You can use `float` to approximate such objects by floating-point numbers. The system internally employs a numerical procedure to determine all (complex) roots of the polynomial:

```
float(%)
  {0.0, 0.9454023333 - 0.6118366938 I,

     - 0.7906671888 - 0.3005069203 I,

     - 0.1547351445 - 1.038380754 I,

     0.9454023333 + 0.6118366938 I,

     - 0.7906671888 + 0.3005069203 I,

     - 0.1547351445 + 1.038380754 I}
```

If you want to solve a collection of equations for possibly several unknowns, specify both the equations and the unknowns as sets or lists. In the following example, we solve two *linear* equations in three unknowns:

```
delete x, y, z:
equations := {x + y + z = 3, x + y = 2}:
```

```
solution := solve(equations, {x, y, z})
```

$$\{[x = 2 - \mathrm{z1}, y = \mathrm{z1}, z = 1]\}$$

If you solve equations for several variables, then MuPAD returns a set of "solved equations" equivalent to the original system of equations. You can now read off the solutions immediately: the unknown z has the value 1, the unknown y may be arbitrary (indicated by the new indeterminate `z1`) and, for any given value of y, we have $x = 2 - y$. It is possible to get a clearer indication of the new parameter not present in the input by giving the option `VectorFormat` to the `solve` command:

```
solve(equations, {x, y, z}, VectorFormat)
```

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \in \left\{ \begin{pmatrix} 2 - \mathrm{z1} \\ \mathrm{z1} \\ 1 \end{pmatrix} \middle| \, \mathrm{z1} \in \mathbb{C} \right\}$$

However, this return format is more difficult to use in subsequent commands.

The call to `solve` does not assign the values to the unknowns; x, y, and z are still unknowns. However, the form of the output as a list of solved equations is chosen in such a way that you can conveniently use `|` (Chapter 5.4) to substitute these values in other objects. For example, you may substitute the solution in the original equations to verify the result:

```
equations | solution[1]
```

$$\{2 = 2, 3 = 3\}$$

You can use `assign(solution[1])` to assign the solution values to the identifiers `x`, `y`, and `z`.

In the next example, we solve two *nonlinear* polynomial equations in several unknowns:

```
equations := {x^2 + y = 1, x - y = 2}:
```

```
solutions := solve(equations, {x, y})
```

$$\left\{\left[x = -\frac{\sqrt{13}}{2} - \frac{1}{2}, y = -\frac{\sqrt{13}}{2} - \frac{5}{2}\right], \left[x = \frac{\sqrt{13}}{2} - \frac{1}{2}, y = \frac{\sqrt{13}}{2} - \frac{5}{2}\right]\right\}$$

MuPAD found two distinct solutions. Again, you can use `|` to substitute the solutions in other expressions:

```
map(equations | solutions[1], expand),
map(equations | solutions[2], expand)
```

$$\{1 = 1, 2 = 2\},\ \{1 = 1, 2 = 2\}$$

Often, solutions are represented by `RootOf` expressions:

```
solve({x^3 + x^2 + 2*x = y, y^2 = x^3}, {x, y})
```

```
  +-   -+    { +-   -+ }
  |  x  |    { |  0  | }
  |     | in { |     | } union
  |  y  |    { |  0  | }
  +-   -+    { +-   -+ }

     { +-                   -+ |
     { |      3              | |
     { |  - z1  + 4 z1 + 4   | |
     { |                     | |
     { |         z1          | |
     { +-                   -+ |

                                                  }
                                                  }
                    4      3      2               }
     z1 in RootOf(z  - 3 z  - 2 z  + 5 z + 8, z) }
                                                  }
                                                  }
```

If you use the option `MaxDegree = n`, then `RootOf` expressions for polynomials of degree up to n are replaced by representations in terms of radicals if this is possible. Note that such solutions tend to be rather complicated:

```
solve({x^3 + x^2 + 2*x = y, y^2 = x^3}, {x, y},
      MaxDegree = 4)
```

$$\left\{\left[x = 7 + \left(\frac{\sqrt{\frac{129\left(\frac{3485}{54} + \frac{\sqrt{3}\sqrt{73}\,5\,i}{18}\right)^{\frac{1}{3}}}{4} + 9\left(\frac{3485}{54} + \frac{\sqrt{3}\sqrt{73}\,5\,i}{18}\right)^{\frac{2}{3}} + 145}}{6\left(\frac{3485}{54} + \frac{\sqrt{3}\sqrt{73}\,5\,i}{18}\right)^{\frac{1}{6}}} + \ldots\right.\right.$$

Specifying the unknowns to solve for is optional. For the special case where you have one unknown, however, the format of the output depends on whether you specify the unknown or not. The general rule is as follows:

> A call of the form `solve(equation, unknown)`, where `unknown` is an identifier, returns a MuPAD object representing a set of numbers. All other forms of `solve` for (one or several) polynomial equations, without special options like `VectorFormat`, return a set of lists of equations or an expression involving `RootOf`.

Here are some examples:

```
solve(x^2 - 3*x + 2 = 0, x), solve(x^2 - 3*x + 2, x)
```

$$\{1, 2\},\ \{1, 2\}$$

```
solve(x^2 - 3*x + 2 = 0), solve(x^2 - 3*x + 2)
```

$$\{[x = 1], [x = 2]\},\ \{[x = 1], [x = 2]\}$$

```
solve({x^2 - 3*x + 2 = 0}, x),
solve({x^2 - 3*x + 2}, x)
```

$$\{1, 2\},\ \{1, 2\}$$

```
solve({x^2 - 3*x + y = 0, y - 2*x = 0}, {x, y})
```

$$\{[x = 0, y = 0], [x = 1, y = 2]\}$$

```
solve({x^2 - 3*x + y = 0, y - 2*x = 0})
```

$$\{[x = 0, y = 0], [x = 1, y = 2]\}$$

If you do not supply unknowns to solve for, solve internally uses the system function indets to find the symbolic identifiers in the equations and regards all of them as unknowns:

```
solve({x + y^2 = 1, x^2 - y = 0})
```

$$\begin{pmatrix} x \\ y \end{pmatrix} \in \left\{ \begin{pmatrix} 1 - \mathrm{z1}^2 \\ \mathrm{z1} \end{pmatrix} \middle| \, \mathrm{z1} \in \mathrm{RootOf}(z^4 - 2\,z^2 - z + 1, z) \right\}$$

By default, solve tries to find all *complex* solutions of the given equation(s). If you want to find only the real solutions of a single equation, use the option Domain=Dom::Real or assume x to be real:

```
solve(x^3 + x = 0, x)
```

$$\{0, -\mathrm{i}, \mathrm{i}\}$$

```
solve(x^3 + x = 0, x) assuming x in R_
```

$$\{0\}$$

Other assumptions, such as x>0, x in Q_, x in Z_ and x>2, etc., are handled as well.

In some (usually exotic) cases, the "real solutions" may include unexpected values:

```
solve(ln(x)^4=PI^4, x) assuming x in R_
```

$$\{-1, \mathrm{e}^{-\pi}, \mathrm{e}^{\pi}\}$$

When replacing the x in our input by -1, we get an $\ln(-1)$, which is $\pi\,\mathrm{i}$, a complex value. In the end, we get a real-valued result, but a common request is to find solutions for a problem restricted to the reals, including all intermediate values ("solving over the reals"). To make solve tackle this problem, use the option Real instead of the assumption:

```
solve(ln(x)^4=PI^4, x, Real)
```

$$\{\mathrm{e}^{-\pi}, \mathrm{e}^{\pi}\}$$

Of course, using the option saves a bit of typing, too.

If all complex numbers satisfy a given equation, then `solve` returns $\mathbb{C}$, the set of all complex numbers:

```
solve(sin(x) = cos(x - PI/2), x)
```

$$\mathbb{C}$$

```
domtype(%)
```

solvelib::BasicSet

There are four such "basic sets": the integers $\mathbb{Z}$ (entered as `Z_` in MuPAD input), the rational numbers $\mathbb{Q}$ (`Q_`), the real numbers $\mathbb{R}$ (`R_`), and the complex numbers $\mathbb{C}$ (`C_`).

You can use the function `float` to find *numerical* solutions. However, with a statement of the form `float(solve(equations, unknowns))`, `solve` first tries to solve the equations symbolically. If an exact solution is found, `float` converts the result to floating-point approximations. If you want to compute in a purely numerical way, you can use `hold` (Section 5.2) to avoid symbolic preprocessing:

```
float(hold(solve)({x^3 + x^2 + 2*x = y, y^2 = x^2},
                  {x, y}))
```

```
  {[x = - 0.5 + 1.658312395 I, y = 0.5 - 1.658312395 I],

     [x = - 0.5 + 0.8660254038 I,

     y = - 0.5 + 0.8660254038 I],

     [x = - 0.5 - 0.8660254038 I,

     y = - 0.5 - 0.8660254038 I],

     [x = - 0.5 - 1.658312395 I, y = 0.5 + 1.658312395 I

     ], [x = 0.0, y = 0.0]}
```

Alternatively, the library `numeric` provides various functions for numerical equation solving such as `numeric::solve`[1], `numeric::fsolve`, or `numeric::realroots`. The help system gives details about these routines: call `?solvers` for a survey or see `?numeric::solve` etc.

[1] In fact, the calls `float(hold(solve)(...))` and `numeric::solve(...)` are equivalent.

Exercise 8.1: Compute the general solution of the system of linear equations

$$\begin{aligned} a + b + c + d + e &= 1, \\ a + 2b + 3c + 4d + 5e &= 2, \\ a - 2b - 3c - 4d - 5e &= 2, \\ a - b - c - d - e &= 3. \end{aligned}$$

How many free parameters does the solution have?

General Equations and Inequalities

`solve` can handle a variety of (non-polynomial) equations. For example, the equation $\exp(x) = 8$ has infinitely many solutions of the form $\ln(8) + 2\,\mathrm{i}\,\pi\,k$ with $k \in \mathbb{Z}$ in the complex plane:

```
S := solve(exp(x) = 8, x)
```

$$\{\ln(8) + 2\,\pi\,k\,\mathrm{i} \mid k \in \mathbb{Z}\}$$

The data type of the returned result is a so-called "image set." It represents a mathematical set of the form $\{f(x) \mid x \in A\}$, where A is some other set:

```
domtype(S)
```

Dom::ImageSet

This data type is capable of representing infinitely many elements.

If you omit the variable to solve for, the system returns a logical formula, using the operator `in`:

```
solve(exp(x) = 8)
```

$$\left(\ x\ \right) \in \left\{ \left(\ \ln(8) + 2\,\pi\,k\,\mathrm{i}\ \right) \middle|\ k \in \mathbb{Z} \right\}$$

You can use `map` to apply a function to an image set:

```
map(S, _plus, -ln(8))
```

$$\{2\,\pi\,k\,\mathrm{i} \mid k \in \mathbb{Z}\}$$

Alternatively, most arithmetical operations can be applied directly:

```
exp(S)
```

$$\{8\}$$

The function `is` (Section 9.3) can handle image sets:

```
S := solve(sin(PI*x/2) = 0, x)
```

$$\{2\,k \mid k \in \mathbb{Z}\}$$

```
is(1 in S), is(4 in S)
```

$$\text{FALSE},\ \text{TRUE}$$

The equation `exp(x) = sin(x)` also has infinitely many solutions, which, however, MuPAD® cannot represent exactly. In this case, it returns the call to `solve` symbolically:

```
solutions := solve(exp(x) = sin(x), x)
```

$$\text{solve}(\mathrm{e}^{x} - \sin(x) = 0, x)$$

Warning: In contrast to polynomial equations, the numerical solver computes at most one solution of a non-polynomial equation:

```
float(solutions)
```

$$\{-226.1946711\}$$

However, you can specify a search range to select a particular numerical solution:

```
numeric::solve(exp(x) = sin(x), x = -10..-9)
```

$$\{-9.424858654\}$$

Using `numeric::realroots`, you can find enclosures for *all* real roots in a given interval:

```
numeric::realroots(exp(x) = sin(x), x = -10..-5)
```

$$[[-9.43359375, -9.423828125], [-6.2890625, -6.279296875]]$$

MuPAD has a special data type for the solution of parametric equations: `piecewise`. For example, the set of solutions $x \in \mathbb{C}$ of the equation $(a\,x^2 - 4)\,(x - b) = 0$ takes on different forms, depending on the value of the parameter a:

```
delete a: p := solve((a*x^2 - 4)*(x - b), x)
```

$$\begin{cases} \{b\} & \text{if } a = 0 \\ \left\{b, -\frac{2}{\sqrt{a}}, \frac{2}{\sqrt{a}}\right\} & \text{if } a \neq 0 \end{cases}$$

```
domtype(p)
```

$$\text{piecewise}$$

The function `map` applies a function to all branches of a `piecewise` object:

```
map(p, _power, 2)
```

$$\begin{cases} \{b^2\} & \text{if} \quad a = 0 \\ \{\frac{4}{a}, b^2\} & \text{if} \quad a \neq 0 \end{cases}$$

After the following substitution, the `piecewise` object is simplified to a set:

```
% | [a = 4, b = 2]
```

$$\{1, 4\}$$

The function `solve` can also handle inequalities. It then returns an interval or a union of intervals, of domain type `Dom::Interval` or image sets, of domain type `Dom::ImageSet`:

```
solve(x^2 < 1, x)
```

$$(-1, 1) \cup \{\, y \,\mathrm{i}|\, y \in \mathbb{R}\}$$

```
domtype(%)
```

DOM_EXPR

```
S := solve(x^2 >= 1, x)
```

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

```
is(-2 in S), is(0 in S)
```

TRUE, FALSE

Differential Equations

The function ode defines an ordinary differential equation. Such an object has two components: an equation and the function to solve for.

```
diffEquation := ode(y'(x) = y(x)^2, y(x))
```

$$\mathrm{ode}\left(y'(x) - y(x)^2, y(x)\right)$$

The following call to solve finds the general solution containing an arbitrary constant C_3:

```
solve(diffEquation)
```

$$\left\{0, -\frac{1}{\mathrm{C3} + x}\right\}$$

Differential equations of higher order can be handled as well:

```
solve(ode(y''(x) = y(x), y(x)))
```

$$\left\{\mathrm{C7}\,\mathrm{e}^{x} + \frac{\mathrm{C6}}{\mathrm{e}^{x}}\right\}$$

You can specify initial conditions or boundary values by passing the differential equation together with the initial/boundary conditions as a set when calling ode:

```
diffEquation :=
ode({y''(x) = y(x), y(0) = 1, y'(0) = 0}, y(x)):
```

MuPAD now adjusts the free constants in the general solution according to the initial/boundary conditions:

```
solve(diffEquation)
```

$$\left\{\frac{1}{2\,\mathrm{e}^{x}} + \frac{\mathrm{e}^{x}}{2}\right\}$$

You can specify systems of equations with several functions in form of a set:

```
solve(ode({y'(x) = y(x) + 2*z(x), z'(x) = y(x)},
          {y(x), z(x)}))
```

$$\left\{\left[z(x) = \frac{\mathrm{C12}\,\mathrm{e}^{2\,x}}{2} - \frac{\mathrm{C11}}{\mathrm{e}^{x}}, y(x) = \frac{\mathrm{C11}}{\mathrm{e}^{x}} + \mathrm{C12}\,\mathrm{e}^{2\,x}\right]\right\}$$

The function `numeric::odesolve` of the `numeric` library solves the ordinary differential equation $Y'(x) = f(x, Y(x))$ with the initial condition $Y(x_0) = Y_0$ numerically. You must supply the right-hand side of the differential equation as a function $f(x, Y)$ of two arguments, where x is a scalar and Y is a vector. If you combine the components y and z in the previous example to a vector $Y = (y, z)$, you can define the right-hand side of the equation

$$\frac{\mathrm{d}}{\mathrm{d}x}\,Y = \frac{\mathrm{d}}{\mathrm{d}x}\begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} y + 2\,z \\ y \end{pmatrix} = \begin{pmatrix} \mathtt{Y[1]} + 2 \cdot \mathtt{Y[2]} \\ \mathtt{Y[1]} \end{pmatrix} =: \mathtt{f(x, Y)}$$

in the form

```
f := (x, Y) -> [Y[1] + 2*Y[2], Y[1]]:
```

Note that $f(x, Y)$ must be a vector. Here, this is realized by means of a list containing the components on the right-hand side of the differential equation. The call

```
numeric::odesolve(0..1, f, [1, 1])
```

$$[9.729448318, 5.04866388]$$

integrates the system of differential equations with the initial values $Y(0) = (y(0), z(0)) = (1, 1)$ over the interval $x \in [0, 1]$; the initial conditions are specified by the list `[1, 1]`. The numerical solver returns the numerical solution vector $Y(1) = (y(1), z(1))$ as a list.

Exercise 8.2: Check the numerical solutions $y(1) = 9.729\ldots$ and $z(1) = 5.048\ldots$ of the system of differential equations

$$y'(x) = y(x) + 2\,z(x)\ ,\ z'(x) = y(x)$$

computed above by substituting the initial values $y(0) = 1$, $z(0) = 1$ in the general symbolic solution, determining the values for the free constants, and evaluating the symbolic solution at $x = 1$.

Exercise 8.3:

1) Compute the general solution $y(x)$ of the differential equation $y' = y^2/x$.

2) Determine the solution $y(x)$ for each of the following initial value problems:

$$a) \quad y' - y\,\sin(x) = 0\,, \qquad y'(1) = 1,$$

$$b) \quad 2\,y' + \frac{y}{x} = 0\,, \qquad y'(1) = \pi\,.$$

3) Find the general solution of the following system of ordinary differential equations in $x(t), y(t), z(t)$:

$$x' = y\,z,\; y' = x\,z,\; z' = t\,z.$$

Recurrence Equations

Recurrence equations are equations for functions depending on a discrete parameter (an "index"). You can generate such an object with the function `rec`, whose arguments are an equation, the function to be determined and, optionally, a set of initial conditions:

```
equation := rec(y(n + 2) = y(n + 1) + 2*y(n), y(n)):
```

```
solve(equation)
```

$$\{(-1)^n \, \mathrm{C1} + 2^n \, \mathrm{C2}\}$$

The general solution contains two arbitrary constants (C_1, C_2 in this case), which are suitably adjusted when you specify initial conditions:

```
solve(rec(y(n + 2) = 2*y(n) + y(n + 1), y(n),
          {y(0) = 1}))
```

$$\{2^n \, \mathrm{C4} - (-1)^n \, (\mathrm{C4} - 1)\}$$

```
solve(rec(y(n + 2) = 2*y(n) + y(n + 1), y(n),
          {y(0) = 1, y(1) = 1}))
```

$$\left\{\frac{(-1)^n}{3} + \frac{2\,2^n}{3}\right\}$$

Exercise 8.4: The Fibonacci numbers are defined by the recurrence $F_n = F_{n-1} + F_{n-2}$ with the initial values $F_0 = 0,\ F_1 = 1$. Use `solve` to find an explicit representation for F_n.

9

Manipulating Expressions

When evaluating objects, MuPAD® automatically performs a variety of simplifications. For example, arithmetic operations between integers are executed or $\exp(\ln(x))$ is simplified to x. Other simplifications such as $\sin(x)^2 + \cos(x)^2 = 1$, $\ln(\exp(x)) = x$, $(x^2 - 1)/(x - 1) = x + 1$, or $\sqrt{x^2} = x$ do not happen automatically. One reason is that many of these rules are not universally valid: for example, $\sqrt{x^2} = x$ is wrong for $x = -2$. Other simplifications such as $\sin(x)^2 + \cos(x)^2 = 1$ are valid universally, but there would be a significant loss of efficiency if MuPAD would always scan expressions for the occurrence of sin and cos terms.

Moreover, it is not clear in general which of several mathematically equivalent representations is the most appropriate. For example, it might be reasonable to replace an expression such as $\sin(x)$ by its complex exponential representation

$$\sin(x) = -\frac{\mathrm{i}}{2}\ \exp(x\,\mathrm{i}) + \frac{\mathrm{i}}{2}\ \exp(-x\,\mathrm{i}).$$

In such a situation, you can control the manipulation and simplification of expressions by explicitly applying appropriate system functions. MuPAD provides the following functions, which we have partly discussed in Section 2.4:

`collect`	:	collecting coefficients
`combine`	:	combining subexpressions
`expand`	:	expansion
`factor`	:	factorization
`normal`	:	normalization of rational expressions
`partfrac`	:	partial fraction decomposition
`radsimp`	:	simplification of radicals
`rectform`	:	Cartesian representation of complex values
`rewrite`	:	applying mathematical identities
`simplify`	:	universal simplifier
`Simplify`	:	universal simplifier

Transforming Expressions

If you enter the command `collect(expression, unknown)`, the system regards the expression as a polynomial in the specified unknown and groups the coefficients of equal powers:

```
x^2 + a*x + sqrt(2)*x + b*x^2 + sin(x) + a*sin(x):
```

```
collect(%, x)
```

$$(b+1)\,x^2+\left(a+\sqrt{2}\right)\,x+(\sin(x)+a\,\sin(x))$$

You can specify several "unknowns," which may themselves be expressions, as a list:

```
collect(%, [x, sin(x)])
```

$$(b+1)\,x^2+\left(a+\sqrt{2}\right)\,x+(a+1)\,\sin(x)$$

The function `combine(expression, option)` combines subexpressions using mathematical identities between functions indicated by `option`. The implemented options are `arctan`, `exp`, `ln`, `log`, `sincos`, `sinhcosh`, and `gamma`.

By default, `combine` only employs the identities $a^b\,a^c=a^{b+c}$, $a^c\,b^c=(a\,b)^c$, and $(a^b)^c=a^{bc}$ for powers, where they are valid[1]:

```
f := x^(n + 1)*x^PI/x^2: f = combine(f)
```

$$\frac{x^{\pi}\,x^{n+1}}{x^2}=x^{\pi+n-1}$$

```
f := a^x*3^y/2^x/9^y: f = combine(f)
```

$$\frac{3^y\,a^x}{2^x\,9^y}=\left(\frac{1}{3}\right)^y\left(\frac{a}{2}\right)^x$$

```
combine(sqrt(6)*sqrt(7)*sqrt(x))
```

$$\sqrt{42\,x}$$

[1] An example where they are not is $((-1)^2)^{1/2}\neq -1$.

```
f := (PI^(1/2))^x: f = combine(f)
```

$$\sqrt{\pi}^{x} = \pi^{\frac{x}{2}}$$

If $|xy| < 1$, the inverse `arctan` of the tangent function satisfies the following identity:

```
f := arctan(x) + arctan(y):
f = combine(f, arctan) assuming 0 < x*y < 1
```

$$\arctan(x) + \arctan(y) = -\arctan\left(\frac{x+y}{x\,y-1}\right)$$

For the exponential function, we have $\exp(x)\exp(y) = \exp(x+y)$:

```
combine(exp(x)*exp(y)^2/exp(-z), exp)
```

$$\mathrm{e}^{x+2\,y+z}$$

Note, however, that the identity $\exp(x)^y = \exp(x\,y)$ known to hold for real values of x does not hold throughout the complex plane. Consequently, without additional assumptions on x, MuPAD® does not combine such terms:

```
combine(exp(x)^y, exp)
```

$$(\mathrm{e}^{x})^{y}$$

```
combine(exp(x)^y, exp) assuming x in R_
```

$$\mathrm{e}^{x\,y}$$

With certain assumptions about x and y, the logarithm satisfies the rules $\ln(x) + \ln(y) = \ln(x\,y)$ and $x\,\ln(y) = \ln(y^x)$:

```
combine(ln(x) + ln(2) + 3*ln(3/2), ln)
```

$$\ln\left(\frac{27\,x}{4}\right)$$

The trigonometric functions satisfy a variety of identities that the system employs to combine products:

```
combine(sin(x)*cos(y), sincos),
combine(sin(x)^2, sincos)
```

$$\frac{\sin(x-y)}{2}+\frac{\sin(x+y)}{2},\ \frac{1}{2}-\frac{\cos(2\,x)}{2}$$

Similar rules are applied to the hyperbolic functions:

```
combine(sinh(x)*cosh(y), sinhcosh),
combine(sinh(x)^2, sinhcosh)
```

$$\frac{\sinh(x-y)}{2}+\frac{\sinh(x+y)}{2},\ \frac{\cosh(2\,x)}{2}-\frac{1}{2}$$

The function `expand` applies the identities used by `combine` in the reverse direction: it transforms special function calls with composite arguments to sums or products of function calls with simpler arguments via "addition theorems:"

```
expand(x^(y + z)), expand(exp(x + y - z + 4)),
expand(ln(2*PI*x*y))
```

$$x^{y}\,x^{z},\ \frac{\mathrm{e}^{4}\,\mathrm{e}^{x}\,\mathrm{e}^{y}}{\mathrm{e}^{z}},\ \ln(2)+\ln(\pi)+\ln(x\,y)$$

```
expand(sin(x + y)), expand(cosh(x + y))
```

$$\cos(x)\,\sin(y)+\cos(y)\,\sin(x)\,,\ \cosh(x)\,\cosh(y)+\sinh(x)\,\sinh(y)$$

```
expand(sqrt(42*x*y))
```

$$\sqrt{42}\,\sqrt{x\,y}$$

Here, the system does not perform certain "expansions" such as $\ln(x\,y)=\ln(x)+\ln(y)$, since such an identity holds only under additional assumptions (for example, for positive real x and y).

The most frequent use of `expand` is for transforming a product of sums into a sum of products:

```
expand((x + y)^2*(x - y)^2)
```

$$x^{4}-2\,x^{2}\,y^{2}+y^{4}$$

The function expand works recursively for all subexpressions:

```
expand((x - y)*(x + y)*sin(exp(x + y + z)))
```

$$x^2 \sin(e^x e^y e^z) - y^2 \sin(e^x e^y e^z)$$

You can supply expressions as additional arguments to expand. These subexpressions are *not* expanded:

```
expand((x - y)*(x + y)*sin(exp(x + y + z)),
       x - y, x + y + z)
```

$$x \sin\left(e^{x+y+z}\right) (x-y) + y \sin\left(e^{x+y+z}\right) (x-y)$$

The function factor factors polynomials and expressions:

```
factor(x^3 + 3*x^2 + 3*x + 1)
```

$$(x+1)^3$$

Here the system factors "over the rational numbers:" it looks for polynomial factors with rational number coefficients. In effect, MuPAD does not return the factorization $x^2 - 2 = (x - \sqrt{2})\,(x + \sqrt{2})$:

```
factor(x^2 - 2)
```

$$\left(x^2 - 2\right)$$

To extend the ring of constants over which to factor, you can explicitly adjoin constants to the field of rational numbers:

```
factor(x^2 - 2, Adjoin = sqrt(2))
```

$$\left(x - \sqrt{2}\right) \cdot \left(x + \sqrt{2}\right)$$

For sums of rational expressions, factor first computes a common denominator and then factors both the numerator and the denominator:

```
f := (x^3 + 3*y^2)/(x^2 - y^2) + 3: f = factor(f)
```

$$\frac{x^3 + 3\,y^2}{x^2 - y^2} + 3 = \frac{x^2 \cdot (x+3)}{(x-y) \cdot (x+y)}$$

MuPAD can factor not only polynomials and rational functions. For more general expressions, the system internally replaces subexpressions such as symbolic

function calls by identifiers, factors the corresponding polynomial or rational function, and re-substitutes the temporary identifiers:

```
factor((exp(x)^2 - 1)/(cos(x)^2 - sin(x)^2))
```

$$\frac{\left(e^{2\,x}-1\right)}{(\cos(x)-\sin(x))\cdot(\cos(x)+\sin(x))}$$

The function `normal` computes a "normal form" for rational expressions. Like `factor`, it first computes a common denominator for sums of rational expressions, but it then expands numerator and denominator instead of factoring them:

```
f := ((x + 6)^2 - 17)/(x - 1)/(x + 1) + 1:
f, factor(f), normal(f)
```

$$\frac{(x+6)^2-17}{(x-1)\,(x+1)}+1,\ \frac{2\cdot(x+3)^2}{(x-1)\cdot(x+1)},\ \frac{2\,x^2+12\,x+18}{x^2-1}$$

Nevertheless, `normal` cancels common factors in numerator and denominator:

```
f := x^2/(x + y) - y^2/(x + y): f = normal(f)
```

$$\frac{x^2}{x+y}-\frac{y^2}{x+y}=x-y$$

Like `factor`, `normal` can handle arbitrary expressions:

```
f := (exp(x)^2-exp(y)^2)/(exp(x)^3 - exp(y)^3):
```

```
f = normal(f)
```

$$\frac{e^{2\,x}-e^{2\,y}}{e^{3\,x}-e^{3\,y}}=\frac{e^{x}+e^{y}}{e^{2\,x}+e^{2\,y}+e^{x}\,e^{y}}$$

The function `partfrac` decomposes a rational expression into a polynomial part plus a sum of rational terms in which the degree of the numerator is smaller than the degree of the corresponding denominator (partial fraction decomposition):

```
f := x^2/(x^2 - 1): f = partfrac(f, x)
```

$$\frac{x^2}{x^2-1}=\frac{1}{2\,(x-1)}-\frac{1}{2\,(x+1)}+1$$

The denominators of the terms are the factors that MuPAD finds when factoring

the common denominator:

```
denominator := x^5 + x^4 - 7*x^3 - 11*x^2 - 8*x - 12:
```

```
factor(denominator)
```

$$(x-3)\cdot\left(x^2+1\right)\cdot(x+2)^2$$

```
partfrac(1/denominator, x)
```

$$\frac{\frac{9\,x}{250}-\frac{13}{250}}{x^2+1}-\frac{1}{25\,(x+2)}-\frac{1}{25\,(x+2)^2}+\frac{1}{250\,(x-3)}$$

Another function for manipulating expressions is `rewrite`. It employs identities to eliminate certain functions completely from an expression and replaces them by other functions. For example, you can always express sin and cos by tan with the half argument:

$$\sin(x)=\frac{2\,\tan(x/2)}{1+\tan(x/2)^2}\,,\quad \cos(x)=\frac{1-\tan(x/2)^2}{1+\tan(x/2)^2}\,.$$

The trigonometric functions are also related to the complex exponential function:

$$\sin(x)=-\frac{\mathrm{i}}{2}\,\exp(\mathrm{i}\,x)+\frac{\mathrm{i}}{2}\,\exp(-\mathrm{i}\,x)\,,$$

$$\cos(x)=\frac{1}{2}\,\exp(\mathrm{i}\,x)+\frac{1}{2}\,\exp(-\mathrm{i}\,x)\,.$$

You can express the hyperbolic functions and their inverse functions in terms of the exponential function and the logarithm:

$$\sinh(x)=\frac{\exp(x)-\exp(-x)}{2}\,,\quad \cosh(x)=\frac{\exp(x)+\exp(-x)}{2}\,,$$

$$\operatorname{arcsinh}(x)=\ln(x+\sqrt{x^2+1})\,,\quad \operatorname{arccosh}(x)=\ln(x+\sqrt{x^2-1})\,.$$

A call of the form `rewrite(expression, target)` employs these identities. Table 9.1 lists the rules implemented in MuPAD.

```
rewrite(D(D(u))(x), diff)
```

$$\frac{\partial^2}{\partial x^2}u(x)$$

target	: *function(s)*	→ *rewritten in terms of*
`andor`	: logical operators `xor`, `==>`, `<=>`	→ `and`, `or`, `not`
`arccos`	: inverse trig. functions	→ `arccos`
`arccosh`	: inverse trig. functions, `ln`	→ `arccosh`
`arccot`	: inverse trig. functions	→ `arccot`
`arccoth`	: inverse trig. functions, `ln`	→ `arccoth`
`arcsin`	: inverse trig. functions	→ `arcsin`
`arcsinh`	: inverse trig. functions, `ln`	→ `arcsinh`
`arctan`	: inverse trig. functions	→ `arctan`
`arctanh`	: inverse trig. functions, `ln`	→ `arctanh`
`bernoulli`	: `euler`	→ `bernoulli`
`cos`	: exponential function `exp`, trig. and hyperbolic functions	→ `cos`
`cosh`	: exponential function `exp`, trig. and hyperbolic functions	→ `cosh`
`cot`	: exponential function `exp`, trig. and hyperbolic functions	→ `cot`
`coth`	: exponential function `exp`, trig. and hyperbolic functions	→ `coth`
`diff`	: differential operator `D`	→ `diff`
`D`	: differentiating function `diff`	→ `D`
`exp`	: powers (`^`), trig. and hyperbolic functions and their inverses, polar angle `arg`, `dawson`	→ `exp`, `ln`
`erf`	: `dawson`	→ `erf`
`fact`	: Γ-function `gamma`, double factorial `fact2`, binomial coefficients `binomial`, β-function `beta`, `pochhammer`	→ `fact`
`gamma`	: factorial `fact`, double factorial `fact2`, binomial coefficients `binomial`, β-function `beta`, `pochhammer`	→ `gamma`

Table 9.1a: Targets of `rewrite`

target	: *function(s)*	→ *rewritten in terms of*
`harmonic`	: `psi`	→ `harmonic`
`heaviside`	: `sign`	→ `heaviside`
`Im`	: `Re`	→ `Im`
`ln`	: inverse trig. and inverse hyperbolic functions, polar angle `arg`, `log`	→ `ln`
`lambertW`	: `wrightOmega`	→ `lambertW`
`max`	: `min`, `abs`	→ `max`
`min`	: `max`, `abs`	→ `min`
`piecewise`	: `sign`, absolute value `abs`, step function `heaviside`, maximum `max`, minimum `min`, `kroneckerDelta`	→ `piecewise`
`psi`	: `harmonic`	→ `psi`
`Re`	: `Im`	→ `Re`
`sign`	: step function `heaviside`, absolute value `abs`	→ `sign`
`sin`	: exponential function `exp`, trig. and hyperbolic functions	→ `sin`
`sincos`	: exponential function `exp`, trig. and hyperbolic functions	→ `sin`, `cos`
`sinh`	: exponential function `exp`, trig. and hyperbolic functions	→ `sinh`
`sinhcosh`	: exponential function `exp`, trig. and hyperbolic functions	→ `sinh`, `cosh`
`tan`	: exponential function `exp`, trig. and hyperbolic functions	→ `tan`
`tanh`	: exponential function `exp`, trig. and hyperbolic functions	→ `tanh`

Table 9.1b: Targets of `rewrite` *(cont.)*

```
rewrite(sin(x)/cos(x), exp) = rewrite(tan(x), exp)
```

$$\frac{\frac{e^{-x\,i}}{2} - \frac{e^{x\,i}}{2}}{\frac{e^{-x\,i}}{2} + \frac{e^{x\,i}}{2}} = -\frac{e^{2\,x\,i}\,i - i}{e^{2\,x\,i} + 1}$$

```
rewrite(arcsinh(x) - arccosh(x), ln)
```

$$\ln\left(x + \sqrt{x^2+1}\right) - \ln\left(x + \sqrt{x^2-1}\right)$$

For expressions representing complex *numbers*, you can easily compute real and imaginary parts by using `Re` and `Im`:

```
z := 2 + 3*I: Re(z), Im(z)
```

$$2,\ 3$$

```
z := sin(2*I) - ln(-1): Re(z), Im(z)
```

$$0,\ \sinh(2) - \pi$$

When an expression contains symbolic identifiers, MuPAD assumes all such unknowns to be complex values. `Re` and `Im` are returned symbolically:

```
Re(a*b + I), Im(a*b + I)
```

$$\Re(a\,b)\,,\ \Im(a\,b) + 1$$

In such a case, you can use the function `rectform` (short for: rectangular form) to decompose the expression into real and imaginary part. The name of this function is derived from the fact that it computes the coordinates of the usual rectangular (Cartesian) coordinate system. MuPAD decomposes the symbols contained in the expression into their real and imaginary parts and expresses the final result accordingly:

```
rectform(a*b + I)
```

$$\Re(a)\ \Re(b) - \Im(a)\ \Im(b) + \Im(a)\ \Re(b)\ i + \Im(b)\ \Re(a)\ i + i$$

```
rectform(exp(x))
```

$$\cos(\Im(x))\ e^{\Re(x)} + \sin(\Im(x))\ e^{\Re(x)}\,i$$

Again, you can extract the real and imaginary parts of the result with `Re` and `Im`, respectively:

```
Re(%), Im(%)
```

$$\cos(\Im(x))\ e^{\Re(x)},\ \sin(\Im(x))\ e^{\Re(x)}$$

As a basic principle, `rectform` regards all symbolic identifiers as representatives of complex numbers. However, you can use `assume` (Section 9.3) to specify that an identifier represents only real numbers:

```
assume(a in R_):
z := rectform(a*b + I)
```

$$a\,\Re(b) + a\,\Im(b)\ \mathrm{i} + \mathrm{i}$$

The results of `rectform` have a special data type:

```
domtype(z)
```

rectform

You can use the function `expr` to convert such an object to a "normal" expression of domain type `DOM_EXPR`:

```
expr(z)
```

$$a\,\Im(b)\ \mathrm{i} + a\,\Re(b) + \mathrm{i}$$

Simplifying Expressions

In some cases, a transformation leads to a simpler expression:

```
f := 2^x*3^x/8^x/9^x: f = combine(f)
```

$$\frac{2^x\,3^x}{8^x\,9^x} = \left(\frac{1}{12}\right)^x$$

```
f := x/(x + y) + y/(x + y): f = normal(f)
```

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

To this end, however, you must inspect the expression and decide yourself which function to use for simplification. There are tools for applying various simplification algorithms to an expression *automatically*: the functions `simplify` and `Simplify`. These are universal simplifiers which MuPAD® uses to achieve a representation of an expression that is as "simple" as possible:

```
f := 2^x*3^x/8^x/9^x: f = simplify(f)
```

$$\frac{2^x\,3^x}{8^x\,9^x} = \frac{1}{2^{2\,x}\,3^x}$$

```
f := (1 + (sin(x)^2 + cos(x)^2)^2)/sin(x):
```

```
f = simplify(f)
```

$$\frac{\left(\cos(x)^2 + \sin(x)^2\right)^2 + 1}{\sin(x)} = \frac{2}{\sin(x)}$$

```
f := x/(x + y) + y/(x + y) - sin(x)^2 - cos(x)^2:
```

```
f = simplify(f)
```

$$\frac{x}{x+y} - \sin(x)^2 - \cos(x)^2 + \frac{y}{x+y} = 0$$

```
f := (exp(x) - 1)/(exp(x/2) + 1): f = simplify(f)
```

$$\frac{\mathrm{e}^x - 1}{\mathrm{e}^{\frac{x}{2}} + 1} = \mathrm{e}^{\frac{x}{2}} - 1$$

```
f := sqrt(997) - (997^3)^(1/6): f = simplify(f)
```

$$\sqrt{997} - 991026973^{\frac{1}{6}} = 0$$

Note that `simplify` operates in a purely heuristic way since there is no general answer what "simple" means. You can control the simplification process by supplying additional arguments. As in the case of `combine`, you can request particular simplifications by means of options. For example, you can tell the simplifier explicitly to simplify expressions containing square roots:

```
f := sqrt(4 + 2*sqrt(3)):
f = simplify(f, sqrt)
```

$$\sqrt{2}\sqrt{\sqrt{3}+2} = \sqrt{3}+1$$

The possible options are `exp`, `ln`, `log`, `cos`, `sin`, `cosh`, `sinh`, `sqrt`, `gamma`, `unit`, `condition`, `logic`, and `relation`. Internally, `simplify` then confines itself to those simplification rules that are valid for the function given as option. The options `logic` and `relation` are for simplifying logical expressions and equations and inequalities, respectively (see also the corresponding help page: `?simplify`).

Instead of `simplify(expression, sqrt)`, you may also use the function `radsimp` to simplify numerical expressions containing square roots or other radicals:

```
f = radsimp(f)
```

$$\sqrt{2}\sqrt{\sqrt{3}+2} = \sqrt{3}+1$$

```
f := 2^(1/4)*2 + 2^(3/4) - sqrt(8 + 6*2^(1/2)):
```

```
f = radsimp(f)
```

$$2\,2^{\frac{1}{4}} - \sqrt{2}\sqrt{3\sqrt{2}+4} + 2^{\frac{3}{4}} = 0$$

In many cases, using `simplify` without options is appropriate. However, such a call is often very time consuming, since the simplification algorithm is quite complex. It may be useful to specify additional options to save computing time, since then simplifications are performed only for special functions.

The second general simplifier, `Simplify` (note the capital `S`), is often slower than `simplify`, but much more powerful and allows much finer tuning:

```
Simplify(1/(x+y) + 1/(x-y)),
Simplify(1/(x+y) + 1/(x-y), Valuation=length)
```

$$\frac{2\,x}{x^2-y^2},\ \frac{1}{x-y}+\frac{1}{x+y}$$

```
g := gamma(n + n!/gamma(n+1)):
g, simplify(g), Simplify(g)
```

$$\Gamma\left(n+\frac{n!}{\Gamma\left(n+1\right)}\right),\ \Gamma\left(\frac{n!+n\,\Gamma\left(n+1\right)}{\Gamma\left(n+1\right)}\right),\ \Gamma\left(n+1\right)$$

```
assume(n, Type::PosInt):
f := (1 + I)^n + (1 - I)^n:
Simplify(f), Simplify(f, Steps = 350)
```

$$(1-\mathrm{i})^n+(1+\mathrm{i})^n,\ 2\,2^{\frac{n}{2}}\,\cos\left(\frac{\pi\,n}{4}\right)$$

Here, we have used the option `Steps` to tell `Simplify` how many "elementary" steps it may try before giving up the search for a simpler expression. In the end, almost all of the steps have led nowhere; we can ask `Simplify` for a list of steps it performed and see that only four steps were actually used:

```
Simplify(f, Steps = 350, OutputType = "Proof")
  Input was
    ((1 - I))^n + ((1 + I))^n
  Lemma:
  ((1 - I))^n = 2^(n/2)*exp(-(PI*n*I)/4)
  Input was
    ((1 - I))^n
  Applying the rule
    #X -> abs(op(#X, 1))^op(#X, 2)*exp(I*op(#X, 2)*arg(\
  op(#X, 1)))
  gives
    2^(n/2)*exp(-(PI*n*I)/4)
  End of lemma

  substituting gives
    2^(n/2)*exp(-(PI*n*I)/4) + ((1 + I))^n
  Lemma:
  ((1 + I))^n = 2^(n/2)*exp((PI*n*I)/4)
  Input was
    ((1 + I))^n
  Applying the rule
    #X -> abs(op(#X, 1))^op(#X, 2)*exp(I*op(#X, 2)*arg(\
  op(#X, 1)))
  gives
    2^(n/2)*exp((PI*n*I)/4)
  End of lemma

  substituting gives
    2^(n/2)*exp(-(PI*n*I)/4) + 2^(n/2)*exp((PI*n*I)/4)
  Applying the rule
    Simplify::rewriteTrig
  gives
    2*exp((n*ln(2))/2)*cos((PI*n)/4)
  Applying the rule
    Simplify::expand
  gives
    2*2^(n/2)*cos((PI*n)/4)
  END OF PROOF
```

As this (admittedly difficult to read) output suggests, `Simplify` works by following a *rule base*. The documentation, accessible at `?Simplify (details)` (note the space and the parentheses!), shows examples with application-specific rule sets.

As we have seen above in the first example, the user can also control `Simplify`'s idea of what is "simple." Exercise 9.4 shows an application.

Exercise 9.1: It is possible to rewrite products of trigonometric functions as linear combinations of sin and cos terms whose arguments are integral multiples of the original arguments (Fourier expansion). Find constants a, b, c, d, and e such that

$$\cos(x)^2 + \sin(x)\,\cos(x) = a + b\,\sin(x) + c\,\cos(x) + d\,\sin(2\,x) + e\,\cos(2\,x)$$

holds.

Exercise 9.2: Use MuPAD to prove the following identities:

1) $$\frac{\cos(5\,x)}{\sin(2\,x)\,\cos^2(x)} = -5\,\sin(x) + \frac{\cos^2(x)}{2\,\sin(x)} + \frac{5\,\sin^3(x)}{2\,\cos^2(x)}\,,$$

2) $$\frac{\sin^2(x) - e^{2\,x}}{\sin^2(x) + 2\,\sin(x)\,e^x + e^{2\,x}} = \frac{\sin(x) - e^x}{\sin(x) + e^x}\,,$$

3) $$\frac{\sin(2\,x) - 5\,\sin(x)\,\cos(x)}{\sin(x)\,(1 + \tan^2(x))} = -\frac{9\,\cos(x)}{4} - \frac{3\,\cos(3\,x)}{4}\,,$$

4) $$\sqrt{14 + 3\sqrt{3 + 2\sqrt{5 - 12\sqrt{3 - 2\sqrt{2}}}}} = \sqrt{2} + 3\,.$$

Exercise 9.3: MuPAD computes the following integral for `f`:

```
f := sqrt(sin(x) + 1): int(f, x)
```

$$\frac{2\,(\sin(x) - 1)\,\sqrt{\sin(x) + 1}}{\cos(x)}$$

Its derivative is not literally identical to the integrand:

```
diff(%, x)
```

$$\frac{\sin(x)-1}{\sqrt{\sin(x)+1}}+2\sqrt{\sin(x)+1}+\frac{2\,\sin(x)\,(\sin(x)-1)\,\sqrt{\sin(x)+1}}{\cos(x)^2}$$

Simplify this expression.

Exercise 9.4: Using the option `Valuation`, we may pass a function to `Simplify` to determine which expressions are "simple:" the function is called with an expression as its argument and returns a number, higher numbers for more complex expressions.

Let us try to write $\tan(x)-\cot(x)$ without the tangent and cotangent functions. The first, somewhat naive approach is to use a valuation function that simply looks for the presence of `tan` and `cot`:[2]

```
noTangent := x -> if has(x, [hold(tan), hold(cot)])
                  then 10
                  else 1 end_if:
Simplify(tan(x) - cot(x), Valuation = noTangent)
```

$$\frac{2\left(\frac{e^{-x\,i}\,i}{2}-\frac{e^{x\,i}\,i}{2}\right)}{e^{-x\,i}+e^{x\,i}}-\frac{2\left(\frac{e^{-x\,i}}{2}+\frac{e^{x\,i}}{2}\right)^2}{\left(\frac{e^{-x\,i}\,i}{2}-\frac{e^{x\,i}\,i}{2}\right)\left(e^{-x\,i}+e^{x\,i}\right)}$$

Now, this expression actually does not contain `tan`, but is hardly "simple." Improve `noTangent` such that the call above returns a simple expression without `tan` and `cot`. You might want to inform yourself about `length` first or use `Simplify::defaultValuation` instead, which is the function used by `Simplify` if nothing is specified.

[2] See Chapter 16 for an explanation of "`if`."

Assumptions about Mathematical Properties

MuPAD® performs transformations or simplifications for objects containing symbolic identifiers only if the corresponding rules apply in the entire complex plane. However, some familiar rules for computing with real numbers are not generally valid for complex numbers. For example, the square root and the logarithm are branches of multi-valued complex functions, and the MuPAD functions internally make certain assumptions about the branch cuts.

$\ln(e^x) = x$	for real x
$\ln(x^n) = n\,\ln(x)$	for real $x > 0$
$\ln(x\,y) = \ln(x) + \ln(y)$	for real $x > 0$ or real $y > 0$
$\sqrt{x^2} = \lvert x \rvert$	for real x
$e^{x/2} = (e^x)^{1/2}$	for real x

You can use the function `assume` to tell the system functions such as `expand`, `simplify`, `limit`, `solve`, and `int` that they may make certain assumptions about the meaning of certain expressions. We only demonstrate some simple examples here. You find more information on the corresponding help page: `?assume`.

You can use an expression such as an inequality or containment in a set, including the basic sets of the solver (cf. page 8-7) to tell MuPAD that a symbolic identifier or expression represents only values corresponding to the mathematical meaning of the type. For example, the commands

```
assume(x in R_): assume(y in R_): assume(n in Z_):
```

restrict x and y to be real numbers and n to be an integer. Now `simplify` can apply additional rules:

```
simplify(ln(exp(x))), simplify(sqrt(x^2))
```

$$x,\ x\,\mathrm{sign}(x)$$

The command

```
assume(x > 0):
```

restricts x to positive real numbers. After this assumption, one obtains:

```
simplify(ln(x^n)), simplify(ln(x*y) - ln(x) - ln(y)),
simplify(sqrt(x^2))
```

$$n \ln(x),\ 0,\ x$$

Transformations and simplifications with constants are executed without additional assumptions since their mathematical meaning is known:

```
expand(ln(2*PI*z)), sqrt((2*PI*z)^2)
```

$$\ln(2) + \ln(\pi) + \ln(z),\ 2\,\pi\,\sqrt{z^2}$$

The arithmetic operators take into account certain mathematical properties automatically:

```
(a*b)^m
```

$$(a\,b)^m$$

```
assume(m in Z_): (a*b)^m
```

$$a^m\,b^m$$

The function `is` checks whether a MuPAD expression is known to be true:

```
is(1 in Z_), is(PI + 1 in R_)
```

TRUE, TRUE

In addition, `is` takes into account the mathematical properties set by `assume`:

```
delete x: is(x in Z_)
```

UNKNOWN

```
assume(x in Z_):
is(x in Z_), is(x in R_)
```

TRUE, TRUE

Here, the Boolean value `UNKNOWN` expresses the fact that the system cannot decide whether `x` represents an integer or not.

In contrast to `is`, the function `testtype` presented in Section 14.1 checks the *technical* type of a MuPAD object:

```
testtype(x, Type::Integer), testtype(x, DOM_IDENT)
```

$$\text{FALSE},\ \text{TRUE}$$

Queries of the following form are also possible:

```
assume(y > 5): is(y + 1 > 4)
```

$$\text{TRUE}$$

To set an assumption only temporarily during a single command, use the `assuming` operator:

```
limit(exp(a*x), x=infinity);
limit(exp(a*x), x=infinity) assuming a < 0
```

$$\begin{cases} 1 & \text{if } a = 0 \\ \infty & \text{if } 0 < a \\ 0 & \text{if } \Re(a) < 0 \\ \lim_{x \to \infty} \mathrm{e}^{a\,x} & \text{if } \Re(a) = 0 \wedge a \neq 0 \vee 0 < \Re(a) \wedge \neg 0 < a \end{cases}$$

$$0$$

The function `getprop` returns a mathematical property of an identifier or an expression, i.e., a set such that $x \in \mathrm{getprop}(x)$:

```
getprop(y), getprop(y^2 + 1)
```

$$(5, \infty),\ (26, \infty)$$

You can delete the properties of an identifier using `unassume` or the keyword `delete`:

```
delete y: is(y > 5)
```

$$\text{UNKNOWN}$$

If none of the sub-expressions in an expression has any properties attached to it, then `getprop` may return the set of complex numbers:

```
getprop(y), getprop(y^2 + 1), getprop(abs(y))
```

$$\mathbb{C},\ \mathbb{C},\ [0, \infty)$$

```
getprop(3), getprop(sqrt(2) + 1)
```

$$\{3\},\ \left\{\sqrt{2}+1\right\}$$

Assuming some property deletes all previous assumptions using the same identifiers:

```
assume(sin(x) > y):
getprop(y)
```

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

```
assume(abs(x) > 1):
getprop(y)
```

$$\mathbb{C}$$

To make an assumption in addition to existing assumptions, use `assumeAlso` instead of `assume`, and `assumingAlso` instead of `assuming`:

```
assume(sin(x) > y):
getprop(y)
```

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

```
assumeAlso(abs(x) > 1):
getprop(y)
```

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

We now illustrate some of the various types of properties with a short example. The equation $(x^a)^b = x^{ab}$ is not generally valid, as the example $x = -1$, $a = 2$, and $b = 1/2$ shows. However, it is valid if b is an integer:

```
assume(b in Z_): (x^a)^b
```

$$x^{a\,b}$$

```
unassume(b): (x^a)^b
```

$$(x^a)^b$$

The function `linalg::isPosDef` checks whether a matrix is positive definite. For a matrix with symbolic entries, it may not be possible to decide this correctly:

```
A := matrix([[1/a, 1], [1, 1/a]])
```

$$\begin{pmatrix} \frac{1}{a} & 1 \\ 1 & \frac{1}{a} \end{pmatrix}$$

```
linalg::isPosDef(A)
  Error: cannot check whether matrix component is positive
  [linalg::factorCholesky]
```

With the additional assumption that the parameter `a` be positive and less than 1, MuPAD can decide that the matrix is positive definite:

```
assume(a, Type::Interval(0, 1))
```

```
linalg::isPosDef(A)
```

TRUE

Properties of this type can be specified in the following more intuitive way as well:

```
assume(0 < a < 1)
```

The function `simplify` reacts to properties:

```
assume(k, Type::Residue(3,4))
```

```
sin(k*PI/2)
```

$$\sin\left(\frac{\pi\,k}{2}\right)$$

```
simplify(%)
```

$$-1$$

The property above can also be specified in any of the following equivalent forms:

```
assume(k in 4*Z_ + 3)
```

```
assume((k - 3)/4 in Z_)
```

The functions `Re`, `Im`, `sign`, and `abs` take properties into account:

```
assume(x > 1):
Re(x*(x - 1)), sign(x*(x - 1)), abs(x*(x - 1))
```

$$x\,(x-1),\ 1,\ x\,(x-1)$$

Since only a limited number of mathematical properties and derivation rules are implemented in MuPAD, the system performs some simplifications when evaluating the properties of a nontrivial expression. Thus the answer of `getprop` or `is` is sometimes not "as close as possible." For example, if x is a real number, then $x^2 - x \geq -1/4$, but `getprop` yields the following less accurate answer:

```
assume(x, Type::Real): getprop(x^2 - x)
```

$$\mathbb{R}$$

Exercise 9.5: Use MuPAD to show:

$$\lim_{x\to\infty} \mathrm{Ei}(ax) = \begin{cases} \infty & \text{for } a > 0\,, \\ 0 & \text{for } a < 0\,. \end{cases}$$

Hint: Use the function `assume` to distinguish the cases. The exponential integral $\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{\mathrm{e}^t}{t}\,\mathrm{d}t$ is available as `Ei`.

10

Chance and Probability

You can use MuPAD®'s random number generators `random`, `frandom` and `stats::xxxRandom` to perform many experiments.

The call `random()` generates a random nonnegative 12 digit integer. You obtain a sequence of 4 such random numbers as follows:

```
random(), random(), random(), random()
```

$$427419669081,\ 321110693270,\ 343633073697,\ 474256143563$$

If you want to generate random integers in a different range, you can construct a random number generator `generator:=random(m..n)`. You call this generator without arguments,[1] and it returns integers between m and n. The call `random(n)` is equivalent to `random(0..n-1)`. Thus you can simulate 15 rolls of a die as follows:

```
die := random(1..6):
```

```
dieExperiment := [die() $ i = 1..15]
```

$$[5, 3, 6, 3, 2, 2, 2, 4, 4, 3, 3, 2, 1, 4, 4]$$

We stress that you must specify a loop variable when using the sequence generator `$`, since otherwise `die()` is called only once and a sequence of copies of this value is generated:

```
die() $ 15
```

$$6,\ 6,\ 6,\ 6,\ 6,\ 6,\ 6,\ 6,\ 6,\ 6,\ 6,\ 6,\ 6,\ 6,\ 6$$

[1] In fact, you can call `generator` with arbitrary arguments, which are ignored when generating random numbers.

Here is a simulation of 8 coin tosses:

```
coin := random(2):
```

```
coinTosses := [coin() $ i = 1..8]
```

$$[0, 0, 0, 1, 1, 1, 0, 0]$$

```
subs(coinTosses, [0 = head, 1 = tail])
```

$$[\text{head}, \text{head}, \text{head}, \text{tail}, \text{tail}, \text{tail}, \text{head}, \text{head}]$$

The following example generates uniformly distributed floating-point numbers from the interval $[0, 1]$. It uses the random generator `frandom`:

```
randomNumbers := [frandom() $ i = 1..10]
  [0.2703581656, 0.8310371787, 0.153156516,

     0.9948127808, 0.2662729021, 0.1801642277,

     0.452083055, 0.6787819563, 0.3549849261,

     0.6818588132]
```

The library `stats` comprises functions for statistical analysis. You obtain information by entering `info(stats)` or `?stats`. For example, the function `stats::mean` computes the mean value $X = \frac{1}{n} \sum_{i=1}^{n} x_i$ of a list of numbers $[x_1, \ldots, x_n]$:

```
stats::mean(dieExperiment),
stats::mean(coinTosses),
stats::mean(randomNumbers)
```

$$\frac{16}{5}, \frac{3}{8}, 0.4863510522$$

The function `stats::variance` returns the variance

$$Var = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - X)^2 :$$

```
stats::variance(dieExperiment),
stats::variance(coinTosses),
stats::variance(randomNumbers)
```

$$\frac{61}{35},\ \frac{15}{56},\ 0.08565360412$$

You can compute the standard deviation $\sqrt{Var}$ with `stats::stdev`:

```
stats::stdev(dieExperiment),
stats::stdev(coinTosses),
stats::stdev(randomNumbers)
```

$$\frac{\sqrt{35}\,\sqrt{61}}{35},\ \frac{\sqrt{14}\,\sqrt{15}}{28},\ 0.29266637$$

If you specify the option `Population`, the system returns $\sqrt{\frac{n-1}{n}\,Var}$ instead:

```
stats::stdev(dieExperiment, Population),
stats::stdev(coinTosses, Population),
stats::stdev(randomNumbers, Population)
```

$$\frac{\sqrt{3}\,\sqrt{122}}{15},\ \frac{\sqrt{15}}{8},\ 0.2776476971$$

The data structure `Dom::Multiset` (see `?Dom::Multiset`) provides a simple means of determining frequencies in sequences. The call `Dom::Multiset(a,b, ...)` returns a multiset, which is printed as a set of lists. The first entry of each such list is one of the arguments; the second entry counts the number of occurrences in the argument sequence:

```
Dom::Multiset(a, b, a, c, b, b, a, a, c, d, e, d)
```

$$\{[a,4],[b,3],[c,2],[d,2],[e,1]\}$$

If you simulate 1000 rolls of a die, you might obtain the following frequencies:

```
rolls := die() $ i = 1..1000:
```

```
Dom::Multiset(rolls)
```

$$\{[1,158],[2,152],[3,164],[4,188],[5,176],[6,162]\}$$

In this case, you would have rolled 158 times a 1, 152 times a 2 etc.

An example from number theory is the distribution of the greatest common divisors (gcd) of random pairs of integers. We use the function `zip` (Section 4.6) to combine two random lists via the function `igcd`, which computes the gcd of integers:

```
list1 := [random() $ i=1..1000]:
```

```
list2 := [random() $ i=1..1000]:
```

```
gcdlist := zip(list1, list2, igcd)
  [1, 7, 1, 1, 1, 5, 1, 3, 1, 1, 1, 3, 6, 1, 3, 5, ...]
```

Now, we use `Dom::Multiset` to count the frequencies of the individual gcds:

```
frequencies := Dom::Multiset(op(gcdlist))
  {[11, 5], [13, 3], [14, 2], ..., [377, 1], [1, 596]}
```

The result becomes much more readable if we sort it by the first entry of the sublists. We employ the function `sort` which takes a function representing a sorting order as second argument. This function decides which of two elements x, y shall precede the other. We refer to the corresponding help page: `?sort`. In this case, x, y are lists with two entries, and we want x to appear before y if we have $x[1] < y[1]$ (i.e., we sort numerically with respect to the first entries):

```
sortingOrder := (x, y) -> (x[1] < y[1]):
```

```
sort([op(frequencies)], sortingOrder)
  [[1, 596], [2, 142], [3, 84], ..., [212, 1], [377, 1]]
```

In this experiment, 596 out of the 1000 chosen random pairs have a gcd of 1 and hence are coprime. Thus, we found 59.6% as an approximation of the probability that two randomly chosen integers are coprime. The theoretical value of this probability is $6/\pi^2 \approx 0.6079.. \widehat{=} 60.79\%$.

The `stats` library provides many stochastic distribution functions. Each such distribution `xxx`, say, consists of four functions: a cumulative distribution function `xxxCDF`, a probability density function `xxxPDF` (or a discrete probability function `xxxPF`, respectively), a quantile function `xxxQuantile`, and a random number generator `xxxRandom`. For example, the following command creates a list of random numbers distributed according to the standard normal distribution with mean 0 and variance 1:

```
randomGenerator := stats::normalRandom(0, 1):
data := [randomGenerator() $ i = 1..1000]:
```

The `stats` library includes an implementation of the classical χ^2-test. We use it here to test whether the random data generated above do indeed satisfy a normal distribution. Pretending that we do not know the mean and the variance of the data, we compute statistical estimates of these parameters:

```
m := stats::mean(data)
```

$$-0.03440235553$$

```
V := stats::variance(data)
```

$$0.9962289589$$

The χ^2-test requires to specify a "cell partitioning" of the real line to compare the observed frequencies of the data falling into the cells with the expected frequencies given a hypothesized distribution of the data. The function `stats::equiprobableCells` is a convenient utility function to compute a cell partitioning consisting of cells that are equiprobable with respect to a given distribution. The following call partitions the real line into 32 cells which are equiprobable with respect to the normal distribution with the empirical mean and variance computed above:

```
cells := stats::equiprobableCells(32,
                   stats::normalQuantile(m, V))
  [[-infinity, -1.89096853], [-1.89096853, -1.553118836],

      ..., [1.939230531, infinity]]
```

The χ^2-goodness-of-fit test is implemented by `stats::csGOFT`. We use it to test how good the random data fit a normal distribution with mean and variance given by the empirical values `m` and `V`:

```
stats::csGOFT(data, cells,
              CDF = stats::normalCDF(m, V))
  [PValue = 0.3862347505, StatValue = 32.64,

     MinimalExpectedCellFrequency = 31.25]
```

The first value returned by `stats::csGOFT` is the significance level attained by the data. Since this value is not small, the given data pass the test well.

Finally, we dote the data by adding 35 zeroes:

```
data := append(data, 0 $ 35):
```

We check whether the new data still may be regarded as a normally distributed sample:

```
m := stats::mean(data): V := stats::variance(data):
cells := stats::equiprobableCells(32,
                    stats::normalQuantile(m, V)):
stats::csGOFT(data, cells,
              CDF = stats::normalCDF(m, V))
```

```
  [PValue = 0.000004764116336, StatValue = 78.87826087,

     MinimalExpectedCellFrequency = 32.34375]
```

Now, the attained significance level $0.0010\ldots$ indicates that the hypothesis of a normal distribution of the data must be rejected at significance levels as low as 0.001.

See the help page of `stats::csGOFT` for further details.

Exercise 10.1: Three dice are thrown simultaneously. For each value between 3 and 18, the following table contains the expected frequencies of the total dice score when rolling the dice 216 times:

score

3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	3	6	10	15	21	25	27	27	25	21	15	10	6	3	1

frequency

Simulate 216 rolls and compare the frequencies that you observe with those from the table.

Exercise 10.2: The Monte-Carlo method for approximating the area of a region $A \subset \mathbb{R}^2$ works as follows. First, we choose a (preferably small) rectangle Q enclosing A. Then we randomly choose n points in Q. If m of these points lie in A, the following estimate holds for sufficiently large n:

$$\text{area of } A \approx \frac{m}{n} \times \text{area of } Q.$$

Let `r()` be a call to a generator `r` of uniformly distributed random numbers in the interval $[0, 1]$. We can use `r` to generate uniformly distributed random vectors in the rectangle $Q = [0, a] \times [0, b]$ via `[a*r(),b*r()]`.

a) Consider the right upper quadrant of the unit circle around the origin. Use Monte-Carlo simulation with $Q = [0, 1] \times [0, 1]$ to approximate its area. This way, one gets stochastic approximations of π.

b) Let $f : x \mapsto x \sin(x) + \cos(x) \exp(x)$. Determine an approximation for $\int_0^1 f(x)\, dx$. For that purpose, find an upper bound M for f on the interval $[0, 1]$ and apply the simulation to $Q = [0, 1] \times [0, M]$. Compare your result to the exact integral.

11

Graphics

Introduction

Since exploring mathematical objects is probably one of the most important uses of computer algebra systems, graphics are a rather central part of modern CAS. The MuPAD® system allows to easily create two- and three-dimensional graphics and animations with high quality output and advanced interactive manipulation capabilities.

To aid the user in fine-tuning a plot, an object browser is provided. It allows to inspect and analyze the tree structure of graphical scenes interactively, giving easy access to all parts of the plot by a mouse-click. After selecting an object, all plot attributes such as color, line width, annotations etc. associated with the object become visible. One may select the attributes with the mouse and change their values interactively (see page 11-50). Thus, almost any detail of a plot generated by a MuPAD call can be changed interactively making it possible to fine-tune both the contents as well as the looks by visual inspection.

The features of the renderer are fully supported by the `plot` library. It provides a list of graphical primitives such as points, lines, polygons etc. that one may use to build up highly complex graphical scenes. The primitives also include graphical objects that are not at all "primitive," but come along with advanced algorithms and much built-in intelligence. Examples are function graphs in 2D and 3D (with automatic clipping for singular functions), implicit plots in 2D and 3D, vector field plots, graphical solutions of ordinary differential equations in 2D and 3D etc.

This chapter gives an introduction to the new graphics system. It explains the basic concepts and ideas and provides many examples.

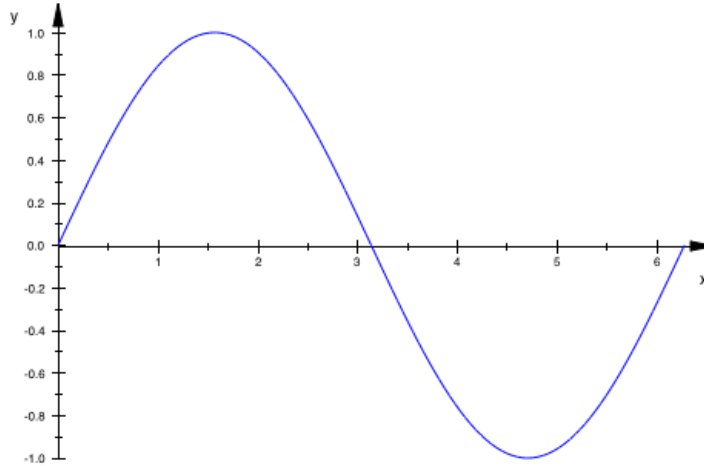
Easy Plotting: Graphs of Functions

Probably the most important graphical task in a mathematical context is to visualize function graphs, i.e., to plot functions. The `plot` command allows to create 2D plots of functions with one argument (such as $f(x) = \sin(x)$) or 3D plots of functions with two arguments (such as $f(x, y) = \sin(x^2 + y^2)$). The calling syntax is simple: just pass the expression that defines the function and, optionally, a range for the independent variable(s). To distinguish 3D plots from 2D animations, add `#3D` to the call.

2D Function Graphs

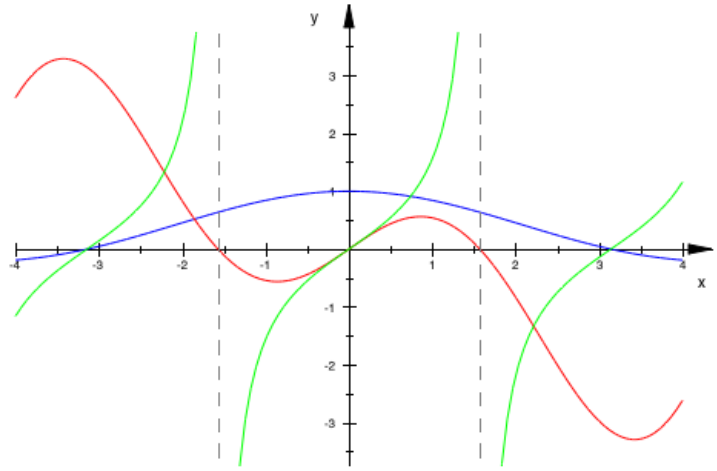
We consider 2D examples, i.e., plots of univariate functions $f(x)$. Here is one period of the sine function:

```
plot(sin(x), x = 0..2*PI)
```



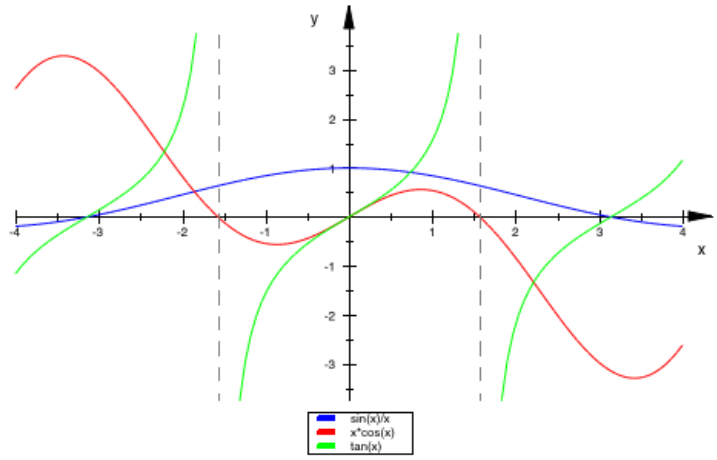
If several functions are to be plotted in the same graphical scene, just pass a sequence of function expressions. All functions are plotted over the specified common range:

```
plot(sin(x)/x, x*cos(x), tan(x), x = -4..4)
```



To get a legend explaining the mapping from color to function, add the option `LegendVisible`:

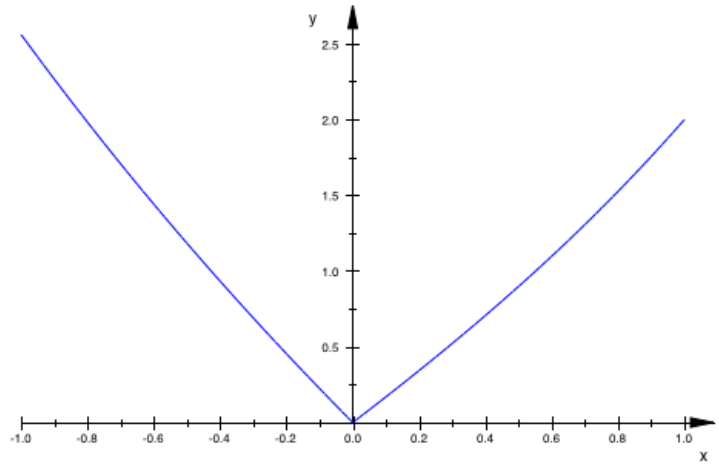
```
plot(sin(x)/x, x*cos(x), tan(x), x = -4..4, LegendVisible)
```



Functions that do not allow a simple symbolic representation by an expression can also be defined by a procedure that produces a numerical value $f(x)$ when called with a numerical value x from the plot range. In the following example we consider the largest eigenvalue of a symmetric 3×3 matrix that contains a parameter x. We plot this eigenvalue as a function of x:

```
f := x -> max(numeric::eigenvalues(
                  matrix([[-x, x, -x  ],
                          [ x, x,  x  ],
                          [-x, x,  x^2]]))):
```

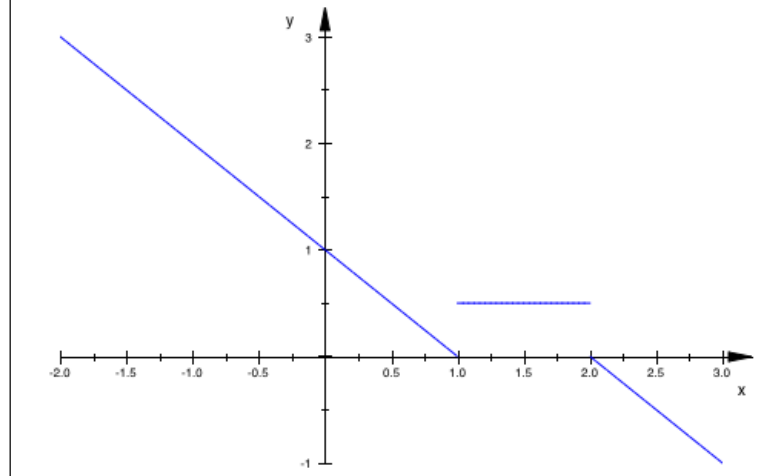
```
plot(f, x = -1..1)
```



The name x used in the specification of the plotting range provides the name that labels the horizontal axis.

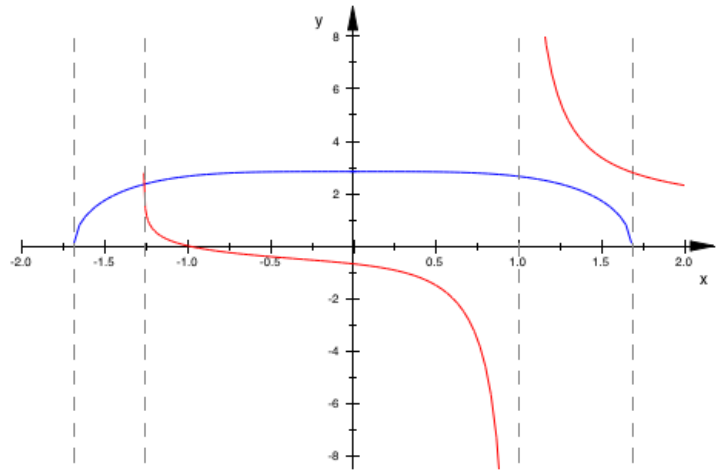
Functions can also be defined by `piecewise` objects:

```
plot(piecewise([x < 1, 1 - x],
               [1 < x and x < 2, 1/2],
               [x > 2, 2 - x]),
     x = -2..3)
```



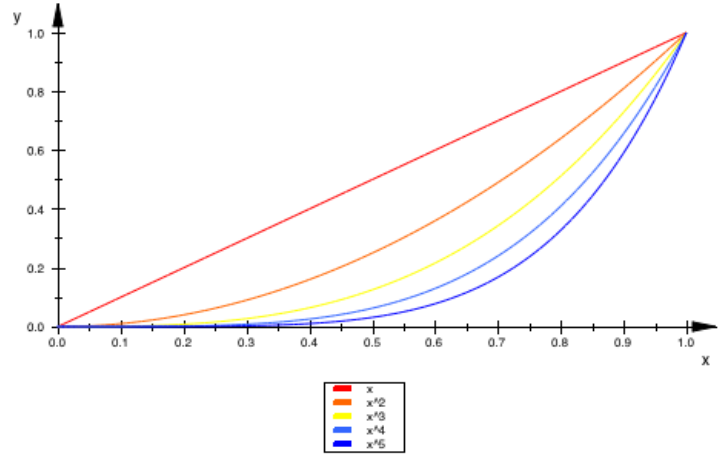
Note that there are gaps in the definition of the function above: no function value is specified for $x = 1$ and $x = 2$. This does not cause any problem, because `plot` simply ignores all points that do not produce real numerical values. Thus, in the following example, the plot is automatically restricted to the regions where the functions produce real values:

```
plot(sqrt(8 - x^4), ln(x^3 + 2)/(x - 1), x = -2 ..2)
```



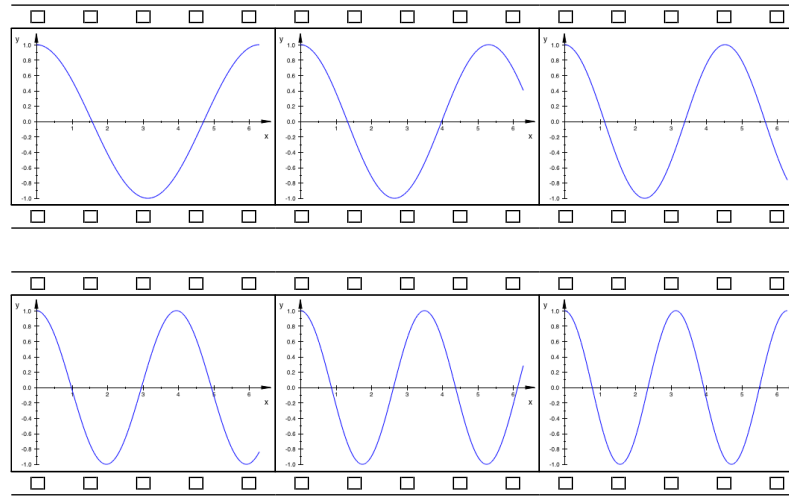
When several functions are plotted in the same scene, they are drawn in different colors that are chosen automatically. With the `Colors` attribute one may specify a list of RGB colors that `plot` shall use:

```
plot(x, x^2, x^3, x^4, x^5, x = 0..1,
     Colors = [RGB::Red, RGB::Orange, RGB::Yellow,
               RGB::BlueLight, RGB::Blue],
     LegendVisible)
```

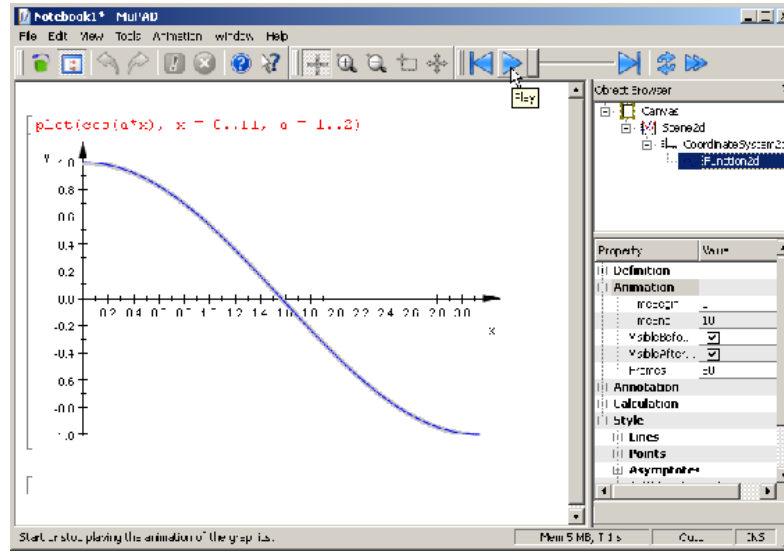


Animated 2D plots of functions are created by passing function expressions depending on a variable (x, say) and an animation parameter (a, say) and specifying a range both for x and a. In this text, animations are represented by a "film strip:"

```
plot(cos(a*x), x = 0..2*PI, a = 1..2)
```

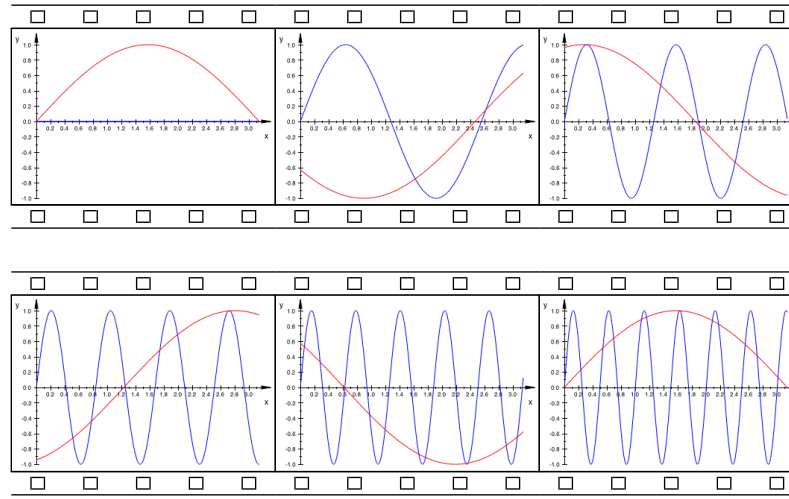


Once the plot is created, the first frame of the picture appears as a static plot. After double-clicking on the picture, the graphics tool allows to start the animation by pushing the “Start” button:



An animation consists of separate frames, i.e., individual still images. The default number of frames is 50. If a different value is desired, just pass the attribute `Frames=n`, where n is the number of frames that shall be created:

```
plot(sin(a*x), sin(x - a), x = 0..PI, a = 0..4*PI,
     Frames = 200, Colors = [RGB::Blue, RGB::Red])
```



Apart from the color specification or the `Frames` number, there is a large number of further attributes that may be passed to `plot`. Each attribute is passed as an equation `AttributeName=AttributeValue`. Here, we only present some selected attributes. Section "Attributes for `plotfunc2d` and `plotfunc3d`" of the online `plot` documentation provides further tables with more attributes.

attribute name	possible values	meaning	default
`Height`	`8*unit::cm`	physical height of the picture	`80*unit::mm`
`Width`	`12*unit::cm`	physical width of the picture	`120*unit::mm`
`Footer`	string	footer text	`""` (no footer)
`Header`	string	header text	`""` (no header)
`Title`	string	title text	`""` (no title)
`TitlePosition`	[real value, real value]	coordinates of the lower left corner of the title	

Table 11.1: Some important attributes for plotting 2D functions

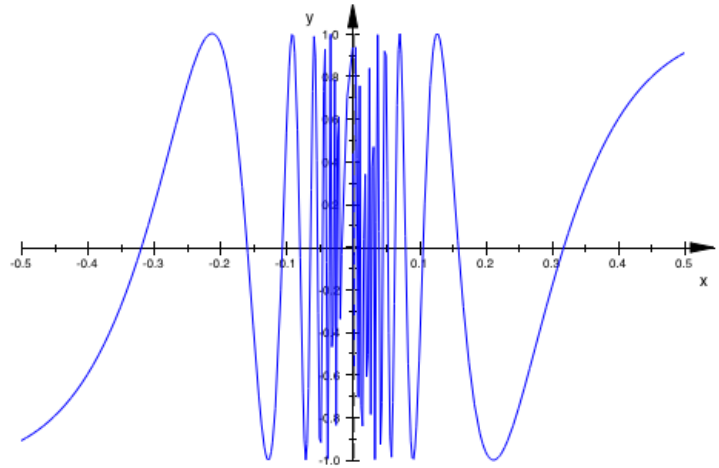
attribute name	possible values	meaning	default
`GridVisible`	`TRUE`, `FALSE`	visibility of "major" grid lines	`FALSE`
`SubgridVisible`	`TRUE`, `FALSE`	visibility of "minor" grid lines	`FALSE`
`AdaptiveMesh`	integer ≥ 0	depth of the adaptive mesh	`2`
`Axes`	`None`, `Automatic`, `Boxed`, `Frame`, `Origin`	axes type	`Automatic`
`AxesVisible`	`TRUE`, `FALSE`	visibility of all axes	`TRUE`
`AxesTitles`	`[string, string]`	titles of the axes	`["x","y"]`
`CoordinateType`	`LinLin`, `LinLog`, `LogLin`, `LogLog`	linear-linear, linear-logarithmic, log-linear, log-log	`LinLin`
`Colors`	list of RGB values	line colors	first 10 entries of `RGB::ColorList`
`Frames`	integer ≥ 0	number of frames of an animation	`50`
`LegendVisible`	`TRUE`, `FALSE`	legend on/off	`TRUE` if plotting more than one function
`LineColorType`	`Dichromatic`, `Flat`, `Functional`, `Monochrome`, `Rainbow`	color scheme	`Flat`
`Mesh`	integer ≥ 2	number of sample points	`121`
`Scaling`	`Automatic`, `Constrained`, `Unconstrained`	scaling mode	`Unconstrained`
`TicksNumber`	`None`, `Low`, `Normal`, `High`	number of labeled ticks	`Normal`
`ViewingBoxYRange`	`ymin..ymax`	restricted viewing range in y direction	`Automatic..Automatic`

Table 11.1: Some important attributes for plotting 2D functions

We now present some examples for the use of these attributes.

First, we will give an example where the number of evaluation points needs adjustment. The following plot example features the notorious function $\sin(1/x)$ that oscillates wildly near the origin:

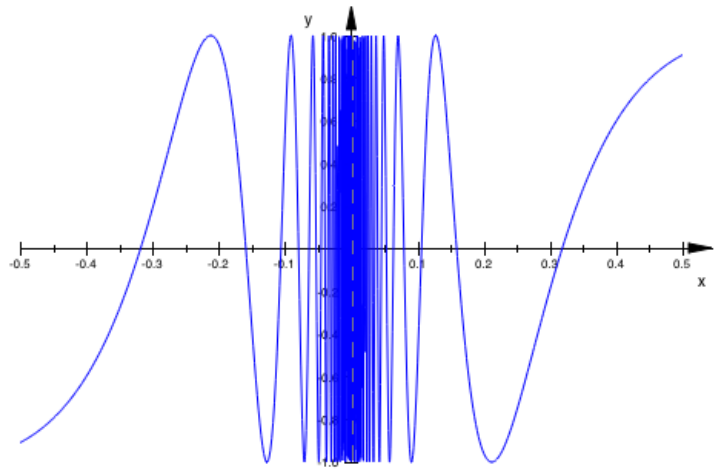
```
plot(sin(1/x), x = -0.5..0.5)
```



Clearly, the default of 121 sample points used by `plot` does not suffice to create a sufficiently resolved plot.

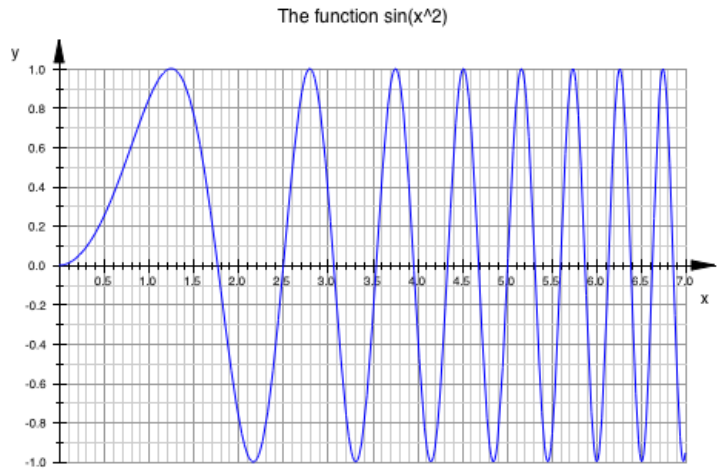
We increase the number of numerical mesh points via the `Mesh` attribute. Additionally, we increase the resolution depth of the adaptive plotting mechanism from its default value `AdaptiveMesh=2` to `AdaptiveMesh=4`:

```
plot(sin(1/x), x = -0.5..0.5, Mesh = 500,
     AdaptiveMesh = 4)
```



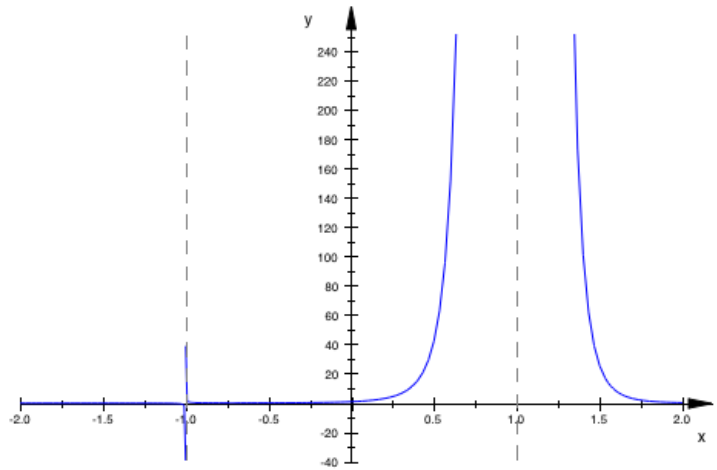
The following call specifies a header via `Header="The function sin(x^2)"`. The distance between labeled ticks is set to 0.5 along the x axis and to 0.2 along the y axis via `XTicksDistance=0.5` and `YTicksDistance=0.2`, respectively. Four additional unlabeled ticks between each pair of labeled ticks are set in the x direction via `XTicksBetween=4`. One additional unlabeled tick between each pair of labeled ticks in the y direction is requested via `YTicksBetween=1`. Grid lines attached to the ticks are "switched on" by `GridVisible=TRUE` and `SubgridVisible=TRUE`:

```
plot(sin(x^2), x = 0..7,
     Header = "The function sin(x^2)",
     XTicksDistance = 0.5, YTicksDistance = 0.2,
     XTicksBetween = 4, YTicksBetween = 1,
     GridVisible = TRUE, SubgridVisible = TRUE)
```



When singularities are found in the function, an automatic clipping is called trying to restrict the vertical viewing range in some way to obtain a "reasonably" scaled picture. This is a heuristic approach that sometimes needs a helping adaption "by hand." In the following example, the automatically chosen range between $y \approx -40$ and $y \approx 260$ in vertical direction is suitable to represent the 6th order pole at $x = 1$, but it does not provide a good resolution of the first order pole at $x = -1$:

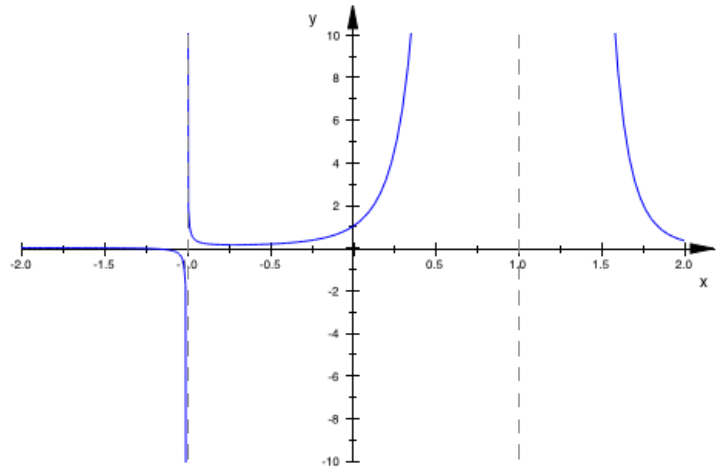
```
plot(1/(x + 1)/(x - 1)^6, x = -2..2)
```



There is no good viewing range that is adequate for both poles because they are of different order. However, some compromise can be found.

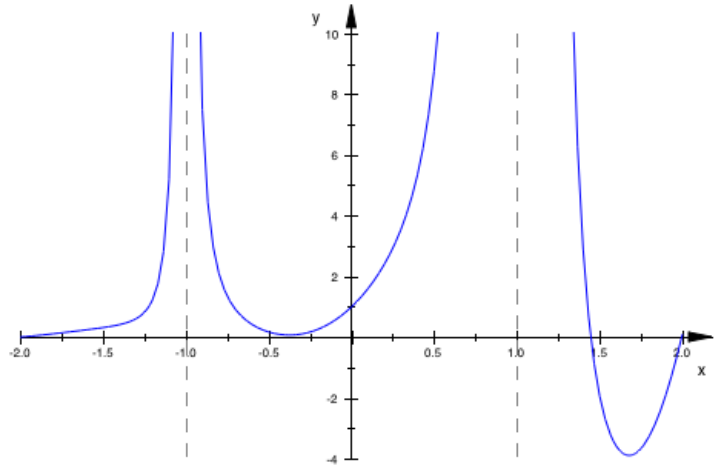
We override the automatic viewing range suggested by `plot` and request a specific viewing range in vertical direction via `ViewingBoxYRange`:

```
plot(1/(x + 1)/(x - 1)^6, x = -2..2,
     ViewingBoxYRange = -10..10)
```



The values of the following function have a lower bound, but no upper bound. We use the attribute `ViewingBoxYRange=Automatic..10` to let `plot` find a lower bound for the viewing box by itself whilst requesting a specific value of 10 for the upper bound:

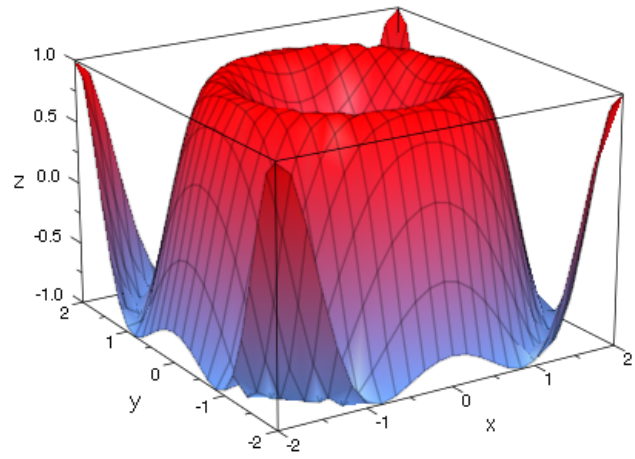
```
plot(exp(x)*sin(PI*x) + 1/(x + 1)^2/(x - 1)^4,
     x = -2..2, ViewingBoxYRange = Automatic..10)
```



3D Function Graphs

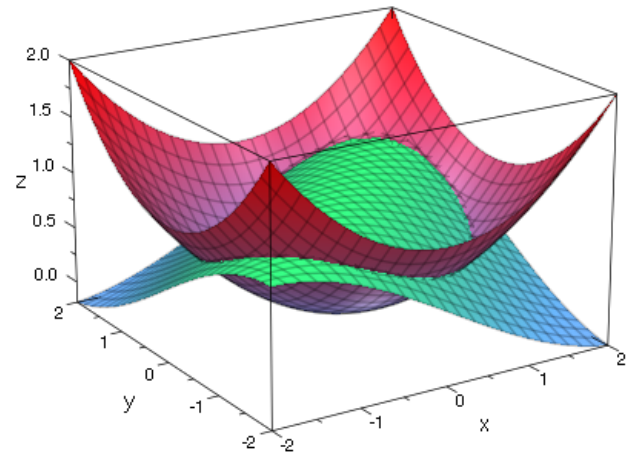
We consider 3D examples, i.e., plots of bi-variate functions $f(x, y)$. As noted above, to distinguish static 3D and animated 2D plots, the former need the option `#3D`. Here is a plot of the function $\sin(x^2 + y^2)$:

```
plot(sin(x^2 + y^2), x = -2..2, y = -2..2, #3D)
```



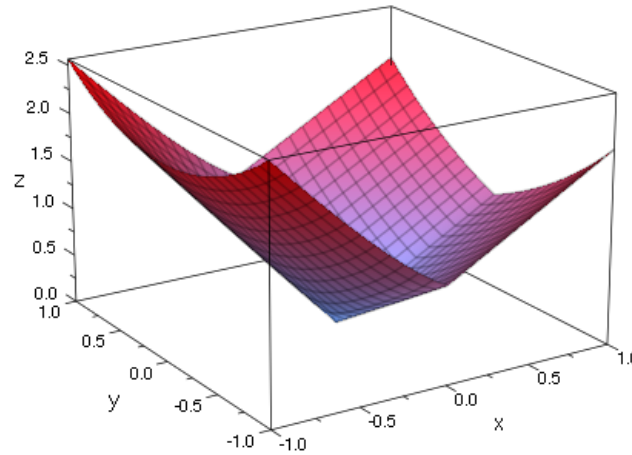
If several functions are to be plotted in the same graphical scene, just pass a sequence of function expressions; all functions are plotted over the specified common range:

```
plot((x^2 + y^2)/4, sin(x - y)/(x - y),
     x = -2..2, y = -2..2, #3D)
```



Functions that do not allow a simple symbolic representation by an expression can also be defined by a procedure that produces a numerical value $f(x, y)$ when called with numerical values x, y from the plot range. In the following example we consider the largest eigenvalue of a symmetric 3×3 matrix that contains two parameters x, y. We plot this eigenvalue as a function of x and y:

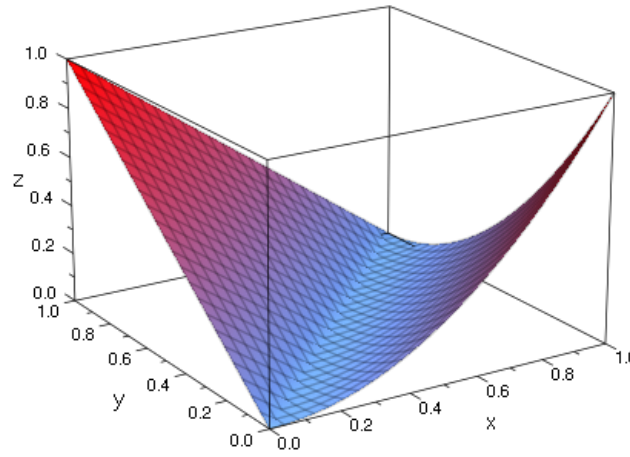
```
f := (x, y) -> max(numeric::eigenvalues(
          matrix([[-y, x, -x  ],
                  [ x, y,  x  ],
                  [-x, x,  y^2]]))):
plot(#3D, f, x = -1..1, y = -1..1)
```



The names x, y used in the specification of the plotting range provide the labels of the corresponding axes.

Functions can also be defined by `piecewise` objects:

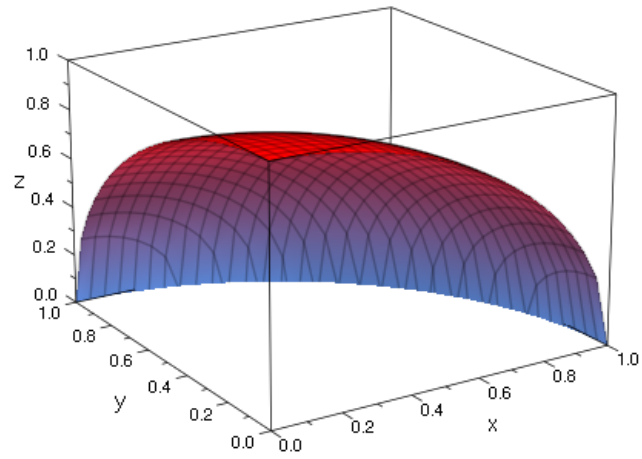
```
plot(piecewise([x < y, y-x], [x > y, (y-x)^2]),
     x = 0..1, y = 0..1, #3D)
```



Note that there are gaps in the definition of the function above: no function value is specified for $x = y$. This does not cause any problem, because `plot` simply ignores points that do not produce real numerical values if it finds suitable values in the neighborhood. Thus, missing points or singularities do not raise an error.

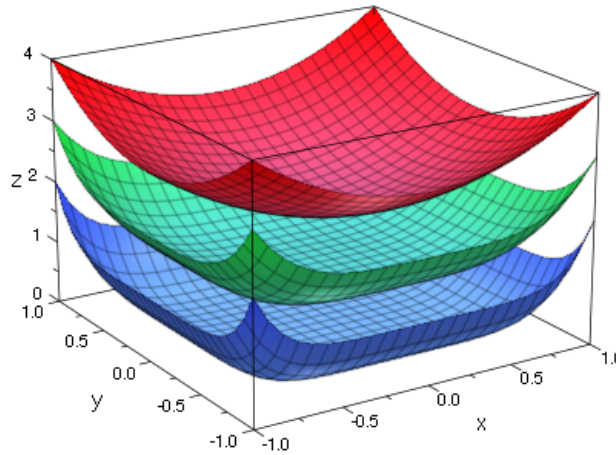
If the function is not real-valued in regions of non-zero measure, gaps will be visible in the output. The following function is real-valued only in the disc $x^2 + y^2 \leq 1$:

```
plot(sqrt(1 - x^2 - y^2), x = 0..1, y = 0..1, #3D)
```



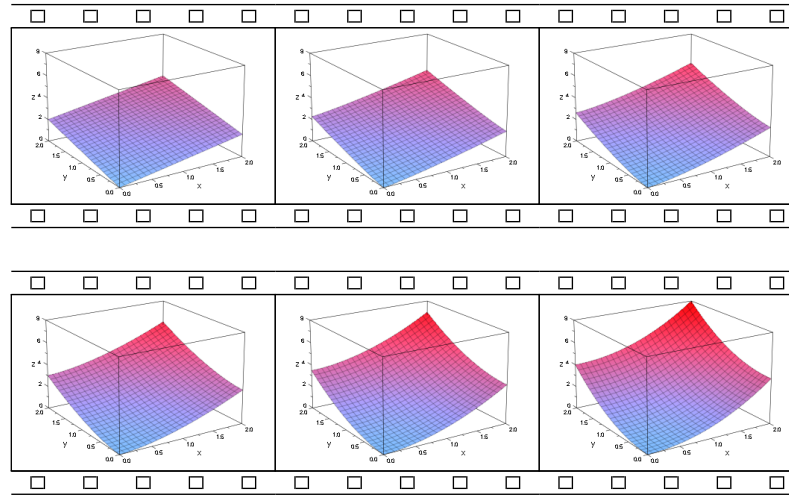
When several functions are plotted in the same scene, they are drawn in different colors that are chosen automatically. With the `Colors` attribute one may specify a list of RGB colors that `plot` shall use:

```
plot(2 + x^2 + y^2, 1 + x^4 + y^4, x^6 + y^6,
     x = -1..1, y = -1..1, #3D,
     Colors = [RGB::Red, RGB::Green, RGB::Blue])
```



Animated 3D plots of functions are created by passing function expressions depending on two variables (x and y, say) and an animation parameter (a, say) and specifying a range for x, y, and a.

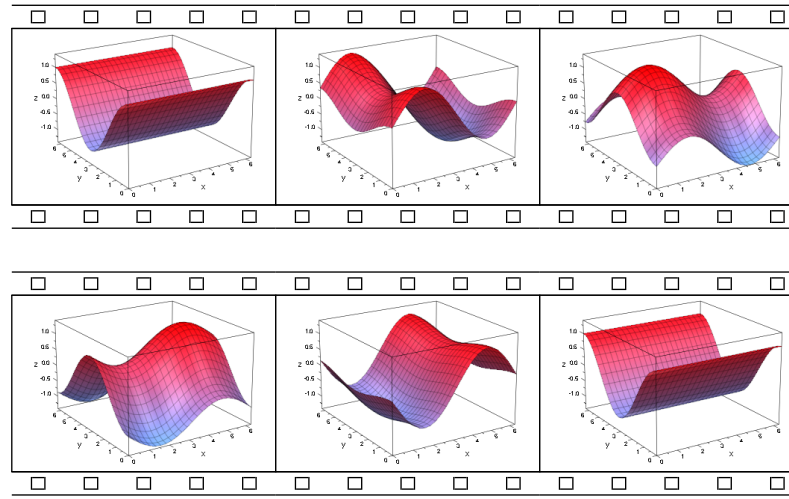
```
plot(x^a + y^a, x = 0..2, y = 0..2, a = 1..2, #3D)
```



Once the plot is created, the first frame of the picture appears as a static plot. After double-clicking on the picture, the graphics tool allows to start the animation by pushing the “Start” button.

An animation consists of separate frames. The default number of frames is 50. If a different value is desired, just pass the attribute `Frames = n`, where n is the number of frames that shall be created:

```
plot(sin(a)*sin(x) + cos(a)*cos(y), x = 0..2*PI,
     y = 0..2*PI, a = 0..2*PI, #3D, Frames = 32)
```



Apart from the color specification or the `Frames` number, there is a large number of further attributes that may be passed to `plot`. Each attribute is passed as an equation to `plot`. Here, we only present some selected attributes. Section "Attributes for `plotfunc2d` and `plotfunc3d`" of the online `plot` documentation provides further tables with more attributes.

attribute name	possible values	meaning	default
`Height`	`8*unit::cm`	physical height of the picture	`80*unit::mm`
`Width`	`12*unit::cm`	physical width of the picture	`120*unit::mm`
`Footer`	string	footer text	`""` (no footer)
`Header`	string	header text	`""` (no header)
`Title`	string	title text	`""` (no title)
`TitlePosition`	[real value, real value]	coordinates of the lower left corner of the title	
`GridVisible`	`TRUE`, `FALSE`	visibility of "major" grid lines	`FALSE`

Table 11.2: Some important attributes for plotting 3D functions

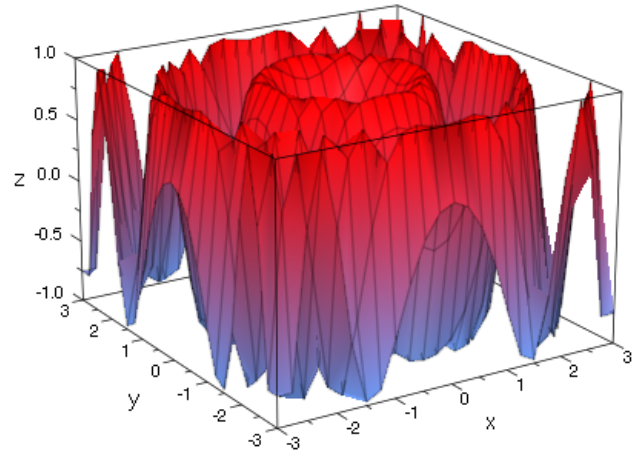
attribute name	possible values	meaning	default
`SubgridVisible`	`TRUE`, `FALSE`	visibility of "minor" grid lines	`FALSE`
`AdaptiveMesh`	integer ≥ 0	depth of the adaptive mesh	`0`
`Axes`	`Automatic`, `Boxed`, `Frame`, `Origin`	axes type	`Boxed`
`AxesVisible`	`TRUE`, `FALSE`	visibility of all axes	`TRUE`
`AxesTitles`	`[string, string, string]`	titles of the axes	`["x","y","z"]`
`Colors`	list of RGB colors	fill colors	
`Frames`	integer ≥ 0	number of frames of the animation	`50`
`LegendVisible`	`TRUE`, `FALSE`	legend on/off	`TRUE` if plotting more than one function
`FillColorType`	`Dichromatic`, `Flat`, `Functional`, `Monochrome`, `Rainbow`	color scheme	`Dichromatic`
`Mesh`	[integer ≥ 2, integer ≥ 2]	number of "major" mesh points	`[25, 25]`
`Submesh`	[integer ≥ 0, integer ≥ 0]	number of "minor" mesh points	`[0, 0]`
`Scaling`	`Automatic`, `Constrained`, `Unconstrained`	scaling mode	`Unconstrained`
`TicksNumber`	`None`, `Low`, `Normal`, `High`	number of labeled ticks	`Normal`
`ViewingBoxZRange`	`zmin .. zmax`	restricted viewing range in z direction	`Automatic .. Automatic`

Table 11.2: Some important attributes for plotting 3D functions

Let us present some example uses of these attributes. As for 2D plots, we'll start with the evaluation density.

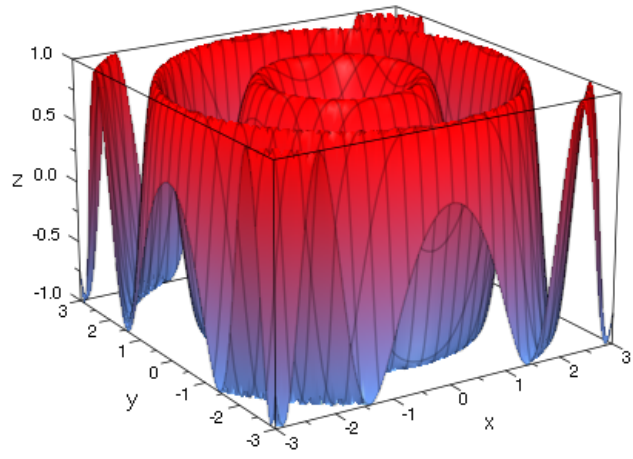
In the following example, the default mesh of 25×25 sample points used by `plot` does not suffice to create a sufficiently resolved plot:

```
plot(sin(x^2 + y^2), x = -3..3, y = -3..3, #3D)
```



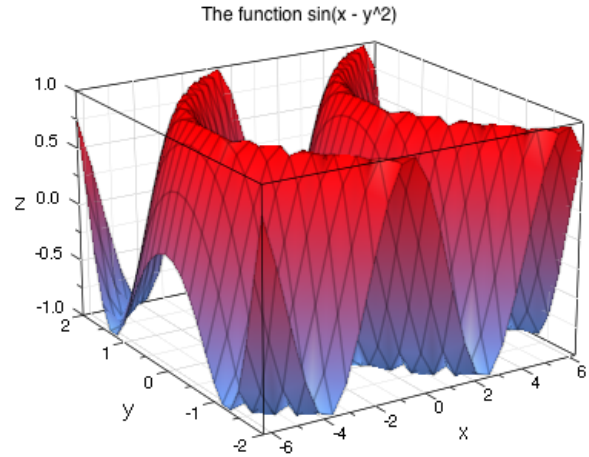
We increase the number of numerical mesh points via the `Submesh` attribute:

```
plot(sin(x^2 + y^2), x = -3..3, y = -3..3,
     Submesh = [3, 3], #3D)
```



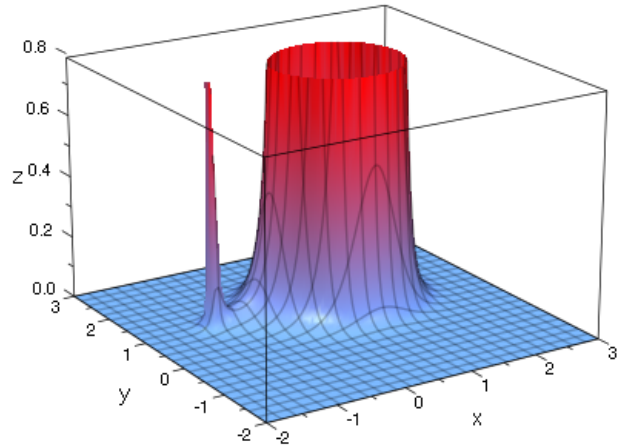
The following call specifies a header via `Header = "The function sin(x - y^2)"`. Grid lines attached to the ticks are "switched on" by `GridVisible` and `SubgridVisible`:

```
plot(sin(x - y^2), x = -2*PI..2*PI, y = -2..2, #3D,
     Header = "The function sin(x - y^2)",
     GridVisible = TRUE, SubgridVisible = TRUE)
```



When singularities are found in the function, an automatic clipping is called trying to restrict the vertical viewing range in some way to obtain a "reasonably" scaled picture. This is a heuristic approach that sometimes needs a helping adaption "by hand." In the following example, the automatically chosen range between $z \approx 0$ and $z \approx 0.8$ in vertical direction is suitable to represent the pole at $x = 1, y = 1$, but it does not provide a good resolution of the pole at $x = -1, y = 1$:

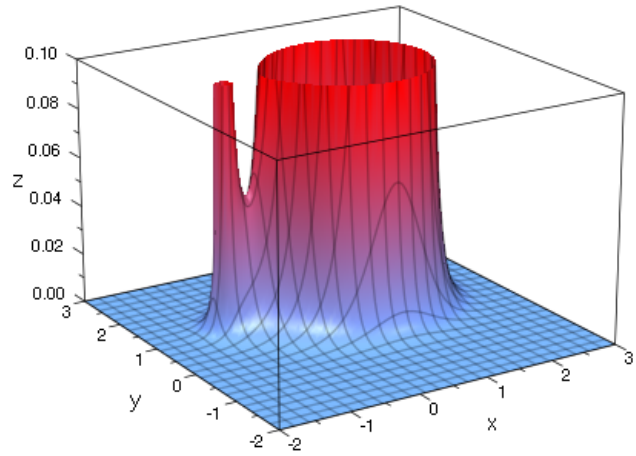
```
plot(1/((x+1)^2 + (y-1)^2)/((x-1)^2 + (y-1)^2)^5,
     x = -2..3, y = -2..3, Submesh = [3, 3], #3D)
```



There is no good viewing range that is adequate for both poles because they are of different order.

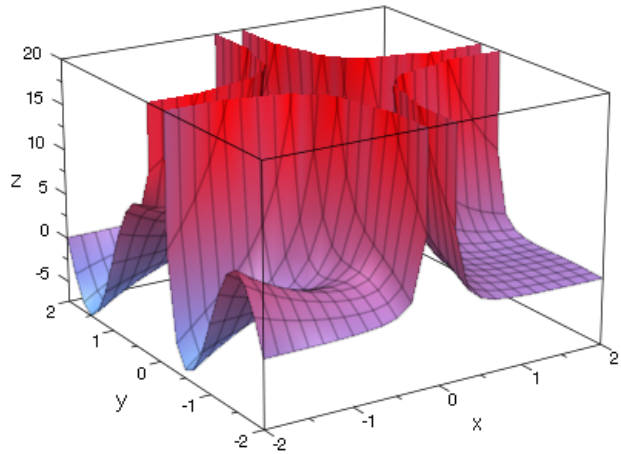
We override the automatic viewing range suggested by `plot` and request a specific viewing range in the vertical direction via `ViewingBoxZRange`:

```
plot(1/((x+1)^2 + (y-1)^2)/((x-1)^2 + (y-1)^2)^5,
     x = -2..3, y = -2..3, Submesh = [3, 3],
     ViewingBoxZRange = 0..0.1, #3D)
```



The values of the following function have a lower bound but no upper bound. We use the attribute `ViewingBoxZRange = Automatic..20` to let `plot` find a lower bound for the viewing box by itself whilst requesting a specific value of 20 for the upper bound:

```
plot(1/x^2/y^2 + exp(-x)*sin(PI*y),
     x = -2..2, y = -2..2, #3D,
     ViewingBoxZRange = Automatic..20)
```



Advanced Plotting: Principles and First Examples

In the previous section, we introduced the `plot` command for plotting functions in 2D and 3D with simple calls in easy syntax. Although there is a large number of attributes to control the graphical output and there is also a large number of additional possibilities for plotting point lists etc., there remains a serious restriction: the attributes are used for all functions simultaneously.

If attributes `attr11` etc. are to be applied to functions individually, one needs to invoke a (slightly) more elaborate calling syntax to generate the plot:

```
plot(
  plot::Function2d(f1, x1 = a1..b1, attr11, attr12, ...),
  plot::Function2d(f2, x2 = a2..b2, attr21, attr22, ...),
  ...
)
```

In this call, each call of `plot::Function2d` creates a separate object that represents the graph of the function passed as the first argument over the plotting range passed as the second argument. An arbitrary number of plot attributes can be associated with each function graph. The objects themselves are not displayed directly. The `plot` command triggers the evaluation of the functions on some suitable numerical mesh and calls the renderer to display these numerical data in the form specified by the given attributes.

In fact, `plot` called on a plain expression does precisely the same: internally, the utility function `plot::easy` generates objects of type `plot::Function2d` or `plot::Function3d`, respectively, and these are rendered.

General Principles

In general, graphical scenes are collections of "graphical primitives." There are simple primitives such as points, line segments, polygons, rectangles and boxes, circles and spheres, histogram plots, pie charts etc. An example of a more advanced primitive is `plot::VectorField2d` that represents a collection of arrows attached to a regular mesh visualizing a vector field over a rectangular region in $\mathbb{R}^2$. Yet more advanced primitives are function graphs and parametrized curves that come equipped with some internal intelligence, e.g., knowing how to evaluate

themselves numerically on adaptive meshes and how to clip themselves in case of singularities. As examples for highly advanced primitives in the `plot` library, we mention `plot::Ode2d`, `plot::Ode3d`, `plot::Streamlines2d`, and `plot::Implicit2d`, `plot::Implicit3d`. The first automatically solve systems of ordinary differential equations numerically and display the solutions graphically. The latter display the solution curves or surfaces of equations $f(x, y) = 0$ or $f(x, y, z) = 0$, respectively, by solving these equations numerically, a typical task in (real) algebraic geometry.

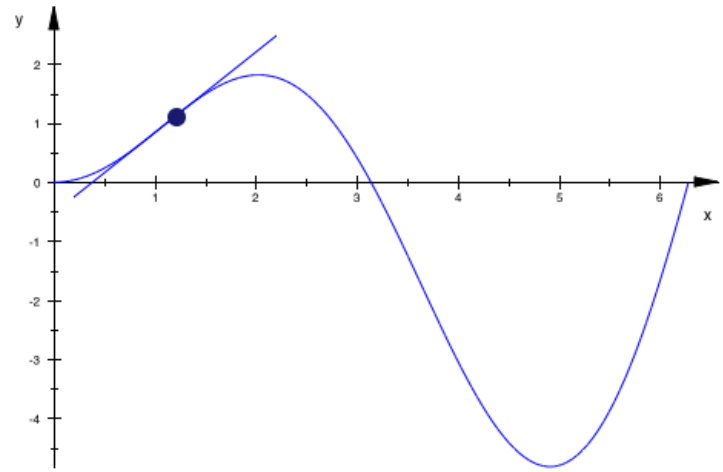
All these primitives are just "objects" representing some graphical entities. They are not rendered directly when they are created, but just serve as data structures encoding the graphical meaning, collecting attributes defining the presentation style and providing numerical routines to convert the input parameters to some numerical data that are to be sent to the renderer.

An arbitrary number of such primitives can be collected to form a graphical scene. Finally, a call to `plot` passing a sequence of all primitives in the scene invokes the renderer to draw the plot. The following example shows the graph of the function $f(x) = x \cdot \sin(x)$. At the point $(x_0, f(x_0))$, a graphical point is inserted and the tangent to the graph through this point is added:

```
f := x -> x*sin(x):
x0:= 1.2: dx := 1:
g := plot::Function2d(f(x), x = 0..2*PI):
p := plot::Point2d(x0, f(x0), PointSize = 3.0*unit::mm):
t := plot::Line2d([x0 - dx, f(x0) - f'(x0)*dx],
                  [x0 + dx, f(x0) + f'(x0)*dx]):
```

The picture is drawn by calling `plot`:

```
plot(g, t, p)
```



Each primitive accepts a variety of plot attributes that may be passed as a sequence of equations `AttributeName=AttributeValue` to the generating call. Most prominently, almost every primitive allows to set the color explicitly:

```
g := plot::Function2d(f(x), x = 0..2*PI,
                      Color = RGB::Blue):
```

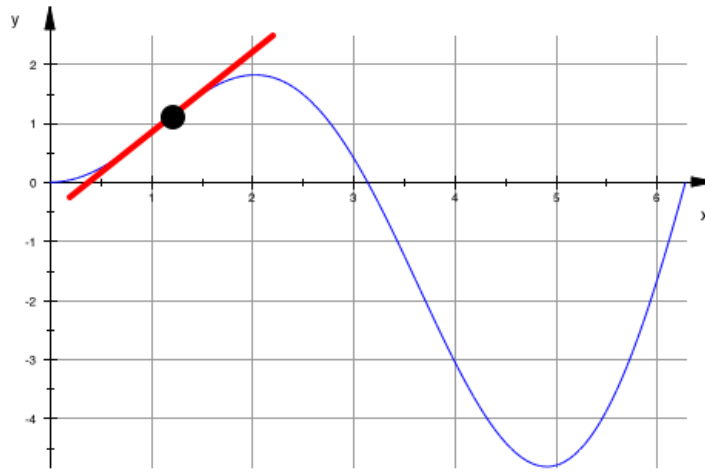
Alternatively, the generated objects allow to set attributes via slot assignments of the form `object::AttributeName:=AttributeValue` as in

```
p::Color := RGB::Black:
p::PointSize := 4.0*unit::mm:
t::Color := RGB::Red:
t::LineWidth := 1.0*unit::mm:
```

The help page of each primitive provides a list of all attributes the primitive is reacting to.

Certain attributes, such as axes style, the visibility of grid lines in the background etc. are associated with the whole scene rather than with the individual primitives. These attributes may be included in the `plot` call:

```
plot(g, t, p, GridVisible = TRUE)
```



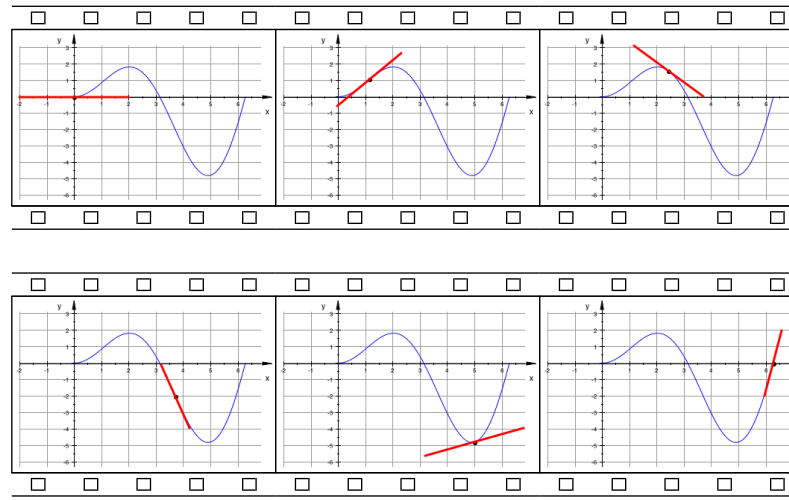
As explained in detail in Section "The Full Picture" on page 11-46, the `plot` command automatically embeds the graphical primitives in a coordinate system, which in turn is embedded in a graphical scene, which is drawn inside a canvas. The various attributes associated with the general appearance of the whole picture are associated with these "grouping structures." A concise list of all such attributes is provided on the help pages of the corresponding primitives: `plot::Canvas`, `plot::Scene2d`, `plot::Scene3d`, `plot::CoordinateSystem2d`, and `plot::CoordinateSystem3d`, respectively.

The MuPAD® object browser allows to select each primitive in the plot. After selection, the attributes of the primitive can be changed interactively in the property inspector (see page 11-50).

Next, we wish to demonstrate an animation. It is remarkably simple to generate an animated picture. Going back to our previous example with the tangent to the function graph, we want to let the point x_0 at which the tangent is added move along the graph of the function. In MuPAD, you do not need to create an animation frame by frame. Instead, each primitive can be told to animate itself by

simply defining it with a symbolic animation parameter and adding an animation range for this parameter. Static and animated objects can be mixed and rendered together. The static function graph of $f(x)$ used in the previous plot is recycled for the animation. The graphical point at $(x_0, f(x_0))$ and the tangent through this point shall be animated using the coordinate `x0` as the animation parameter. Deleting its value set above, we can use the same definitions as before, now with a symbolic `x0`. We just have to add the range specification `x0=0..2*PI` for this parameter:

```
delete x0:
dx := 2/sqrt(1 + f'(x0)^2):
p := plot::Point2d(x0, f(x0), x0 = 0..2*PI,
                   Color = RGB::Black,
                   PointSize = 2.0*unit::mm):
t := plot::Line2d([x0 - dx, f(x0) - f'(x0)*dx],
                  [x0 + dx, f(x0) + f'(x0)*dx],
                  x0 = 0..2*PI, Color = RGB::Red,
                  LineWidth = 1.0*unit::mm):
plot(g, p, t, GridVisible = TRUE)
```



Details on animations and further examples are provided starting on page 11-71.

We summarize the construction principles for graphics with MuPAD's `plot` library:

- *Graphical scenes are built from graphical primitives. The section starting on page 11-54 provides a survey of the primitives that are available in the* `plot` *library.*
- *Primitives generated by the* `plot` *library are symbolic objects that are not rendered directly. The call* `plot(Primitive1, Primitive2, …)` *generates the pictures.*
- *Graphical attributes are specified as equations* `AttributeName = Value`*. Attributes for a graphical primitive may be passed in the call that generates the primitive. The help page of each primitive provides a complete list of all attributes the primitive reacts to.*
- *Attributes determining the general appearance of the picture may be given in the* `plot` *call. The help pages of* `plot::CoordinateSystem2d`*,* `plot::CoordinateSystem3d`*,* `plot::Scene2d`*,* `plot::Scene3d`*, and* `plot::Canvas`*, respectively, provide a complete list of all attributes determining the general appearance.*
- *Almost all attributes can be changed interactively in the viewer.*
- *Presently, 2D and 3D plots are strictly separated. Objects of different dimension cannot be rendered in the same plot.*
- *Animations are not created frame by frame but per object (see also Section 11.9). An object is animated by generating it with a symbolic animation parameter and providing a range for this parameter in the generating call. The section starting on page 11-71 provides further details on animations.*

Presently, it is not possible to add objects to an existing plot. However, using animations, it is possible to let primitives appear one after another in the animated picture.

Some Examples

Example 1: We wish to visualize the interpolation of a discrete data sample by cubic splines. First, we define the data sample, consisting of a list of points $[[x_1, y_1], [x_2, y_2], \dots]$. Suppose they are equidistant sample points from the graph of the function $f(x) = x \cdot e^{-x} \cdot \sin(5\,x)$:

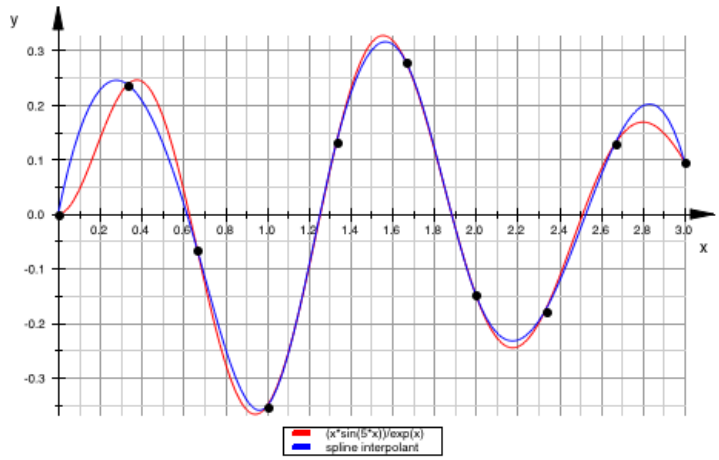
```
f := x -> x*exp(-x)*sin(5*x):
data := [[i, f(i)] $ i = 0..3 step 1/3]:
```

We use `numeric::cubicSpline` to define the cubic spline interpolant through these data:

```
S := numeric::cubicSpline(op(data)):
```

The plot shall consist of the function $f(x)$ that provides the data of the sample points and of the spline interpolant $S(x)$. The graphs of $f(x)$ and $S(x)$ are generated via `plot::Function2d`. The data points are plotted as a `plot::PointList2d`:

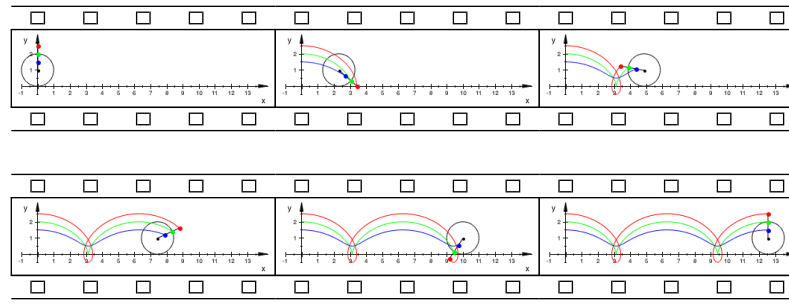
```
plot(plot::Function2d(f(x), x = 0..3, Color = RGB::Red,
                      LegendText = expr2text(f(x))),
     plot::Function2d(S(x), x = 0..3, Color = RGB::Blue,
                      LegendText = "spline interpolant"),
     plot::PointList2d(data, Color = RGB::Black),
     GridVisible = TRUE, SubgridVisible = TRUE,
     LegendVisible = TRUE)
```



Example 2: A cycloid is the curve that you get when following a point fixed to a wheel rolling along a straight line. We visualize this construction by an animation in which we use the x coordinate of the hub as the animation parameter. The wheel is realized as a circle. There are 3 points fixed to the wheel: a green point on the rim, a red point inside the wheel and a blue point outside the wheel:

```
WheelRadius := 1:
r :=  [1.5*WheelRadius, 1.0*WheelRadius, 0.5*WheelRadius]:
WheelCenter := [x, WheelRadius]:
WheelRim := plot::Circle2d(WheelRadius, WheelCenter,
                           x = 0..4*PI,
                           LineColor = RGB::Black):
WheelHub := plot::Point2d(WheelCenter, x = 0..4*PI,
                          PointColor = RGB::Black):
WheelSpoke := plot::Line2d(WheelCenter,
   [WheelCenter[1] + max(r)*sin(x),
    WheelCenter[2] + max(r)*cos(x)],
   x = 0..4*PI, LineColor = RGB::Black):
color:= [RGB::Red, RGB::Green, RGB::Blue]:
for i from 1 to 3 do
 Point[i] := plot::Point2d([WheelCenter[1] + r[i]*sin(x),
                            WheelCenter[2] + r[i]*cos(x)],
                            x = 0..4*PI,
                           PointColor = color[i],
                           PointSize = 2.0*unit::mm):
 Cycloid[i] := plot::Curve2d([y + r[i]*sin(y),
                              WheelRadius + r[i]*cos(y)],
                             y = 0..x, x = 0..4*PI,
                             LineColor = color[i]):
end_for:
```

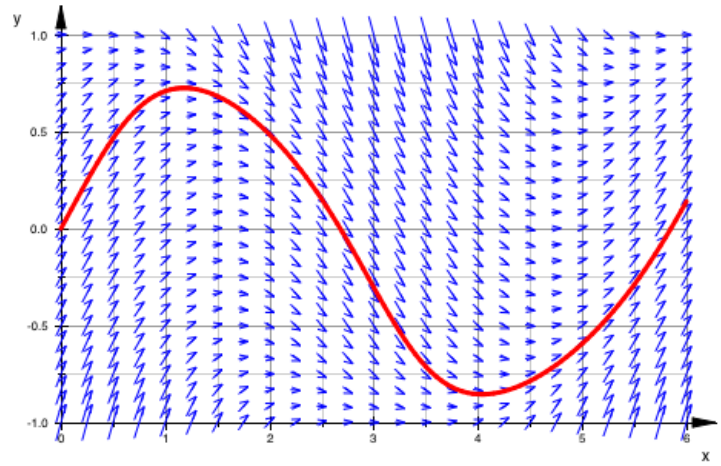
```
plot(WheelRim, WheelHub, WheelSpoke, Point[i] $ i = 1..3,
     Cycloid[i] $ i = 1..3, Scaling = Constrained,
     Width = 120*unit::mm, Height = 35*unit::mm)
```



Example 3: We wish to visualize the solution of the ordinary differential equation (ODE) $y'(x) = -y(x)^3 + \cos(x)$ with the initial condition $y(0) = 0$. The solution shall be drawn together with the vector field $\vec{v}(x, y) = (1, -y^3 + \cos(x))$ associated with this ODE (along the solution curve, the vectors of this field are tangents of the curve). We use `numeric::odesolve2` to generate the solution as a function plot. Since the numerical integrator returns the result as a list of one floating point value, one has to pass the single list entry as $Y(x)[1]$ to `plot::Function2d`. The vector field is generated via `plot::VectorField2d`:

```
f := (x, y) -> -y^3 + cos(x):
Y := numeric::odesolve2(
        numeric::ode2vectorfield({y'(x) = f(x, y),
                                  y(0) = 0}, [y(x)])):
```

```
plot(
 plot::VectorField2d([1, f(x, y)], x = 0..6, y = -1..1,
                     Color = RGB::Blue, Mesh = [25, 25]),
 plot::Function2d(Y(x)[1], x = 0..6, Color = RGB::Red,
                  LineWidth = 0.7*unit::mm),
 GridVisible = TRUE, SubgridVisible = TRUE, Axes = Frame)
```



Example 4: The radius r of an object with rotational symmetry around the x-axis is measured at various x positions:

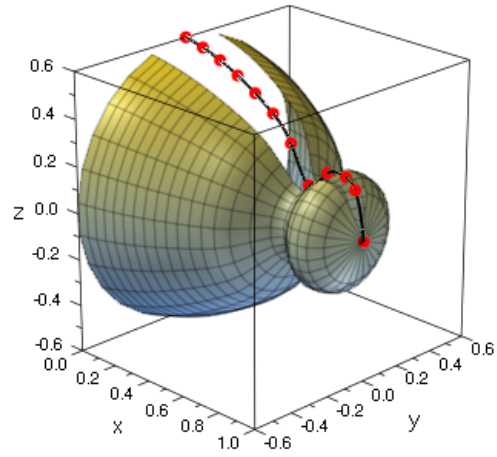
x	0.00	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90	0.95	1.00
$r(x)$	0.60	0.58	0.55	0.51	0.46	0.40	0.30	0.15	0.23	0.24	0.20	0.00

A spline interpolation is used to define a smooth function $r(x)$ from the measurements:

```
samplepoints :=
 [0.00, 0.60], [0.10, 0.58], [0.20, 0.55], [0.30, 0.51],
 [0.40, 0.46], [0.50, 0.40], [0.60, 0.30], [0.70, 0.15],
 [0.80, 0.23], [0.90, 0.24], [0.95, 0.20], [1.00, 0.00]:
r := numeric::cubicSpline(samplepoints):
```

We reconstruct the object as a surface of revolution via `plot::XRotate`. The rotation angle is restricted to a range that leaves a gap in the surface. The spline curve and the sample points are added as a `plot::Curve3d` and a `plot::PointList3d`, respectively, and displayed in this gap:

```
plot(
  plot::XRotate(r(x), x = 0..1,
                AngleRange = 0.6*PI..2.4*PI,
                Color = RGB::MuPADGold),
  plot::Curve3d([x, 0, r(x)], x = 0..1,
                LineWidth = 0.5*unit::mm,
                Color = RGB::Black),
  plot::PointList3d([[p[1], 0, p[2]] $ p in samplepoints],
                    PointSize = 2.0*unit::mm,
                    Color = RGB::Red),
  CameraDirection = [70, -70, 40])
```



The Full Picture: Graphical Trees

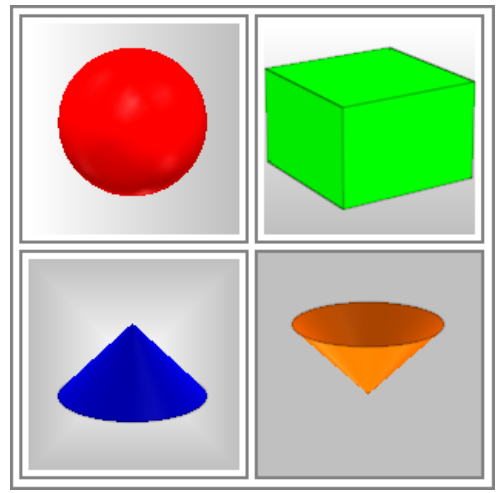
For a full understanding of the interactive features of the viewer to be discussed in the next section, we need to know the structure of a MuPAD® plot as a "graphical tree."

The root is the "canvas"; this is the drawing area into which all parts of the plot are rendered. The type of the canvas object is `plot::Canvas`. Its physical size may be specified via the attributes `Width` and `Height`.

Inside the canvas, one or more "graphical scenes" can be displayed. All of them must be of the same dimension, i.e., objects of type `plot::Scene2d` or `plot::Scene3d`, respectively. The following command displays four different 3D scenes. We set the `BorderWidth` for all objects of type `plot::Scene3d` to some positive value so that the drawing areas of the scenes become visible more clearly:

```
S1 := plot::Scene3d(plot::Sphere(1, [0, 0, 0],
                                 Color = RGB::Red),
                    BackgroundStyle = LeftRight):
S2 := plot::Scene3d(plot::Box(-1..1, -1..1, -1..1,
                              Color = RGB::Green),
                    BackgroundStyle = TopBottom):
S3 := plot::Scene3d(plot::Cone(1, [0, 0, 0], [0, 0, 1],
                               Color = RGB::Blue),
                    BackgroundStyle = Pyramid):
S4 := plot::Scene3d(plot::Cone(1, [0, 0, 1], [0, 0, 0],
                               Color = RGB::Orange),
                    BackgroundStyle = Flat,
                    BackgroundColor = RGB::Gray):
```

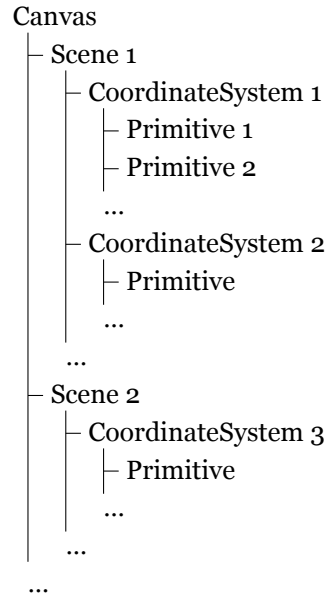
```
plot(plot::Canvas(
  S1, S2, S3, S4,
  Width = 80*unit::mm, Height = 80*unit::mm,
  Axes = None, BorderWidth = 0.5*unit::mm,
  plot::Scene3d::BorderWidth = 0.5*unit::mm))
```



See Section “Layout of Canvas and Scenes” of the online `plot` documentation for details on how the layout of a canvas containing several scenes is set.

Coordinate systems exist inside a 2D scene or a 3D scene and are of type `plot::CoordinateSystem2d` or `plot::CoordinateSystem3d`, respectively. There may be one or more coordinate systems in a scene. Inside the coordinate systems, an arbitrary number of primitives (of the appropriate dimension) can be displayed. Thus, we always have a canvas, containing one or more scenes, with each scene containing one or more coordinate systems. The graphical primitives (or groups of such primitives) are contained in the coordinate systems.

Hence, any MuPAD plot has the following general structure:



This is the "graphical tree" that is displayed by the "object browser" (see page 11-50).

Shortcuts: For simple plots containing primitives inside one coordinate system only that resides in a single scene of the canvas, it would be rather cumbersome to generate the picture by a command such as

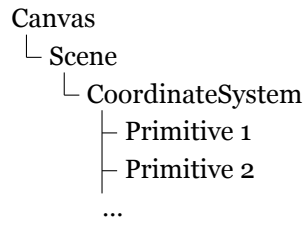
```
plot(
 plot::Canvas(
  plot::Scene2d(
   plot::CoordinateSystem2d(Primitive1, Primitive2, ...))))
```

In fact, the command

```
plot(Primitive1, Primitive2, ...)
```

suffices: It is a shortcut of the above command. It automatically generates a coordinate system that contains the primitives, embedded in a scene which is automatically placed inside a canvas object.

Thus, this command implicitly creates the graphical tree



that is displayed in the object browser.

Viewer, Browser, and Inspector: Interactive Manipulation

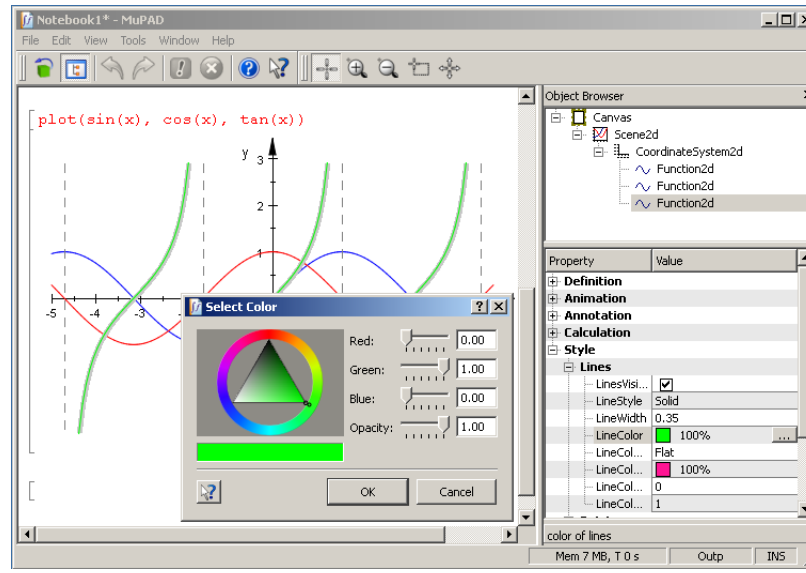
After a plot command is executed in a notebook, the plot will typically appear below the input region, embedded in the notebook. To “activate” the plot, click on it with the left mouse button. This leaves the plot embedded in the notebook, but the notebook menus are replaced by the graphics menus.

Make sure that the item ‘Object Browser’ is enabled under the ‘View’ menu. A pane titled ‘Object Browser’ is visible containing two sub-windows, which we will refer to as the ‘object browser’ and the ‘property inspector.’

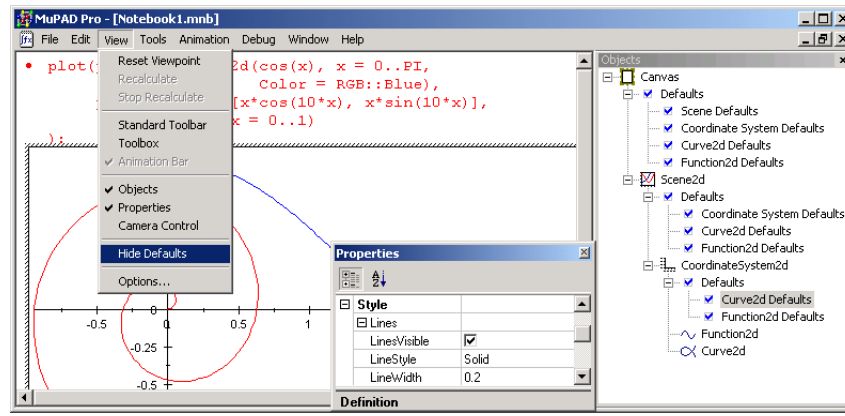
In the object browser, the graphical tree of the plot as introduced in the preceding section is visible. It allows to select any node of the graphical tree by a mouse click.

After selecting an object in the object browser, the corresponding part of the plot is highlighted in some way allowing to identify the selected object visually. The property inspector then displays the attributes the selected object reacts to. For example, for an object of type `Function2d`, the attributes visible in the property inspector are categorized as ‘Definition,’ ‘Animation,’ ‘Annotation,’ ‘Calculation,’ and ‘Style’ with the latter split into the sub-categories (Style of) ‘Lines,’ (Style of) ‘Points,’ (Style of) ‘Asymptotes.’ Opening one of these categories, one finds the attributes and their current values that apply to the currently selected object.

After selection of an attribute with the mouse, its value can be changed:



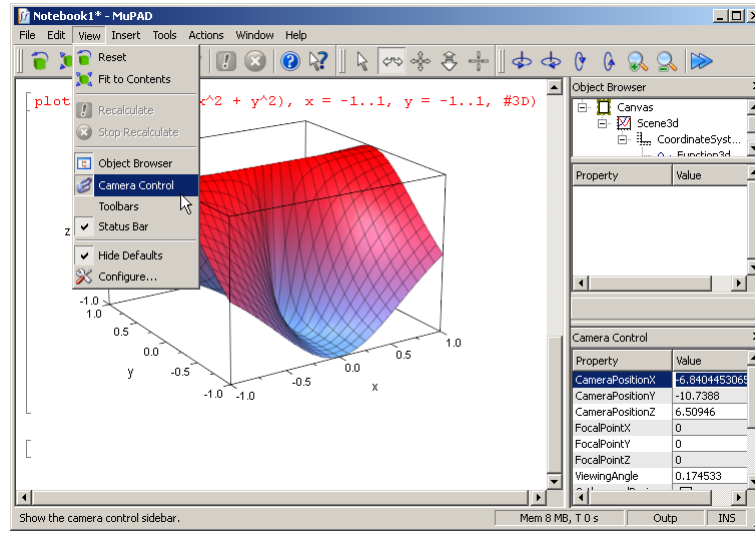
There is a sophisticated way of setting defaults (see Section 11.7 and Section 11.7 for an explanation) for the attributes via the object browser and the property inspector. The 'View' menu provides an item 'Hide Defaults.' Disabling 'Hide Defaults,' the object browser shows entries for defaults:



At each node of the graphical tree, default values for the inheritable attributes can be set via the property inspector. These defaults are valid for all primitives that exist below this node, unless these defaults are re-defined at some other node further down in the tree hierarchy.

This mechanism is particularly useful when there are many primitives of the same kind in the plot. Imagine a picture consisting of 1000 points. If you wish to change the color of all points to green, it would be quite cumbersome to set `PointColor=RGB::Green` in all 1000 points. Instead, you can set `PointColor=RGB::Green` in the `PointColor` defaults entry at some tree node that contains all the points (e.g., the canvas), either directly in the `plot` call or via the object browser/property inspector. Similarly, if there are 1000 points in one scene and another 1000 points in a second scene, you can change the color of all points in the first scene by an appropriate default entry in the first scene, whilst the default entry for the second scene can be set to a different value. See page 11-58 for an example.

A 3D plot can be rotated and shifted by the mouse. Also zooming in and out is possible. In fact, these operations are realized by moving the camera around, closer to, or farther away from the scene, respectively. There is a camera control that may be switched on and off via the 'Camera Control' item of the 'View' menu or the camera icon in the toolbar. It provides the current viewing parameters such as camera position, focal point and the angle of the camera lens:



The section starting on page 11-112 provides more information on cameras.

Primitives

In this section, we give a brief survey of the graphical primitives, grouping constructs, transformations etc. provided by the `plot` library.

The following table lists the 'low-level' primitives:

`Arc2d`, `Arc3d`	circular arc,
`Arrow2d`, `Arrow3d`	arrow,
`Box`	rectangular box in 3D,
`Circle2d`, `Circle3d`	circle,
`Cone`	cone/conical frustum in 3D,
`Cylinder`	cylinder in 3D,
`Ellipse2d`, `Ellipse3d`	ellipse,
`Ellipsoid`	ellipsoid in 3D,
`Line2d`, `Line3d`	(finite or infinite) line,
`Parallelogram2d`, `Parallelogram3d`	parallelogram,
`Plane`	infinte plane in 3D,
`Point2d`, `Point3d`	point,
`PointList2d`, `PointList3d`	list of points,
`Polygon2d`, `Polygon3d`	line segments forming a polygon,
`Rectangle`	rectangle in 2D,
`Sphere`	sphere in 3D,
`SurfaceSet`	surface in 3D (as a collection of triangles),
`SurfaceSTL`	import of 3D stl surfaces,
`Text2d`, `Text3d`	text object.

In addition, there are primitives `plot::Tetrahedron`, `plot::Hexahedron`, `plot::Octahedron`, `plot::Dodecahedron`, and `plot::Icosahedron` for Plato's regular polyhedra and `plot::Waterman` for Waterman polyhedra.

The following table lists the 'high-level' primitives and 'special purpose' primitives:

`Bars2d`, `Bars3d`	(statistical) data plot,
`Boxplot`	(statistical) box plot,
`Conformal`	plot of conformal functions in 2D,
`Curve2d`, `Curve3d`	parameterized curve,
`Cylindrical`	surface in cylindrical coordinates in 3D,
`Density`	density plot in 2D,
`Function2d`, `Function3d`	function graph,
`Hatch`	hatched region in 2D,
`Histogram2d`	(statistical) histogram plot in 2D,
`Implicit2d`	implicitly defined curves in 2D,
`Implicit3d`	implicitly defined surfaces in 3D,
`Inequality`	visualization of inequalities in 2D,
`Iteration`	visualization of iterations in 2D,
`Listplot`	finite list of points,
`Lsys`	Lindenmayer system in 2D,
`Matrixplot`	visualization of matrices in 3D,
`Ode2d`, `Ode3d`	graphical solution of an ODE,
`Piechart2d`, `Piechart3d`	(statistical) pie chart,
`Polar`	curve in polar coordinates in 2D,
`Prism`	prism in 3D,
`Pyramid`	pyramids and their frustrums in 3D,
`QQPlot`	(statistical) quantile-quantile plots,
`Raster`	raster and bitmap plots in 2D,
`Rootlocus`	dependency of rational root on parameter,
`Scatterplot`	(statistical) scatter plot in 2D,
`Sequence`	sequence (given by formula) in 2D,
`SparseMatrixplot`	sparsity pattern of a matrix, 2D,
`Spherical`	surface in spherical coordinates in 3D,
`Streamlines2d`	vector field visualization in 2D,
`Sum`	visualization of symbolic sums in 2D,
`Sweep`	sweep surface spanned by two curves in 3D,
`Tube`	tube plot in 3D,
`Turtle`	turtle plot in 2D,
`VectorField2d`, `VectorField3d`	vector field plot,
`Surface`	parameterized surface in 3D,
`XRotate`	surface of revolution in 3D,
`ZRotate`	surface of revolution in 3D.

The following table lists the various light sources available to illuminate 3D plots:

`AmbientLight`	ambient (undirected) light,
`DistantLight`	directed light (sun light),
`PointLight`	(undirected) point light,
`SpotLight`	(directed) spot light.

The following table lists various grouping constructs:

`Canvas`	drawing area,
`Scene2d`	container for 2D coordinate systems,
`Scene3d`	container for 3D coordinate systems,
`CoordinateSystem2d`	container for 2D primitives,
`CoordinateSystem3d`	container for 3D primitives,
`Group2d`	group of primitives in 2D,
`Group3d`	group of primitives in 3D.

Primitives or groups of primitives can be transformed by the following objects:

`Scale2d`, `Scale3d`	scaling,
`Reflect2d`, `Reflect3d`	reflection,
`Rotate2d`, `Rotate3d`	rotation,
`Translate2d`, `Translate3d`	translation,
`Transform2d`, `Transform3d`	general linear transformation.

Additionally, there are:

`Camera`	camera in 3D,
`ClippingBox`	clipping box in 3D.

Attributes

The `plot` library provides more than 400 attributes for fine-tuning of the graphical output. Because of this large number, the attributes are grouped into various categories in the object browser (see page 11-50) and the documentation:

category	meaning
Animation	parameters relevant for animations
Annotation	footer, header, titles, and legends
Axes	axes style and axes titles
Calculation	numerical evaluation
Definition	parameters that change the object itself
Grid Lines	grid lines in the background (rulings)
Layout	layout parameters for canvas and scenes
Style	parameters that do not change the objects but their presentation (visibility, color, line width, point size etc.)
Arrows	style parameters for arrows
Lines	style parameters for line objects
Points	style parameters for point objects
Surface	style parameters for surface objects in 3D and filled areas in 2D
Tick Marks	axes tick marks: style and labels

On the help page for each primitive, there is a complete list of all attributes the primitive reacts to. Clicking on an attribute, you are lead to the help page for this attribute which provides all the necessary information concerning its semantics and admissible values. The examples on the help page demonstrate the typical use of the attribute.

Default Values

Most attributes have a default value that is used if no value is specified explicitly. As an example, we consider the attribute `LinesVisible` that is used by several primitives such as `plot::Box`, `plot::Circle2d`, `plot::Cone`, `plot::Curve2d`, `plot::Raster` etc. Although they all use the same attribute named `LinesVisible`, its default value differs among the different primitive types. The specific defaults

are accessible by passing the slots `plot::Box::LinesVisible`, `plot::Circle2d::LinesVisible` etc. to `plot::getDefault`:

```
plot::getDefault(plot::Box::LinesVisible),
plot::getDefault(plot::Circle2d::LinesVisible),
plot::getDefault(plot::Cone::LinesVisible),
plot::getDefault(plot::Raster::LinesVisible)
```

```
TRUE, TRUE, TRUE, FALSE
```

If any of the default values provided by the MuPAD® system do not seem appropriate for your applications, change these defaults via `plot::setDefault`:

```
plot::setDefault(plot::Box::LinesVisible = FALSE)
```

```
TRUE
```

(The return value is the previously valid default value.) Several defaults can be changed simultaneously:

```
plot::setDefault(plot::Box::LinesVisible = FALSE,
                 plot::Circle2d::LinesVisible = FALSE,
                 plot::Circle2d::Filled = TRUE)
```

```
FALSE, TRUE, FALSE
```

```
plot::getDefault(plot::Box::LinesVisible)
```

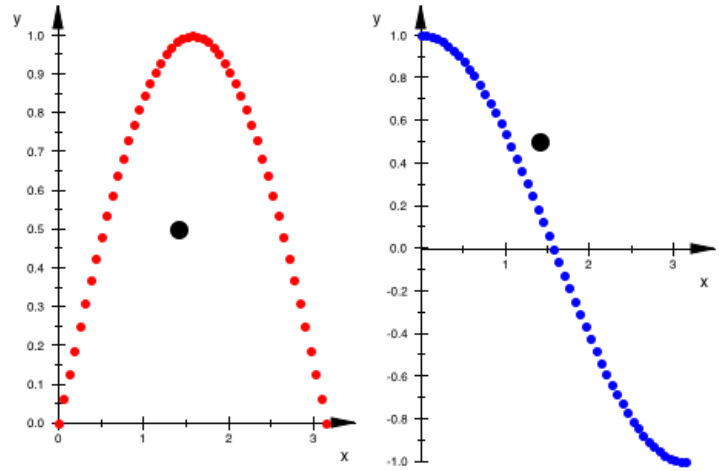
```
FALSE
```

Inheritance of Attributes

The setting of default values for attributes is quite sophisticated. Assume that you have two scenes that are to be displayed in one single canvas. Both scenes consist of 51 graphical points, each:

```
points1 := plot::Point2d(i/50*PI, sin(i/50*PI)) $ i=0..50:
points2 := plot::Point2d(i/50*PI, cos(i/50*PI)) $ i=0..50:
S1 := plot::Scene2d(points1):
S2 := plot::Scene2d(points2):
```

```
plot(S1, S2)
```



If we wish to color all points in both scenes red, we can set a default for the point color in the `plot` command:

```
plot(S1, S2, PointColor = RGB::Red)
```

If we wish to color all points in the first scene red and all points in the second scene blue, we can give each of the points the desired color in the generating call to `plot::Point2d`. Alternatively, we can set separate defaults for the point color inside the scenes:

```
S1 := plot::Scene2d(points1, PointColor = RGB::Red):
S2 := plot::Scene2d(points2, PointColor = RGB::Blue):
plot(S1, S2)
```

Here is the general rule for setting defaults inside a graphical tree (see page 11-46):

> When an attribute is specified in a node of the graphical tree, the specified value serves as the default value for the attribute for all primitives that exist in the subtree starting from that node.

The default value can be overwritten by another value at each node further down in the subtree (e.g., finally, by a specific value in the primitives). In the following call, the default color 'red' is set in the canvas. This value is accepted and used in

the first scene. The second scene sets the default color 'blue' overriding the default value 'red' set in the canvas. Additionally, there is an extra point with a color explicitly set to 'black'. This point ignores the defaults set further up in the tree hierarchy:

```
extrapoint := plot::Point2d(1.4, 0.5,
                            PointSize = 3*unit::mm,
                            PointColor = RGB::Black):
S1 := plot::Scene2d(points1, extrapoint):
S2 := plot::Scene2d(points2, extrapoint,
                    PointColor = RGB::Blue):
plot(plot::Canvas(S1, S2, PointColor = RGB::Red))
```

The following call generates the same result. Note that the `plot` command automatically creates a canvas object and passes the attribute `PointColor=RGB::Red` to the canvas:

```
plot(S1, S2, PointColor = RGB::Red)
```

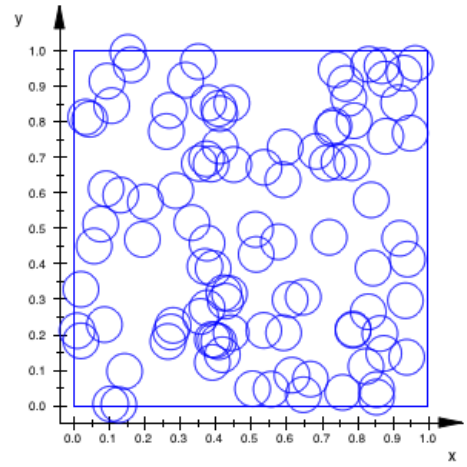
We note that there are different primitives that react to the same attribute. We used `LinesVisible` in the previous section to demonstrate this fact. One of the rules for inheriting attributes in a graphical tree is:

> If an attribute such as `LinesVisible=TRUE` is specified in some node of the graphical tree, *all* primitives below this node that react to this attribute use the specified value as the default value.
> If a type specific attribute (using the "fully qualified" syntax) such as `plot::Circle2d::LinesVisible=TRUE` is specified, the new default value is valid only for primitives of that specific type.

In the following example, we consider 100 randomly placed circles with a rectangle indicating the area into which all circle centers are placed:

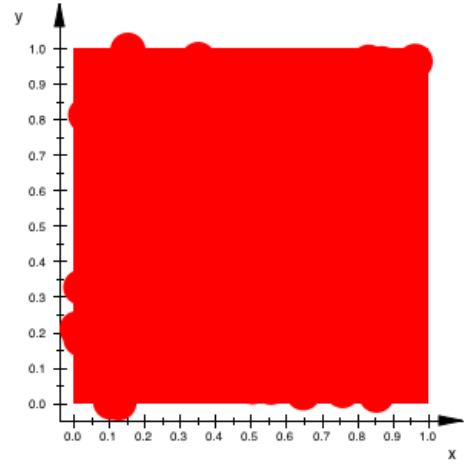
```
rectangle := plot::Rectangle(0..1, 0 ..1):
circles := plot::Circle2d(0.05, [frandom(), frandom()])
                                          $ i = 1..100:
```

```
plot(rectangle, circles, Axes = Frame)
```



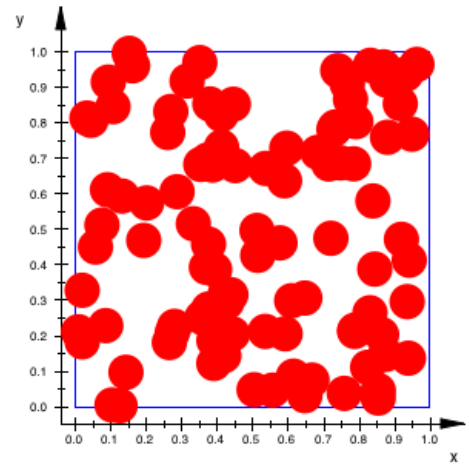
We wish to turn the circles into filled circles by setting `Filled=TRUE` and `LinesVisible=FALSE`:

```
plot(rectangle, circles,
     Filled = TRUE, FillPattern = Solid,
     LinesVisible = FALSE, Axes = Frame)
```



This is not quite what we wanted: Not only the circles, but also the rectangle reacts to the attributes `Filled`, `FillPattern`, and `LinesVisible`. The following command restricts these attributes to the circles:

```
plot(rectangle, circles,
     plot::Circle2d::Filled = TRUE,
     plot::Circle2d::FillPattern = Solid,
     plot::Circle2d::LinesVisible = FALSE,
     Axes = Frame)
```



Primitives Requesting Special Scene Attributes: "Hints"

The default values for the attributes are chosen in such a way that they produce reasonable pictures in "typical" plot commands. For example, the default axes type in 3D scenes is `Axes=Boxed`, because this is the most appropriate axes type in the majority of 3D plots:

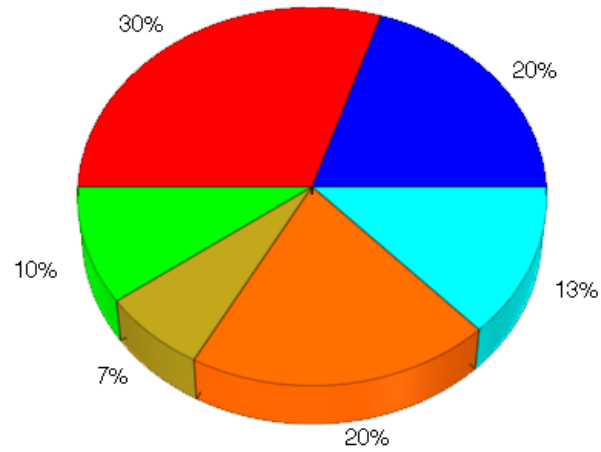
```
plot::getDefault(plot::CoordinateSystem3d::Axes)
```

```
    Boxed
```

However, there are exceptions. E.g., a plot containing a 3D pie chart should probably have no axes at all. Since it is not desirable to use `Axes=None` as the default setting for all plots, exceptional primitives such as `plot::Piechart3d` are

given a chance to override the default axes automatically. In a pie chart plot, no axes are used by default:

```
plot(plot::Piechart3d([20, 30, 10, 7, 20, 13],
             Titles = [1 = "20%", 2 = "30%",
                       3 = "10%", 4 =  "7%",
                       5 = "20%", 6 = "13%"]))
```



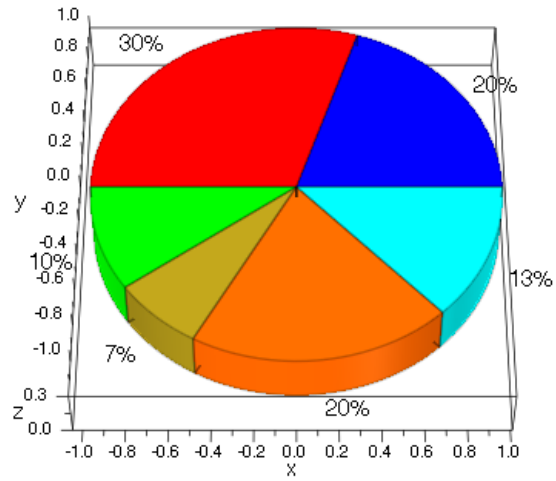
Note that the attribute `Axes` that is in charge of switching axes on and off is a scene attribute and hence cannot be processed by pie chart objects directly. Hence, a separate mechanism for requesting special scene attributes by primitives is implemented: so-called “hints.”

A “hint” is an attribute of one of the superordinate nodes in the graphical tree, i.e., an attribute of a coordinate system, a scene, or the canvas. The help pages of the primitives give information on what hints are sent by the primitive. If several primitives send conflicting hints, the first hint is used.

Hints are implemented internally and cannot be switched off by the user. Note, however, that the hints concept only concerns *default values* for attributes. You can always specify the attribute explicitly if you think that a default value or a hint is not appropriate.

As an example, we explicitly request `Axes = Boxed` in the following call:

```
plot(plot::Piechart3d([20, 30, 10, 7, 20, 13],
             Titles = [1 = "20%", 2 = "30%",
                       3 = "10%", 4 =  "7%",
                       5 = "20%", 6 = "13%"]),
     Axes = Boxed)
```



The Help Pages of Attributes

We have a brief look at a typical help page for a plot attribute to explain the information provided there:

`UMesh, VMesh, USubmesh, VSubmesh` – number of sample points

The attributes `UMesh` etc. determine the number of sample points used for the numerical approximation of parameterized plot objects such as curves and surfaces.

→ **Examples**

Attribute	Type	Value
USubmesh, VSubmesh	inherited	non-negative integer
UMesh, VMesh	inherited	positive integer

See Also

`AdaptiveMesh`, `Mesh`, `Submesh`, `XMesh`, `YMesh`, `ZMesh`

Objects reacting to `UMesh`, `VMesh`, `USubmesh`, `VSubmesh`

`plot::Surface`, `plot::Cylindrical`, `plot::XRotate`, `plot::Spherical`, `plot::ZRotate`	25 (`UMesh`, `VMesh`) 0 (`USubmesh`, `VSubmesh`)
`plot::Tube`	60 (`UMesh`) 11 (`VMesh`) 0 (`USubmesh`) 1 (VSubmesh)

...

Detail

- Many plot objects have to be evaluated numerically on a discrete mesh.
  ...

Example 1 ...

In these help pages,

- the table entry "**Type**" ("inherited") states that these attributes may not only be specified in the generating call of graphical primitives. They can also be specified at higher nodes of a graphical tree to be inherited by the primitives in the corresponding subtree (see page 11-58).
- the table entry "**Value**" states the type of the number `n` that is admissible when passing the attributes `UMesh=n`, `VMesh=n` etc.
- the sections "**Object types reacting to** `UMesh`" etc. provide complete listings of all primitives reacting to the attribute(s) together with the specific default values.

  the "**Details**" section gives further information on the semantics of the attribute(s). Finally, there are "**Examples**" of plot commands using the described attribute(s).

Colors

The most prominent plot attribute, available for almost all primitives, is `Color`, setting the 'primary' color of an object. MuPAD®'s `plot` library knows 3 different types of primary colors:

- The attribute `PointColor` refers to the color of points in 2D and 3D (of type `plot::Point2d` and `plot::Point3d`, respectively).
- The attribute `LineColor` refers to the color of line objects in 2D and 3D. This includes the color of function graphs in 2D, curves in 2D and 3D, polygon lines in 2D and 3D etc. Also 3D objects such as function graphs in 3D, parametrized surfaces etc. react to the attribute `LineColor`; it defines the color of the coordinate mesh lines that are displayed on the surface.
- The attribute `FillColor` refers to the color of closed and filled polygons in 2D and 3D as well as hatched regions in 2D. Further, it sets the surface color of function graphs in 3D, parametrized surfaces etc. This includes spheres, cones etc.

The primitives also accept the attribute `Color` as a shortcut for one of these colors. Depending on the primitive, either `PointColor`, `LineColor`, or `FillColor` is set with the `Color` attribute.

RGB Colors

MuPAD® uses the RGB color model, i.e., colors are specified by lists `[r, g, b]` of red, green, and blue values between 0 and 1, or in HTML syntax as a hash mark followed by a six-digit hexadecimal number, as in `#cc0099`. Black and white correspond to `[0, 0, 0]` (`#000000`) and `[1, 1, 1]` (`#ffffff`), respectively. The library `RGB` contains numerous color names with corresponding RGB values:

```
RGB::Black, RGB::White, RGB::Red, RGB::SkyBlue
```

$$[0.0, 0.0, 0.0],\ [1.0, 1.0, 1.0],\ [1.0, 0.0, 0.0],\ [0.0, 0.8, 1.0]$$

List all color names available in the `RGB` library via `info(RGB)`. Alternatively, there is the command `RGB::ColorNames()` returning a complete list of names. With

```
RGB::ColorNames(Red)
  [CadmiumRedDeep, CadmiumRedLight, CardinalRed,
     CarmineRed, DarkRed, EnglishRed, IndianRed,
     OrangeRed, PermanentRedViolet, Red, VenetianRed,
     VioletRed, VioletRedMedium, VioletRedPale]
```

one searches for all color names that contain 'Red.' To actually see the colors found, use `RGB::plotColorPalette`:

```
RGB::plotColorPalette("red")
```

After exporting the color library via `use(RGB)`, you can use the color names in the short form `Black`, `White`, `IndianRed` etc. Note, however, that this command will assign values to a little over 350 identifiers, precluding their use in other meanings.

RGBa color values consist of lists `[r, g, b, a]` containing a fourth entry: the "opacity" value a between 0 and 1. (In HTML-like notation, the hexadecimal specification has eight characters.) For $a = 0$, a surface patch painted with this RGBa color is fully transparent (i.e., invisible). For $a = 1$, the surface patch is opaque, i.e., it hides plot objects that are behind it. For $0 < a < 1$, the surface patch is semi-transparent, i.e., plot objects behind it shine through.

RGBa color values can be constructed easily via the `RGB` library. One only has to append a fourth entry to the $[r, g, b]$ lists provided by the color names. The easiest

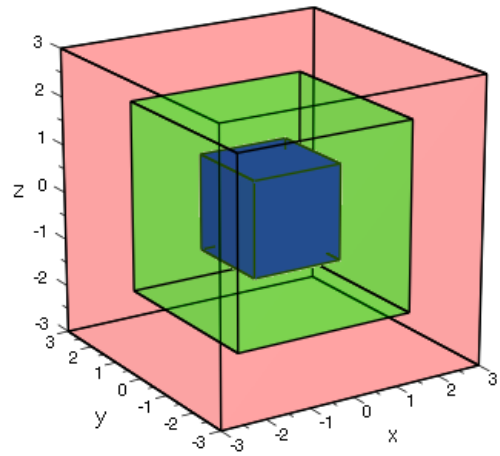
way to do this is to append the list $[a]$ to the RGB list via the concatenation operator '.'. We create a semi-transparent gray:

```
RGB::Gray.[0.5]
```

$$[0.752907, 0.752907, 0.752907, 0.5]$$

The following command plots a highly transparent red box, containing a somewhat less transparent green box with an opaque blue box inside:

```
plot(plot::Box(-3..3, -3..3, -3..3,
               FillColor = RGB::Red.[0.2]),
     plot::Box(-2..2, -2..2, -2..2,
               FillColor = RGB::Green.[0.3]),
     plot::Box(-1..1, -1..1, -1..1,
               FillColor = RGB::Blue),
     LinesVisible = TRUE, LineColor = RGB::Black,
     Scaling = Constrained)
```



HSV Colors

Besides the RGB model, there are various other popular color formats used in computer graphics. One is the HSV model (Hue, Saturation, Value). The RGB library provides the routines RGB::fromHSV and RGB::toHSV to convert HSV

colors to RGB colors and vice versa:

```
hsv := RGB::toHSV(RGB::Orange)
```

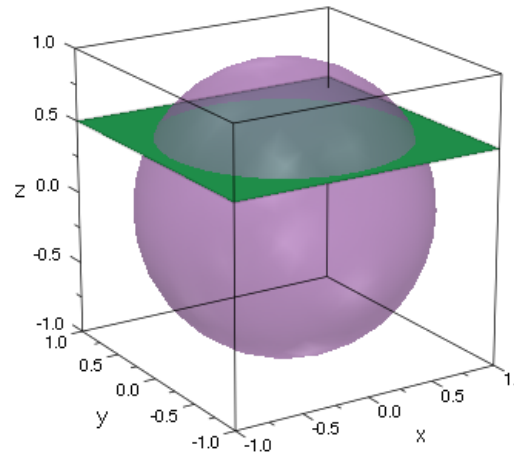
$$[24.0, 1.0, 1.0]$$

```
RGB::fromHSV(hsv) = RGB::Orange
```

$$[1.0, 0.4, 0.0] = [1.0, 0.4, 0.0]$$

With the `RGB::fromHSV` utility, all colors in a MuPAD plot can be specified easily as HSV colors. For example, the color 'violet' is given by the HSV values `[290, 0.4, 0.6]`, whereas 'dark green' is given by the HSV specification `[120, 1, 0.4]` (120 for 'green,' 0.4 for 'dark'). Hence, a semi-transparent violet sphere intersected by an opaque dark green plane may be specified as follows:

```
plot(plot::Sphere(1, [0, 0, 0],
        Color = RGB::fromHSV([290, 0.4, 0.6, 0.5])),
     plot::Surface([x, y, 0.5], x = -1..1, y = -1..1,
        Mesh = [2, 2],
        Color = RGB::fromHSV([120, 1, 0.4])))
```



Animations

Generating Simple Animations

Each primitive of the `plot` library knows how many specifications of type "range" it has to expect.

> *Whenever a graphical primitive receives a "surplus" range specification by an equation such as* `a = amin..amax`*, the parameter* `a` *is interpreted as an "animation parameter" assuming values from* `amin` *to* `amax`*.*

What is a "surplus" range? This is a specification that is not necessary for generating a static object. For example, a static univariate function graph in 2D such as

```
plot::Function2d(sin(x), x = 0..2*PI):
```

expects one plot range for the x coordinate, whereas a static bi-variate function graph in 3D expects two plot ranges for the x and y coordinate:

```
plot::Function3d(sin(x^2 + y^2), x = 0..2, y = 0..2):
```

A static contour plot in 2D expects 2 ranges for the x and y coordinate:

```
plot::Implicit2d(x^2 + y^2 - 1, x = -2..2, y = -2..2):
```

A static contour plot in 3D expects 3 ranges for the x, y, and z coordinate:

```
plot::Implicit3d(x^2 + y^2 + z^2 - 1, x = -2..2,
                 y = - 2..2, z = - 2..2):
```

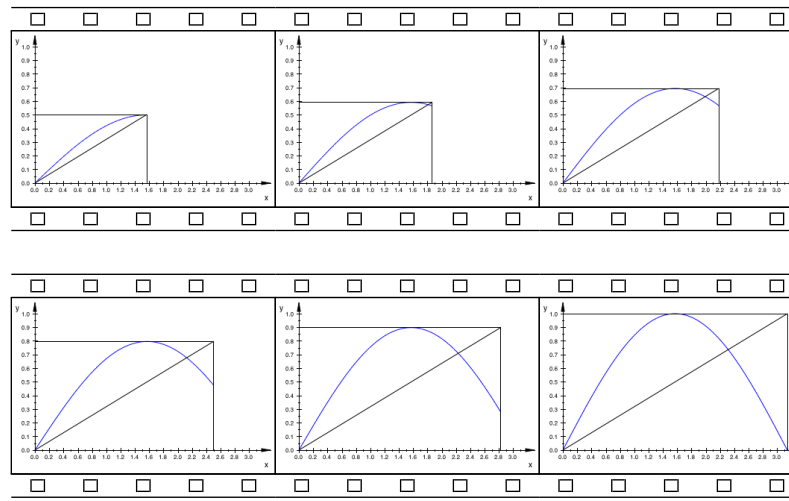
Any additional equation involving a range makes the primitive create an animation. Since a static 2D function graph expects only one range (for the independent variable), the following call is interpreted as an instruction to generate an animated 2D function graph:

```
plot::Function2d(sin(x - a), x = 0..2*PI, a = 0..2*PI):
```

Thus, it is very easy indeed to create animated objects: just pass a "surplus" range equation `a = amin..amax` to the generating call of the primitive. The name `a` of the animation parameter is irrelevant; any symbolic name may be used. All other

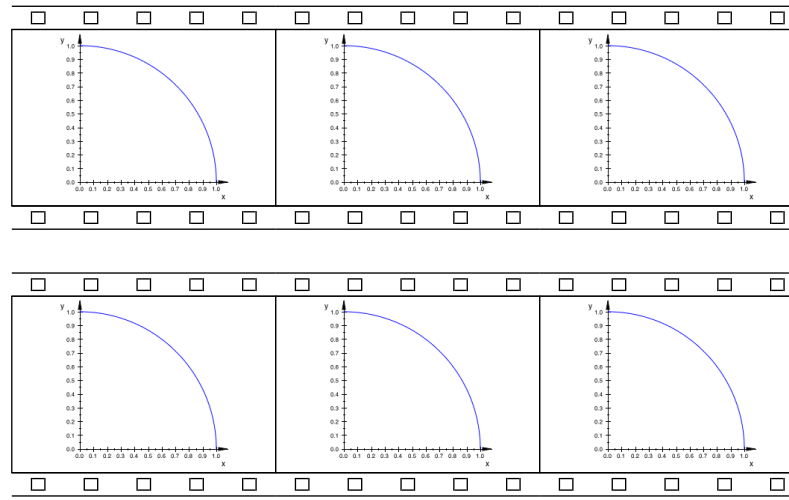
entries and attributes of the primitive that are symbolic expressions of the animation parameter will be animated. In the following call, both the function expression as well as the x range of the function graph depend on the animation parameter. Also, the ranges defining the width and the height of the rectangle as well as the end point of the line depend on it:

```
plot(plot::Function2d(a*sin(x), x = 0..a*PI, a = 0.5..1),
     plot::Rectangle(0..a*PI, 0..a, a = 0.5..1,
                     LineColor = RGB::Black),
     plot::Line2d([0, 0], [PI*a, a], a = 0.5 ..1,
                  LineColor = RGB::Black))
```



Here is an animated arc whose radius and "angle range" depend on the animation parameter:

```
plot(plot::Arc2d(1 + a, [0, 0], AngleRange = 0..a*PI,
                 a = 0..1))
```



Here, an additional range specification for the attribute `AngleRange` is used. However, the attribute is identified by its name as an admissible attribute for the primitive and thus not assumed to be the specification of an animation parameter.

> **Beware!** *Do make sure that attributes are specified by their correct names. If an incorrect attribute name is used, it may be mistaken for an animation parameter!*

In the following examples, we wish to define a static semi-circle, using `plot::Arc2d` with `AngleRange=0..PI`. However, `AngleRange` is spelled incorrectly. A plot is created. It is an animated full circle with the animation parameter `AngelRange`!

```
plot(plot::Arc2d(1, [0, 0], AngelRange = 0..PI))
```

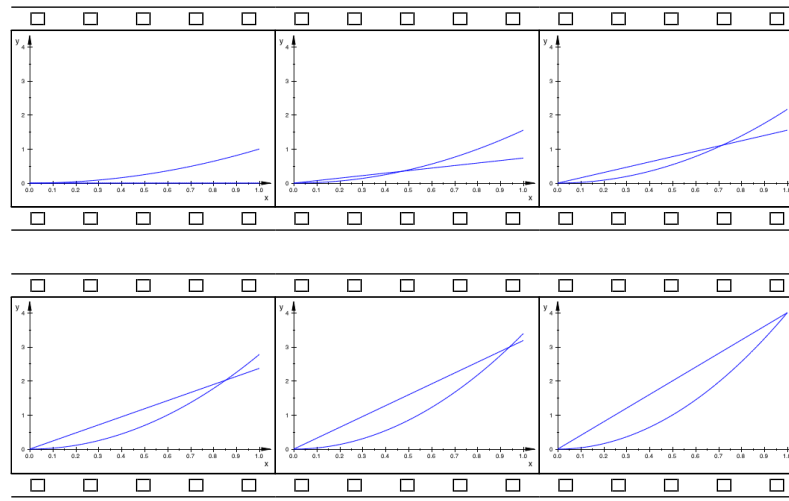
The animation parameter may be any symbolic parameter (identifier or indexed identifier) that is different from the symbols used for the mandatory range

specifications (such as the names of the independent variables in function graphs). The parameter must also be different from any of the protected names of the plot attributes.

> *Animations are created object by object. The names of the animation parameters in different objects need not coincide.*

In the following example, different names `a` and `b` are used for the animation parameters of the two functions:

```
plot(plot::Function2d(4*a*x, x = 0..1, a = 0..1),
     plot::Function2d(b*x^2, x = 0..1, b = 1..4))
```

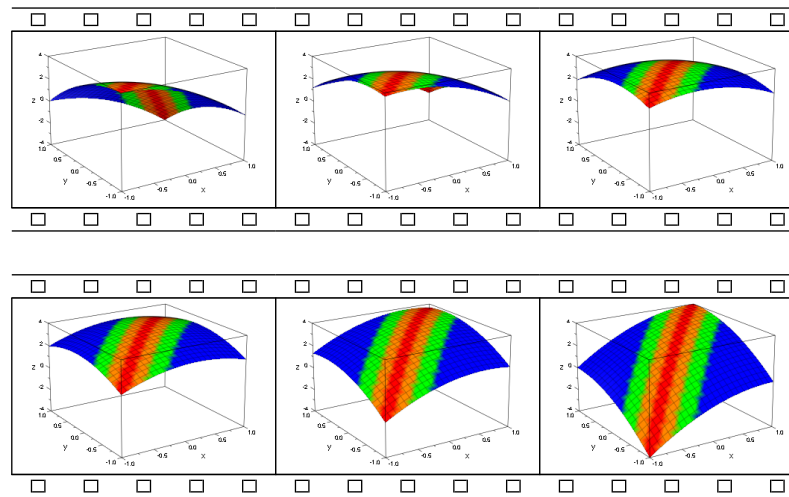


An animation parameter is a global symbolic name. It can be used as a global variable in procedures defining the graphical object. The following example features the 3D graph of a bi-variate function that is defined by a procedure using the globally defined animation parameter. Further, a fill color function `mycolor` is defined that changes the color in the course of the animation. It could use the animation parameter as a global parameter, just like the function f does. Alternatively, the animation parameter may be declared as an additional input parameter. Refer to the help page of `FillColorFunction` to find out how many input parameters the fill color function expects and which of the input parameters

receives the animation parameter. One finds that for `Function3d`, the fill color function is called with the coordinates x, y, z of the points on the graph. The next input parameter (the 4th argument of `mycolor`) is the animation parameter:

```
f := (x, y) -> 4 - (x - a)^2 - (y - a)^2:
mycolor := proc(x, y, z, a)
  local t;
begin
  t := sqrt((x - a)^2 + (y - a)^2):
  if t < 0.1 then return(RGB::Red)
  elif t < 0.4 then return(RGB::Orange)
  elif t < 0.7 then return(RGB::Green)
  else return(RGB::Blue)
  end_if;
end:
```

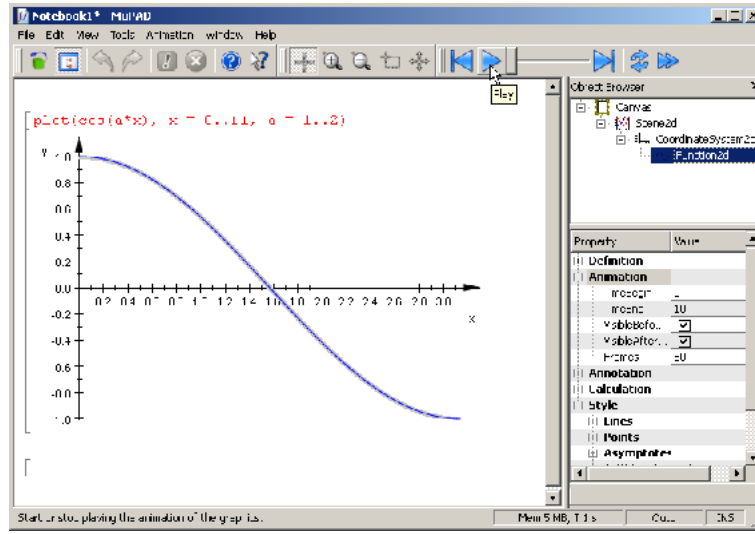
```
plot(plot::Function3d(f, x = -1..1, y = -1..1, a = -1..1,
                      FillColorFunction = mycolor))
```



Playing Animations

When an animated plot is created in a MuPAD® notebook, the first frame of the animation appears as a static picture below the input region. To start the

animation, click on the plot. An icon for stopping and re-starting the animation will appear (make sure the item 'Animation Bar' of the 'View' menu is enabled):



One can also use the slider to animate the picture "by hand."

The Number of Frames and the Time Range

By default, an animation consists of 50 frames. The number of frames can be set to be any positive number n by specifying the attribute `Frames=n`. This attribute can be set in the generating call of the animated primitives, or at some higher node of the graphical tree. In the latter case, this attribute is inherited by all primitives that exist below the node. With `a=amin..amax`, `Frames=n`, the i-th frame consists of a snapshot of the primitive with

$$a = a_{\min} + \frac{i-1}{n-1} \cdot (a_{\max} - a_{\min}), \quad i = 1, \ldots, n.$$

Increasing the number of frames does not mean that the animation runs longer; the renderer does not work with a fixed number of frames per second but processes all frames within a fixed time interval.

In the background, there is a "real time clock" used to synchronize the animation of different animated objects. An animation has a time range measured by this clock. The time range is set by the attributes `TimeBegin=t0`, `TimeEnd=t1` or, equivalently, `TimeRange=t0..t1`, where `t0`, `t1` are real numerical values representing physical times in seconds. These attribute can be set in the generating call of the animated primitives, or at some higher node of the graphical tree. In the latter case, these attributes are inherited by all primitives that exist below the node.

The absolute value of `t0` is irrelevant if all animated objects share the same time range. Only the time difference `t1 - t0` matters. It is (an approximation of) the physical time in seconds that the animation will last.

The parameter range `amin..amax` *in the specification of the animation parameter* `a=amin..amax` *together with* `Frames=n` *defines an equidistant time mesh in the time interval set by* `TimeBegin=t0` *and* `TimeEnd=t1`. *The frame with* `a=amin` *is visible at the time* `t0`, *the frame with* `a=amax` *is visible at the time* `t1`.

With the default `TimeBegin=0`, *the value of the attribute* `TimeEnd` *gives the physical time of the animation in seconds. The default value is* `TimeEnd=10`, *i.e., an animation using the default values will last about 10 seconds. The number of frames set by* `Frames=n` *does not influence the time interval, but changes the number of frames displayed in this time interval.*

Here is a simple example:

```
plot(plot::Point2d([a, sin(a)], a = 0..2*PI,
                   Frames = 50, TimeRange = 0..5))
```

The point will be animated for about 5 physical seconds in which it moves along one period of the sine graph. Each frame is displayed for about 0.1 seconds. After increasing the number of frames by a factor of 2, each frame is displayed for about 0.05 seconds, making the animation somewhat smoother:

```
plot(plot::Point2d([a, sin(a)], a = 0..2*PI,
                   Frames = 100, TimeRange = 0..5))
```

Note that the human eye cannot distinguish between different frames if they change with a rate of more than 25 frames per second. Thus, the number of frames n set for the animation should satisfy

$$n < 25 \cdot (t_1 - t_0).$$

Hence, with the default time range `TimeBegin=0`, `TimeEnd=10` (seconds), it does not make sense to specify `Frames=n` with $n > 250$. If a higher frame number is required to obtain a sufficient resolution of the animated object, one should also increase the time for the animation by a sufficiently high value of `TimeEnd`.

What Can Be Animated?

We may regard a graphical primitive as a collection of plot attributes. (This is actually what they are internally. Indeed, even the function expression `sin(x)` in `plot::Function2d(sin(x), x=0..2*PI)` is realized as the attribute `Function=sin(x)`.) So, the question is:

"Which attributes can be animated?"

The answer is: *"Almost any attribute can be animated!"* Instead of listing the attributes that allow animation, it is much easier to characterize the attributes that cannot be animated:

- None of the canvas attributes can be animated. This includes layout parameters such as the physical size of the picture. See the help page of `plot::Canvas` for a complete list of all canvas attributes.
- None of the attributes of 2D scenes and 3D scenes can be animated. This includes layout parameters, background color and style, automatic camera positioning in 3D etc. See the help pages of `plot::Scene2d` and `plot::Scene3d` for a complete list of all scene attributes.

  Note that there are camera objects of type `plot::Camera` that can be placed in a 3D scene. These camera objects can be animated and allow to realize a "flight" through a 3D scene. See Section 11.16 for details.
- None of the attributes of 2D coordinate systems and 3D coordinate systems can be animated. This includes viewing boxes, axes, axes ticks, and grid

lines (rulings) in the background. See the help pages of `plot::CoordinateSystem2d` and `plot::CoordinateSystem3d` for a complete list of all attributes for coordinate systems.

Although the `ViewingBox` attribute of a coordinate system cannot be animated, the user can still achieve animated visibility effects in 3D by clipping box objects of type `plot::ClippingBox`.

- None of the attributes that are declared as "**Attribute Type: inherited**" on their help page can be animated. This includes size specifications such as `PointSize`, `LineWidth` etc.
- RGB and RGBa values cannot be animated. However, it is possible to animate the coloring of lines and surfaces via user defined procedures. See the help pages of `LineColorFunction` and `FillColorFunction` for details.
- The texts of annotations such as `Footer`, `Header`, `Title`, legend entries etc. cannot be animated. The position of titles, however, can be animated.

  There are special text objects `plot::Text2d` and `plot::Text3d` that allow to animate the text as well as their position.
- Fonts cannot be animated.
- Attributes such as `DiscontinuitySearch = TRUE` or `FillPattern = Solid` that can assume only finitely many values from a fixed discrete set cannot be animated.

Nearly all attributes not falling into one of these categories can be animated. You will find detailed information on this issue on the corresponding help pages of primitives and attributes.

Advanced Animations: The Synchronization Model

As already explained on page 11-76, there is a "real time clock" running in the background that synchronizes the animation of different animated objects.

Each animated object has its own separate "real time life span" set by the attributes `TimeBegin = t0`, `TimeEnd = t1` or, equivalently, `TimeRange = t0..t1`. The values `t0`, `t1` represent seconds measured by the "real time clock."

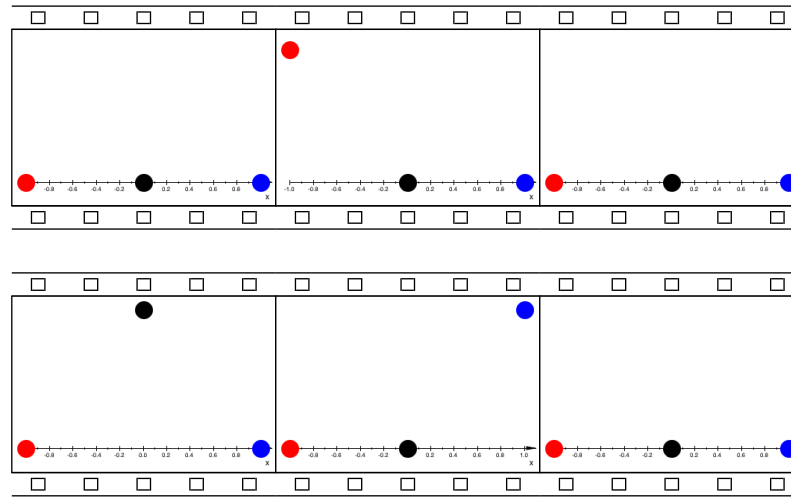
In most cases, there is no need to bother about specifying the life span. If `TimeBegin` and `TimeEnd` are not specified, the default values `TimeBegin=0` and `TimeEnd=10` are used, i.e., the animation will last about 10 seconds. These values only need to be modified

- if a shorter or longer real time period for the animation is desired, or
- if the animation contains several animated objects, where some of the animated objects are to remain static while others change.

Here is an example for the second situation. The plot consists of 3 jumping points. For the first 5 seconds, the left point jumps up and down, while the other points remain at their initial position. Then, all points stay static for 1 second. After a total of 6 seconds, the middle point starts its animation by jumping up and down, while the left point remains static in its final position and the right points stays static in its initial position. After 9 seconds, the right point begins to move as well. The overall time span for the animation is the hull of the time ranges of all animated objects, i.e., 15 seconds in this example:

```
p1 := plot::Point2d(-1, sin(a), a = 0..PI,
                    Color = RGB::Red,
                    TimeBegin = 0, TimeEnd = 5):
p2 := plot::Point2d(0, sin(a), a = 0..PI,
                    Color = RGB::Black,
                    TimeBegin = 6, TimeEnd = 12):
p3 := plot::Point2d(1, sin(a), a = 0..PI,
                    Color = RGB::Blue,
                    TimeBegin = 9, TimeEnd = 15):
```

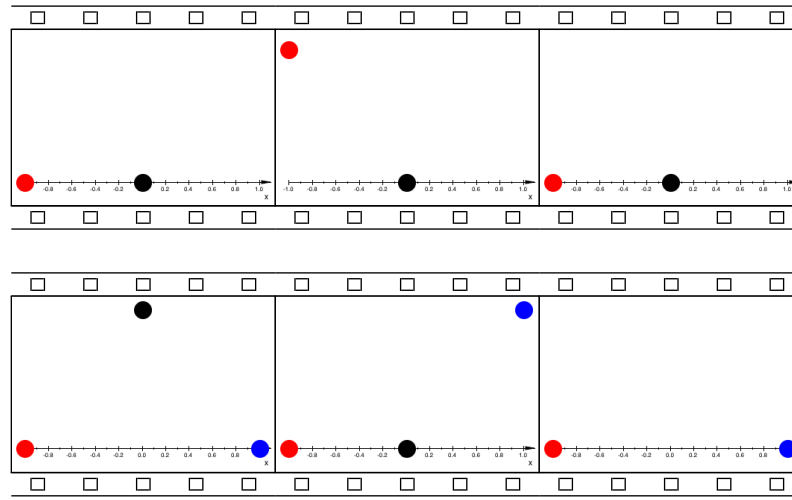
```
plot(p1, p2, p3, PointSize = 8.0*unit::mm,
     YAxisVisible = FALSE)
```



Here, all points use the default settings `VisibleBeforeBegin=TRUE` and `VisibleAfterEnd=TRUE` which make them visible as static objects outside the time range of their animation. We set `VisibleAfterEnd=FALSE` for the middle point, so that it disappears after the end of its animation. With `VisibleBeforeBegin=FALSE`, the right point is not visible until its animation starts:

```
p2::VisibleAfterEnd := FALSE:
p3::VisibleBeforeBegin := FALSE:
```

```
plot(p1, p2, p3, PointSize = 8.0*unit::mm,
     YAxisVisible = FALSE)
```



We summarize the synchronization model of animations:

> *The total real time span of an animated plot is the physical real time given by the minimum of the* `TimeBegin` *values of all animated objects in the plot to the maximum of the* `TimeEnd` *values of all the animated objects.*

- When a plot containing animated objects is created, the real time clock is set to the minimum of the `TimeBegin` values of all animated objects in the plot. The real time clock is started when pushing the 'play' button for animations in the graphical user interface.
- Before the real time reaches the `TimeBegin` value `t0` of an animated object, this object is static in the state corresponding to the begin of its animation. Depending on the attribute `VisibleBeforeBegin`, it may be visible or invisible before `t0`.
- During the time from `t0` to `t1`, the object changes from its original to its final state.

- After the real time reaches the `TimeEnd` value `t1`, the object stays static in the state corresponding to the end of its animation. Depending on the value of the attribute `VisibleAfterEnd`, it may stay visible or become invisible after `t1`.
- The animation of the entire plot ends with the physical time given by the maximum of the `TimeEnd` values of all animated objects in the plot.

Frame by Frame Animations

There are some special attributes such as `VisibleAfter` that are very useful to build animations from purely static objects:

> *With* `VisibleAfter=t0`*, an object is invisible from the start of the animation until time* `t0`*. Then it will appear and remain visible for the rest of the animation.*

> *With* `VisibleBefore=t1`*, an object is visible from the start of the animation until time* `t1`*. Then it will disappear and remain invisible for the rest of the animation.*

These attributes should not be combined to define a "visibility range" from `t0` to `t1`. Use the attribute `VisibleFromTo` instead:

> *With* `VisibleFromTo=t0..t1`*, an object is invisible from the start of the animation until time* `t0`*. Then it will appear and remain visible until time* `t1`*, when it will disappear and remain invisible for the rest of the animation.*

We continue the example of the previous section in which we defined the following animated points:

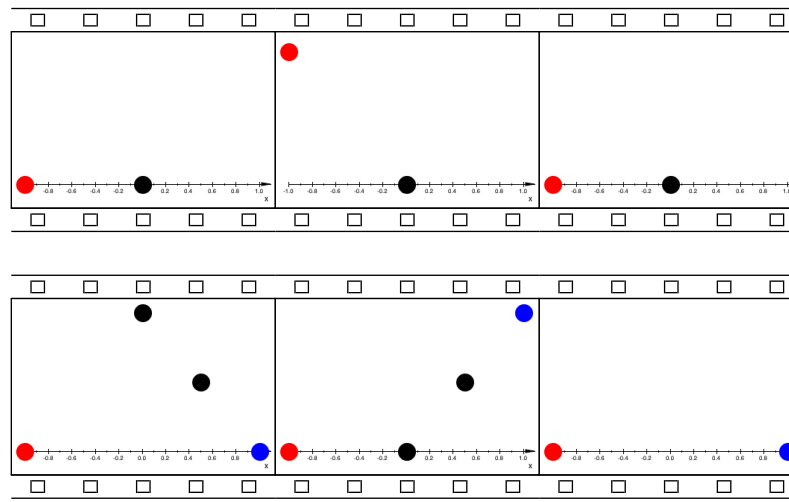
```
p1 := plot::Point2d(-1, sin(a), a = 0..PI,
                    Color = RGB::Red,
                    TimeBegin = 0, TimeEnd = 5):
p2 := plot::Point2d(0, sin(a), a = 0..PI,
                    Color = RGB::Black,
                    TimeBegin = 6, TimeEnd = 12):
p3 := plot::Point2d(1, sin(a), a = 0..PI,
                    Color = RGB::Blue,
                    TimeBegin = 9, TimeEnd = 15):
p2::VisibleAfterEnd := FALSE:
p3::VisibleBeforeBegin := FALSE:
```

We add a further point p4 that is not animated. We make it invisible at the start of the animation via the attribute `VisibleFromTo`. It is made visible after 7 seconds to disappear again after 13 seconds:

```
p4 := plot::Point2d(0.5, 0.5, Color = RGB::Black,
                    VisibleFromTo = 7..13):
```

The start of the animation is determined by `p1` which bears the attribute `TimeBegin=0`, the end of the animation is determined by `p3` which has set `TimeEnd=15`:

```
plot(p1, p2, p3, p4, PointSize = 8.0*unit::mm,
     YAxisVisible = FALSE)
```



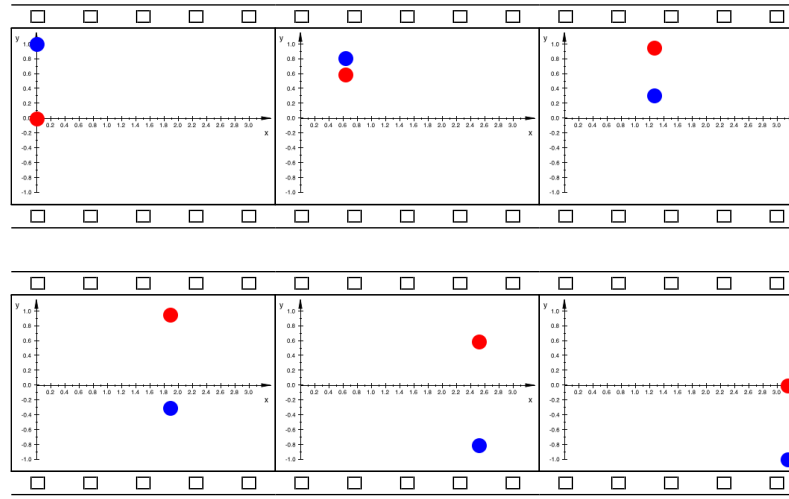
Although a typical MuPAD animation is generated object by object, each animated object taking care of its own animation, we can also use the attributes `VisibleAfter`, `VisibleBefore`, `VisibleFromTo` to build up an animation frame by frame:

> **"Frame by frame animations:"** *Choose a collection of (typically static) graphical primitives that are to be visible in the i-th frame of the animation. Set* `VisibleFromTo=t[i]..t[i+1]` *for these primitives, where* `t[i]..t[i+1]` *is the real time life span of the i-th frame (in seconds). Finally, plot all frames in a single* `plot` *command.*

Here is an example. We let two points wander along the graphs of the sine and the cosine function, respectively. Each frame is to consist of a picture of two points. We use `plot::Group2d` to define the frame; the group forwards the attribute `VisibleFromTo` to all its elements:

```
for i from 0 to 101 do
    t[i] := i/10;
end_for:
for i from 0 to 100 do
  x := i/100*PI;
  myframe[i] := plot::Group2d(
        plot::Point2d([x, sin(x)], Color = RGB::Red),
        plot::Point2d([x, cos(x)], Color = RGB::Blue),
        VisibleFromTo = t[i]..t[i + 1]);
end_for:
```

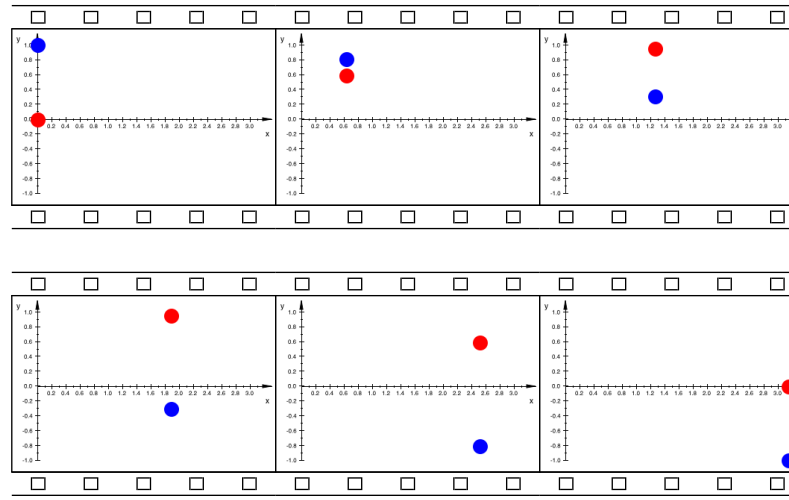
```
plot(myframe[i] $ i = 0..100, PointSize = 7.0*unit::mm)
```



This “frame by frame” animation certainly needs a little bit more coding effort than the equivalent object-wise animation, where each of the points is animated.

Compare:

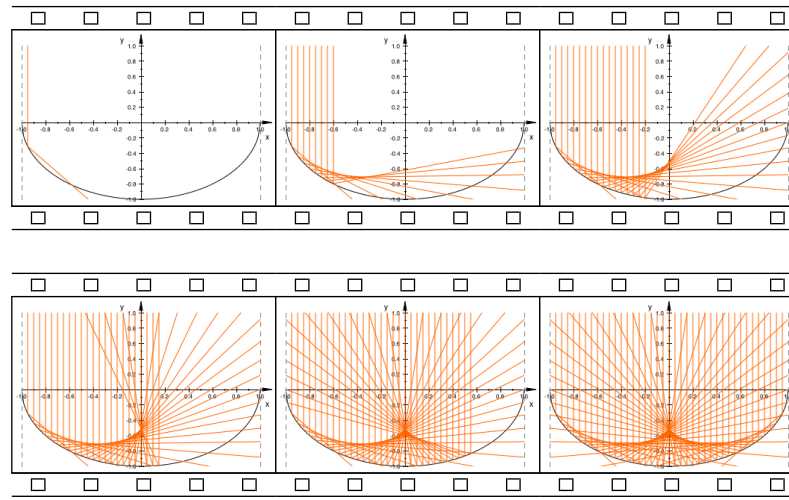
```
delete i:
plot(plot::Point2d([i/100*PI, sin(i/100*PI)],
                   i = 0..100, Color = RGB::Red),
     plot::Point2d([i/100*PI, cos(i/100*PI)],
                   i = 0..100, Color = RGB::Blue),
     Frames = 101, PointSize = 7.0*unit::mm)
```



There is a special kind of plot where “frame by frame” animations are very useful. Note that in the present version of the graphics, new plot objects cannot be added to a scene that is already rendered. With the special “visibility” animations for static objects, however, one can easily simulate a plot that gradually builds up: fill the frames of the animation with static objects that are visible for a limited time only. The visibility can be chosen very flexibly by the user. For example, the static objects can be made visible only for one frame (`VisibleFromTo`) which allows to simulate moving objects.

In the following example, we use `VisibleAfter` to fill up the plot gradually. We demonstrate the caustics generated by sunlight in a tea cup. The rim of the cup, regarded as a mirror, is given by the function $f(x) = -\sqrt{1-x^2}$, $x \in [-1, 1]$ (a semi-circle). Sun rays parallel to the y-axis are reflected by the rim. After reflection at the point $(x, f(x))$ of the rim, a ray heads into the direction $(-1, -(f'(x) - 1/f'(x))/2)$ if x is positive. It heads into the direction $(1, (f'(x) - 1/f'(x))/2)$ if x is negative. Sweeping through the mirror from left to right, the incoming rays as well as the reflected rays are visualized as lines. In the animation, they become visible after the time $5\,x$, where x is the coordinate of the rim point at which the ray is reflected:

```
f := x -> -sqrt(1 - x^2):
plot(// The static rim:
     plot::Function2d(f(x), x = -1..1,
                      Color = RGB::Black),
     // The incoming rays:
     plot::Line2d([x, 2], [x, f(x)],
                  Color = RGB::Orange,
                  VisibleAfter = 5*x
                 ) $ x in [-1 + i/20 $ i = 1..39],
     // The reflected rays leaving to the right:
     plot::Line2d([x, f(x)],
                  [1, f(x)+(1-x)*(f'(x) - 1/f'(x))/2],
                  Color = RGB::Orange,
                  VisibleAfter = 5*x
                 ) $ x in [-1 + i/20 $ i =  1..19],
     // The reflected rays leaving to the left:
     plot::Line2d([x, f(x)],
                  [-1, f(x) - (x+1)*(f'(x) - 1/f'(x))/2],
                  Color = RGB::Orange,
                  VisibleAfter = 5*x
                 ) $ x in [-1 + i/20 $ i = 21..39],
     ViewingBox = [-1..1, -1..1])
```

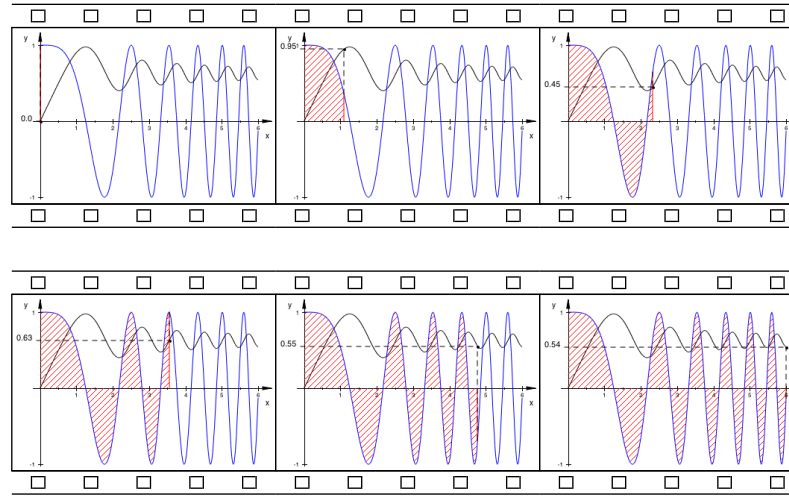


Examples

Example 1: We build a 2D animation that displays a function $f(x)$ together with the integral $F(x) = \int_0^x f(y)\,\mathrm{d}y$. The area between the graph of f and the x-axis is displayed as an animated hatch object. The current value of $F(x)$ is displayed by an animated text:

```
DIGITS := 2:
// the function, f(x):
f := x -> cos(x^2):
// the anti-derivative, F(x):
F := x -> numeric::int(f(y), y = 0..x):
// the graph of f(x):
g := plot::Function2d(f(x), x = 0..6,
                      Color = RGB::Blue):
// the graph of F(x):
G := plot::Function2d(F(x), x = 0..6,
                      Color = RGB::Black):
// a point moving along the graph of F(x):
p := plot::Point2d([a, F(a)], a = 0..6,
                   Color = RGB::Black):
// hatched region between the origin and
// the moving point p:
h := plot::Hatch(g, 0, 0..a, a = 0..6,
                 Color = RGB::Red):
// the right border line of the hatched region:
l := plot::Line2d([a, 0], [a, f(a)], a = 0..6,
                  Color = RGB::Red):
// a dashed vertical line from f(x) to F(x):
L1 := plot::Line2d([a, f(a)], [a, F(a)],
                   a = 0..6, Color = RGB::Black,
                   LineStyle = Dashed):
// a dashed horizontal line from the y axis to F(x):
L2 := plot::Line2d([-0.1, F(a)], [a, F(a)],
                   a = 0..6, Color = RGB::Black,
                   LineStyle = Dashed):
// the current value of F at the moving point p:
t := plot::Text2d(a -> F(a), [-0.2, F(a)], a = 0..6,
                  HorizontalAlignment = Right):
```

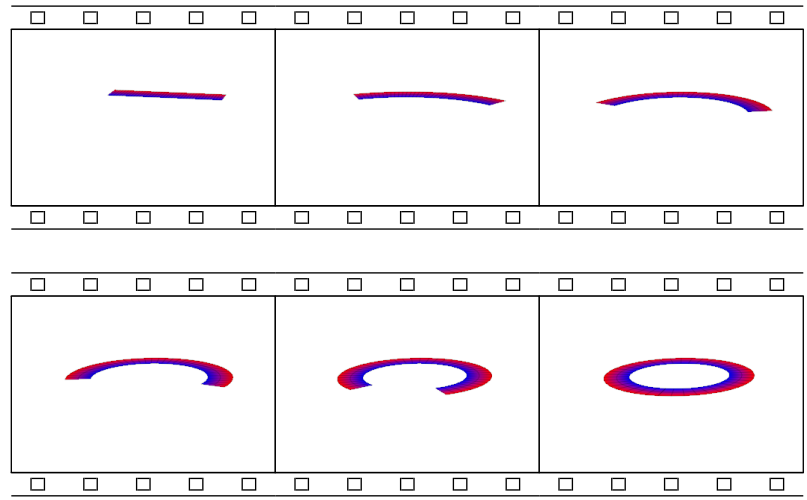
```
plot(g, G, p, h, l, L1, L2, t,
     YTicksNumber = None, YTicksAt = [-1, 1]):
delete DIGITS:
```



Example 2: We build two 3D animations. The first one starts with a rectangular strip that is deformed to an annulus in the x, y plane:

```
c := a -> 1/2 *(1 - 1/sin(PI/2*a)):
mycolor := (u, v, x, y, z) ->
                [(u - 0.8)/0.4, 0, (1.2 - u)/0.4]:
rectangle2annulus := plot::Surface(
   [c(a) + (u-c(a))*cos(PI*v), (u-c(a))*sin(PI*v), 0],
   u = 0.8..1.2, v = -a..a, a = 1/10^10..1,
   FillColorFunction = mycolor, Mesh = [3, 40],
   Frames = 40):
```

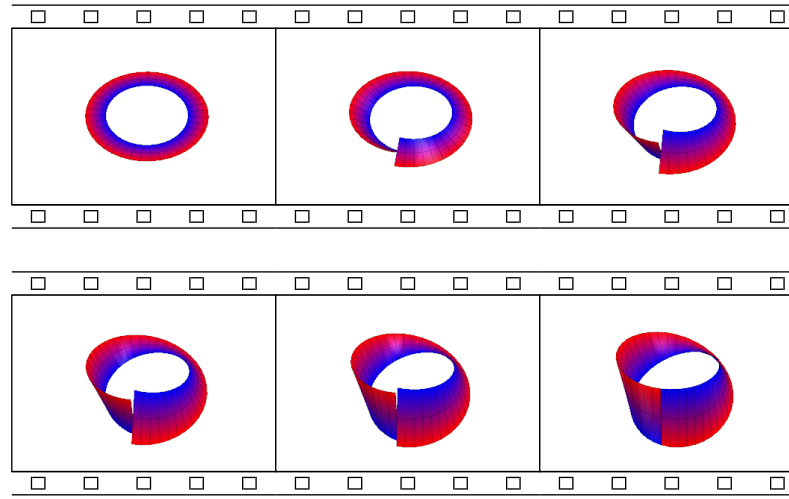
```
plot(rectangle2annulus, Axes = None,
     CameraDirection = [-11, -3, 3])
```



The second animation twists the annulus to become a Moebius strip:

```
annulus2moebius := plot::Surface(
   [((u - 1)*cos(a*v*PI/2) + 1)*cos(PI*v),
    ((u - 1)*cos(a*v*PI/2) + 1)*sin(PI*v),
     (u - 1)*sin(a*v*PI/2)],
   u = 0.8..1.2, v = -1..1, a = 0..1, Mesh = [3, 40],
   FillColorFunction = mycolor, Frames = 20):
```

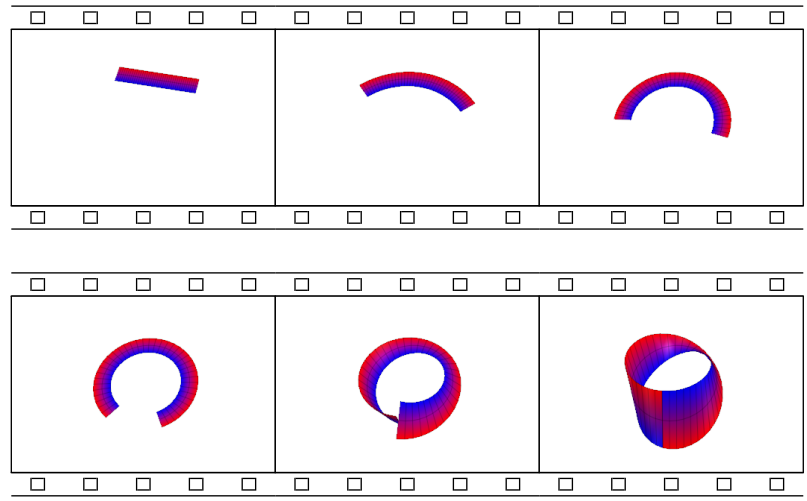
```
plot(annulus2moebius, Axes = None,
     CameraDirection = [-11, -3, 3])
```



Note that the final frame of the first animation coincides with the first frame of the second animation. To join the two separate animations, we can set appropriate visibility ranges and plot them together. After 5 seconds, the first animation object vanishes and the second takes over:

```
rectangle2annulus::VisibleFromTo := 0..5:
annulus2moebius::VisibleFromTo := 5..7:
```

```
plot(rectangle2annulus, annulus2moebius,
     Axes = None, CameraDirection = [-11, -3, 3])
```



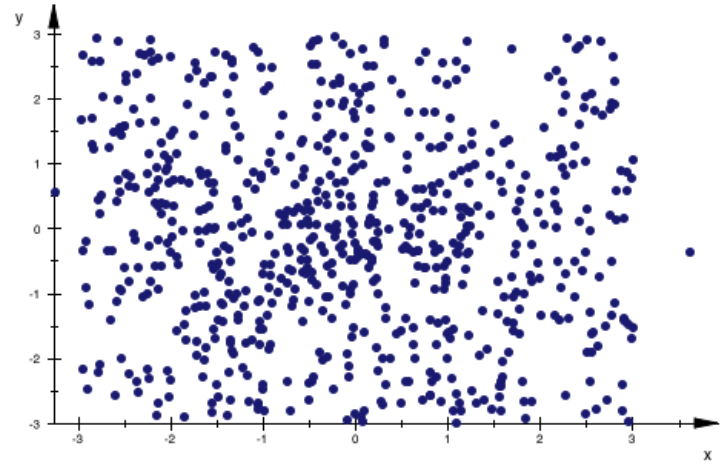
Groups of Primitives

An arbitrary number of graphical primitives in 2D or 3D can be collected in groups of type `plot::Group2d` or `plot::Group3d`, respectively. This is useful for handing down attribute values to all elements in a group.

In the following example, we visualize random generators with different distributions by using them to position random points:

```
r1 := stats::normalRandom(0, 1):
group1 := plot::Group2d(plot::Point2d(r1(), r1())
                                          $ i = 1..200):
r2 := stats::uniformRandom(-3, 3):
group2 := plot::Group2d(plot::Point2d(r2(), r2())
                                          $ i = 1..500):
```

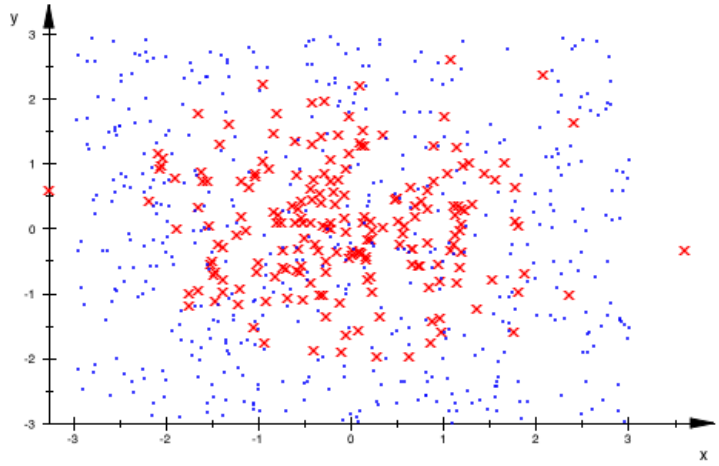
```
plot(group1, group2, Axes = Frame)
```



We cannot distinguish between the two kinds of points. Due to the grouping, it is very easy to change their color, size, and style by setting the appropriate attributes in the groups. Now, the two kinds of points can be distinguished easily:

```
group1::PointColor := RGB::Red:
group1::PointStyle := XCrosses:
group2::PointColor := RGB::Blue:
group2::PointSize := 0.7*unit::mm:
```

```
plot(group1, group2, Axes = Frame)
```



Transformations

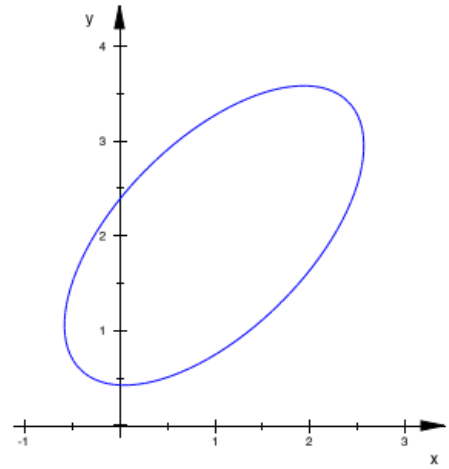
Affine linear transformations $\vec{x} \to A \cdot \vec{x} + \vec{b}$ with a vector $\vec{b}$ and a matrix A can be applied to graphical objects via transformation objects. There are special transformations such as translations, scaling, and rotations as well as general affine linear transformations:

- `plot::Translate2d([b1, b2], Primitive1, Primitive2, ...)` applies the translation $\vec{x} \to \vec{x} + \vec{b}$ by the vector $\vec{b} = (b_1, b_2)^T$ to all points of 2D primitives.
- `plot::Translate3d([b1, b2, b3], Primitive1, ...)` applies the translation $\vec{x} \to \vec{x} + \vec{b}$ by the vector $b = (b_1, b_2, b_3)^T$ to all points of 3D primitives.
- `plot::Reflect2d([x1, y1], [x2, y2], Primitive1, ...)` reflects all 2D primitives about the line through the points `[x1, y1]` and `[x2, y2]`.
- `plot::Reflect3d([x, y, z], [nx, ny, nz], Primitive1, ...)` reflects all 3D primitives about the plane through the point `[x, y, z]` with the normal `[nx, ny, nz]`.
- `plot::Rotate2d(angle, [c1, c2], Primitive1, ...)` rotates all points of 2D primitives counter clockwise by the given angle about the pivot point $(c_1, c_2)^T$.
- `plot::Rotate3d(angle, [c1, c2, c3], [d1, d2, d3], Primitive1, ...)` rotates all points of 3D primitives by the given angle around the rotation axis specified by the pivot point $(c_1, c_2, c_3)^T$ and the direction $(d_1, d_2, d_3)^T$.
- `plot::Scale2d([s1, s2], Primitive1, ...)` applies the diagonal scaling matrix $\mathrm{diag}(s_1, s_2)$ to all points of 2D primitives.
- `plot::Scale3d([s1, s2, s3], Primitive1, ...)` applies the diagonal scaling matrix $\mathrm{diag}(s_1, s_2, s_3)$ to all points of 3D primitives.
- `plot::Transform2d([b1, b2], A, Primitive1, ...)` applies the general affine linear transformation $\vec{x} \to A \cdot \vec{x} + \vec{b}$ with a 2×2 matrix A and a vector $\vec{b} = (b_1, b_2)^T$ to all points of 2D primitives.
- `plot::Transform3d([b1, b2, b3], A, Primitive1, ...)` applies the general affine linear transformation $\vec{x} \to A \cdot \vec{x} + \vec{b}$ with a 3×3 matrix A and a vector $\vec{b} = (b_1, b_2, b_3)^T$ to all points of 3D primitives.

The ellipses `plot::Ellipse2d` provided by the `plot` library have axes parallel to the coordinate axes. We use a rotation to create an ellipse with a different orientation:

```
center := [1, 2]:
ellipse := plot::Ellipse2d(2, 1, center):
```

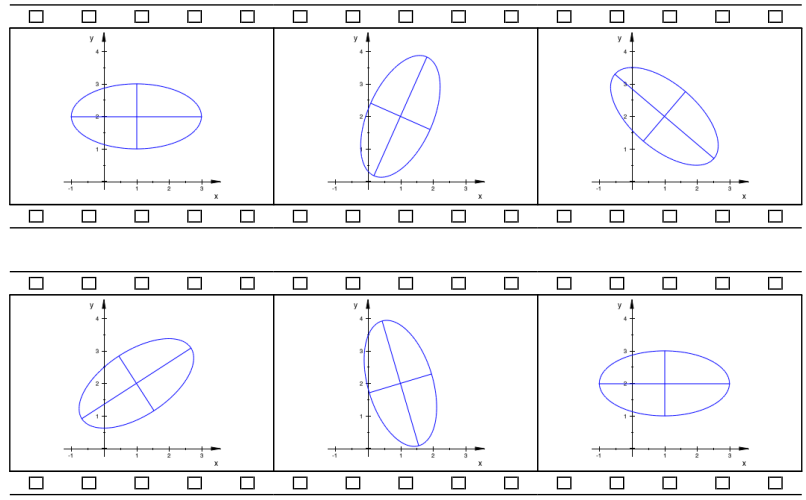
```
plot(plot::Rotate2d(PI/4, center, ellipse))
```



Transform objects can be animated. We build a group consisting of the ellipse and its symmetry axes. An animated rotation is applied to the group:

```
g := plot::Group2d(ellipse,
          plot::Line2d([center[1] + 2, center[2]],
                       [center[1] - 2, center[2]]),
          plot::Line2d([center[1], center[2] + 1],
                       [center[1], center[2] - 1])):
```

```
plot(plot::Rotate2d(a, center, a = 0..2*PI, g))
```

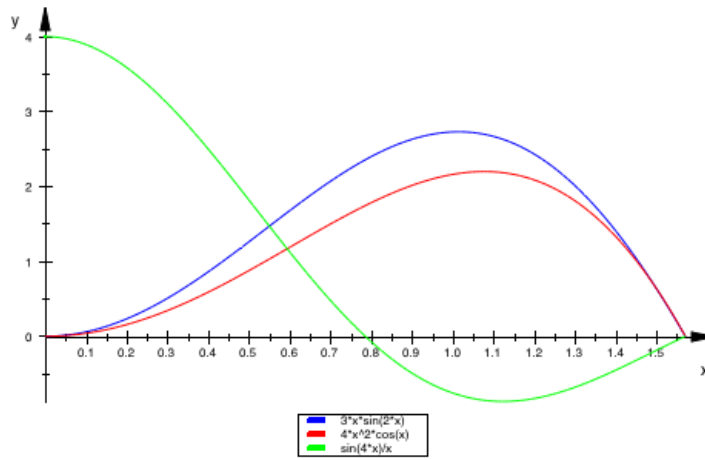


Legends

The annotations of a MuPAD® plot may include a legend. A legend is a small table that relates the color of an object with some text explaining the object:

```
f := 3*x*sin(2*x):
g := 4*x^2*cos(x):
h := sin(4*x)/x:
```

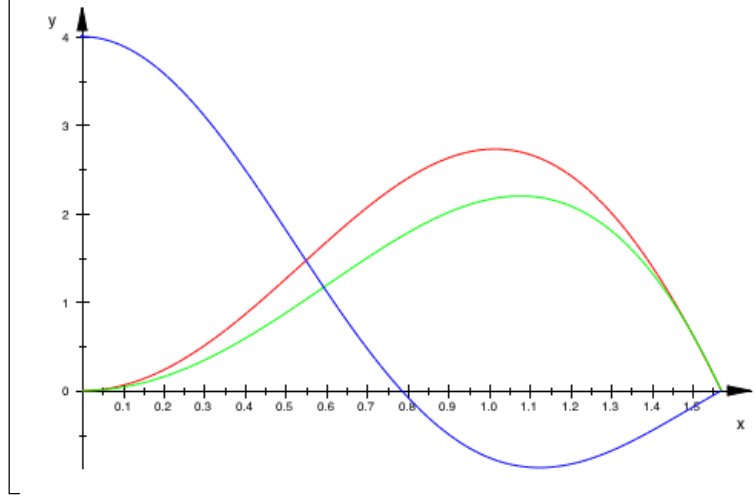
```
plotfunc2d(f, g, h, x = 0..PI/2):
```



By default, legends are provided only by `plotfunc2d` and `plotfunc3d`. These routines define the legend texts as the expressions with which the functions are passed to `plotfunc2d` or `plotfunc3d`, respectively.

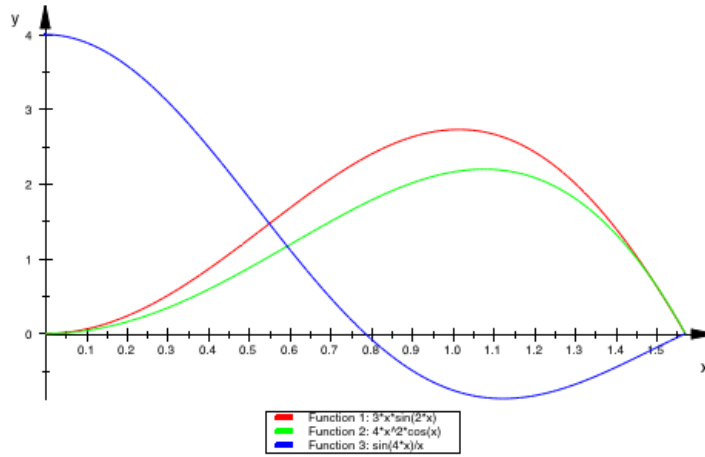
A corresponding plot command using primitives of the `plot` library does not generate the legend automatically:

```
plot(plot::Function2d(f, x = 0..PI/2, Color = RGB::Red),
     plot::Function2d(g, x = 0..PI/2, Color = RGB::Green),
     plot::Function2d(h, x = 0..PI/2, Color = RGB::Blue))
```



However, legends can be requested explicitly:

```
plot(plot::Function2d(f, x = 0..PI/2, Color = RGB::Red,
                      Legend = "Function 1: ".expr2text(f)),
     plot::Function2d(g, x = 0..PI/2, Color = RGB::Green,
                      Legend = "Function 2: ".expr2text(g)),
     plot::Function2d(h, x = 0..PI/2, Color = RGB::Blue,
                      Legend = "Function 3: ".expr2text(h)))
```



Each graphical primitive accepts the attribute `Legend`. Passing this attribute to an object triggers several actions:

- The object attribute `LegendText` is set to the given string.
- The object attribute `LegendEntry` is set to `TRUE`.
- A hint is sent to the scene containing the object advising it to use the scene attribute `LegendVisible = TRUE`.

The attributes `LegendText` and `LegendEntry` are visible in the “object inspector” of the interactive viewer (page 11-50) and can be manipulated interactively for each single primitive after selection in the “object browser.” The attribute `LegendVisible` is associated with the scene object accessible via the “object browser.”

At most 20 entries can be displayed in a legend. If more entries are specified in a plot command, the surplus entries are ignored. Further, the legend may not cover more than 50% of the height of the drawing area of a scene. Only those legend entries fitting into this space are displayed; remaining entries are ignored.

If the attribute `LegendEntry = TRUE` is set for a primitive, its legend entry is determined as follows:

- If the attribute `LegendText` is specified, its value is used for the legend text.
- If no `LegendText` is specified, but the `Name` attribute is set, the name is used for the legend text.
- If no `Name` attribute is specified either, the type of the object such as `Function2d`, `Curve2d` etc. is used for the legend text.

Here are all attributes relevant for legends:

attribute name	possible values	meaning	default
`Legend`	string	sets `LegendText` to the given string, `LegendEntry` to `TRUE`, and `LegendVisible` to `TRUE`.	
`LegendEntry`	`TRUE`, `FALSE`	add this object to the legend?	`TRUE` for function graphs, curves, and surfaces, `FALSE` otherwise
`LegendText`	string	legend text	
`LegendVisible`	`TRUE`, `FALSE`	legend on/off	`TRUE` for `plotfunc2d/3d` plotting more than one function, `FALSE` otherwise
`LegendPlacement`	`Top`, `Bottom`	vertical placement	`Bottom`
`LegendAlignment`	`Left`, `Center`, `Right`	horizontal alignment	`Center`
`LegendFont`	see page 11-104	font for the legend text	Sans-Serif 8

Table 11.10: Attributes for the legend

Fonts

The plot attributes for specifying fonts are `AxesTitleFont`, `FooterFont`, `HeaderFont`, `LegendFont`, `TextFont`, `TicksLabelFont`, and `TitleFont`. Each such font is specified by a MuPAD® list containing any of the following:

- A string denoting the font family: the available font families depend on the fonts that are installed on your machine. Typical font families available on Windows systems are `"Times New Roman"` (of type `"serif"`), `"Arial"` (of type `"sans-serif"`), or `"Courier New"` (of type `"monospace"`).

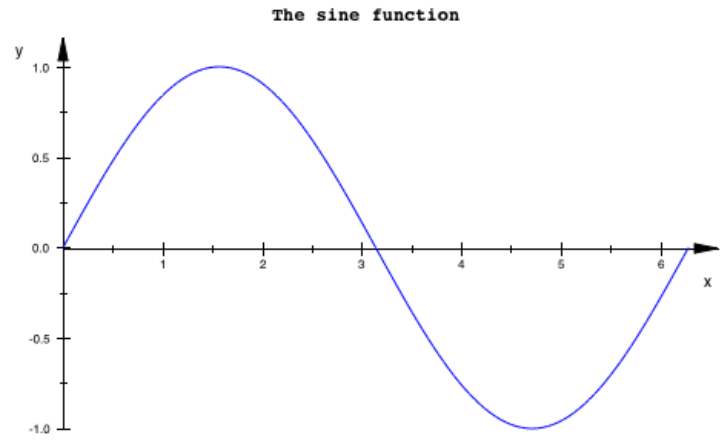
  To find out which fonts are available on your machine, open the menu 'Format,' submenu 'Font' in your MuPAD notebook. The first column in the font dialog provides the names of the font families that you may specify.

  The most portable way of defining fonts is to use one the three generic family names `"serif"`, `"sans-serif"`, or `"monospace"`. The system will automatically choose one of the available font families of the specified type for you.

- A positive integer specifying the size of the font in points.
- When the parameter `Bold` is specified, the font is bold.
- When the parameter `Italic` is specified, the font is italic.
- A color specification.
- An alignment specification: it must be one of the flags `Left`, `Center`, or `Right`.

In the following example, we specify the font for the canvas header and footer:

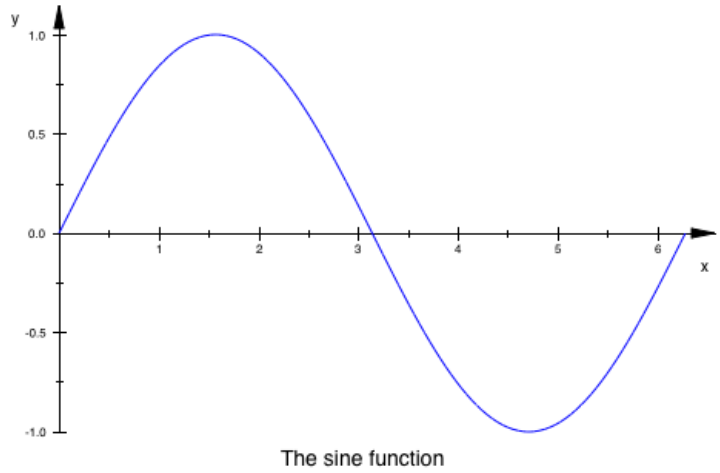
```
plot(plot::Function2d(sin(x), x = 0..2*PI),
     Header = "The sine function",
     HeaderFont = ["monospace", 12, Bold],
     Footer = "The sine function",
     FooterFont = ["Times New Roman", 14, Italic])
```



The sine function

All font parameters are optional; some default values are chosen for entries that are not specified. For example, if you do not care about the footer font family for your plot, but you insist on a specific font size, you may specify a 14 pt font as follows:

```
plot(plot::Function2d(sin(x), x = 0..2*PI),
     Footer = "The sine function", FooterFont = [14])
```



Saving and Exporting Pictures

Interactive Saving and Exporting

The MuPAD® kernel uses an xml format to communicate with the renderer. Usually, a `plot` command sends a stream of xml data directly to the viewer which renders the picture.

After clicking on the picture, the notebook provides a menu item 'File/Export Graphics...' that opens a dialog allowing to save the picture in a variety of graphical formats:

- The image may also be stored in various standard bitmap formats such as png, jpg etc.
- Graphics can be exported in eps (encapsulated postscript) format. Since eps does not support transparency, however, graphics making use of transparency will degrade in quality in this step. This includes most 3D plots.
- MuPAD's native format is indicated by the file extension 'xvz' for the compressed and 'xvc' for the uncompressed version.

  One can use MuPAD to open such files and display and manipulate the plots contained therein.
- Further, there is jvx export (JavaView).
- MuPAD animations can be saved as animated gif files.
- On Windows® and Macintosh® systems, MuPAD animations can also be saved as avi files.
- In a Mac OS® system, MuPAD animations can furthermore be saved as Quicktime® movie files.

Batch Mode

MuPAD plots can also be saved in "batch mode" by specifying the attribute `OutputFile = filename` in a `plot` call:

```
plot(Primitive1, Primitive2, ...,
     OutputFile = "mypicture.xvz")
```

Here, the extension `xvz` of the file name indicates that MuPAD's xml data are to be written in compressed format. Alternatively, the extension `xvc` may be used to write the xml data without compression of the file (the resulting ascii file can be read with any text editor). Files in both formats can be opened by MuPAD to generate the plot encoded by the xml data.

If the MuPAD environment variable `WRITEPATH` does not have a value, the previous call creates the file in the directory where MuPAD is installed. An absolute pathname can be specified to place the file anywhere else:

```
plot(Primitive1, Primitive2, ...,
     OutputFile = "C:\\Documents\\mypicture.xvz")
```

Alternatively, the environment variable `WRITEPATH` can be set:

```
WRITEPATH := "C:\\Documents":
plot(Primitive1, Primitive2, ...,
     OutputFile = "mypicture.xvz")
```

Now, the plot data are stored in the file '`C:\Documents\mypicture.xvz`'.

Apart from saving files as xml data, MuPAD pictures can also be saved in a variety of standard graphical formats such as jpg, eps, svg, bmp etc. In batch mode, the export is triggered by the `OutputFile` attribute in the same way as for saving in xml format. Just use an appropriate extension of the filename indicating the format. The following commands save the plot in four different files in jpg, eps, svg, and bmp format, respectively:

```
plot(Primitive1, ..., OutputFile = "mypicture.jpg"):
plot(Primitive1, ..., OutputFile = "mypicture.eps"):
plot(Primitive1, ..., OutputFile = "mypicture.svg"):
plot(Primitive1, ..., OutputFile = "mypicture.bmp"):
```

An animated MuPAD plot can be exported to avi format:

```
plot(plot::Function2d(exp(a*x), x = 0..1, a = -1..1)
     OutputFile = "myanimation.avi")
```

If no file extension is specified by the file name, the default extension `xvc` is used, i.e., uncompressed xml data are written.

If a notebook is saved to a file, its location in the file system is available inside the notebook as the value of the environment variable `NOTEBOOKPATH`. If you wish to save your plot in the same folder as the notebook, you may call

```
plot(Primitive1, Primitive2, ...,
     OutputFile = NOTEBOOKPATH."mypicture.xvz")
```

In addition to `OutputFile`, there is the attribute `OutputOptions` to specify parameters for some of the export formats. The help page of this attribute provides detailed information.

Importing Pictures

MuPAD® does not provide for many tools to import standard graphical *vector* formats, yet. Presently, the only supported vector type is the stl format, popular in stereo lithography, which encodes 3D surfaces. It can be imported via the routine `plot::SurfaceSTL`.

In contrast to graphical *vector* formats, there are many standard *bitmap* formats such as bmp, gif, jpg, ppm etc. that can be imported. One can read such a file via `import::readbitmap`, thus creating a MuPAD array of RGB color values or an equivalent three-dimensional hardware float array that can be manipulated at will. In particular, it can be fed into the routine `plot::Raster` which creates an object that can be used in any 2D MuPAD plot. Note, however, that the import of bitmap data consumes a lot of memory, i.e., only reasonably small bitmaps (up to a few hundred pixels in each direction) should be processed. Memory consumption is much smaller with hardware floating point arrays, which is why `import::reasonably` uses them by default.

In the following example, we plot the probability density function and the cumulative density function of the standard normal ("Gaussian") distribution. Paying tribute to Carl Friedrich Gauss, we wish to display his picture in this plot. Assume that we have his picture as a png bitmap file Gauss.png. We import the file via `import::readbitmap` that provides us with the width and height in pixels and the color data:

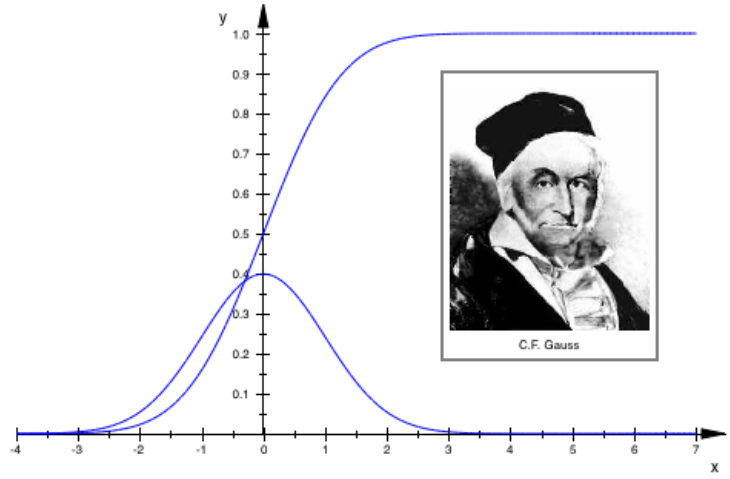
```
[width, height, gauss] := import::readbitmap("Gauss.png"):
```

Note the colon at the end of the above command! Without it, MuPAD will print the color information on the screen, and formatting such a huge output takes time.

We have to use `Scaling=Constrained` to preserve the aspect ratio of the image. Unfortunately, this setting is not appropriate for the function plots. So we use two different scenes that are positioned via `Layout=Relative`. See Section 'Layout of Canvas and Scenes' of the online `plot` documentation for details on how the layout of a canvas containing several scenes is set.

```
pdf := stats::normalPDF(0, 1):
cdf := stats::normalCDF(0, 1):
```

```
plot(plot::Scene2d(plot::Function2d(pdf(x), x = -4..7),
                   plot::Function2d(cdf(x), x = -4..7),
                   Width = 1, Height = 1),
     plot::Scene2d(plot::Raster(gauss),
                   Scaling = Constrained,
                   Width = 0.3, Height = 0.6,
                   Bottom = 0.25, Left = 0.6,
                   BorderWidth = 0.5*unit::mm,
                   Footer = "C.F. Gauss",
                   FooterFont = [8]),
     Layout = Relative)
```



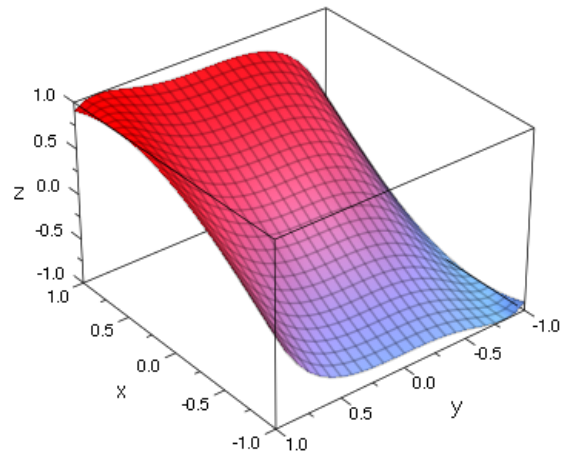
Cameras in 3D

The MuPAD® 3D graphics model includes an observer at a specific position, pointing a camera with a lens of a specific opening angle to some specific focal point. The specific parameters “position,” “angle,” and “focal point” determine the picture that the camera will take.

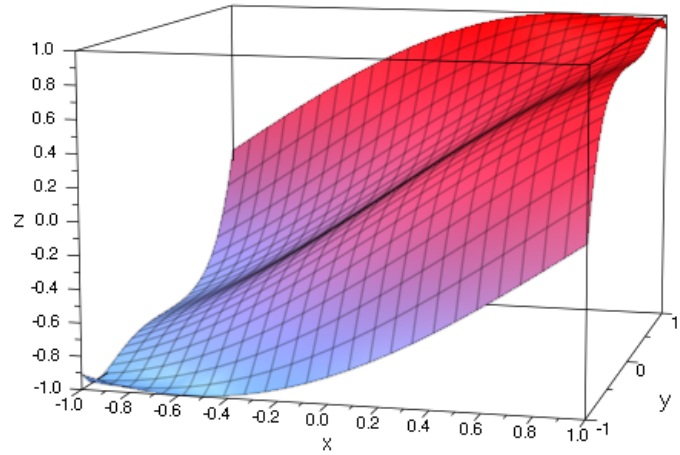
When a 3D picture is created,a camera with an appropriate default lens is positioned automatically. Its focal point is chosen as the center of the graphical scene. The interactive viewer allows to rotate the scene which, in fact, is implemented internally as a change of the camera position. Also interactive zooming in and zooming out is realized by moving the camera closer to or farther away from the scene.

Apart from interactive camera motions, the perspective of a 3D picture can also be set in the calls generating the plot. One way is to specify the direction from which the camera is pointing towards the scene. This is done via the attribute `CameraDirection`:

```
plot(plot::Function3d(sin(x + y^3), x = -1..1, y = -1..1),
     CameraDirection = [-25, 20, 30])
```



```
plot(plot::Function3d(sin(x + y^3), x = -1..1, y = -1..1),
     CameraDirection = [10, -40, 10])
```



In these calls, `CameraDirection` does not fully specify the position of the camera. This attribute just requests the camera to be placed at some large distance from the scene along the ray in the direction given by the attribute. The actual distance from the scene is determined automatically to let the graphical scene fill the picture optimally.

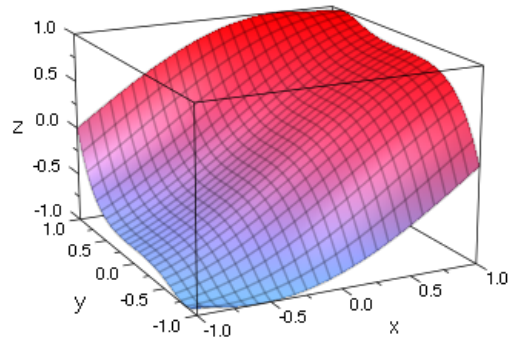
For a full specification of the perspective, there are camera objects of type `plot::Camera` that allow to specify the position of the camera, its focal point and the opening angle of its lens:

```
position := [-5, -10, 5]:
focalpoint := [0, 0, 0]:
angle := PI/12:
camera := plot::Camera(position, focalpoint, angle):
```

This camera can be passed like any graphical object to the `plot` command generating the scene.

Once a camera object is specified in a graphical scene, it determines the view. No "automatic camera" is used any more:

```
plot(plot::Function3d(sin(x + y^3), x = -1..1, y = -1..1),
     camera)
```

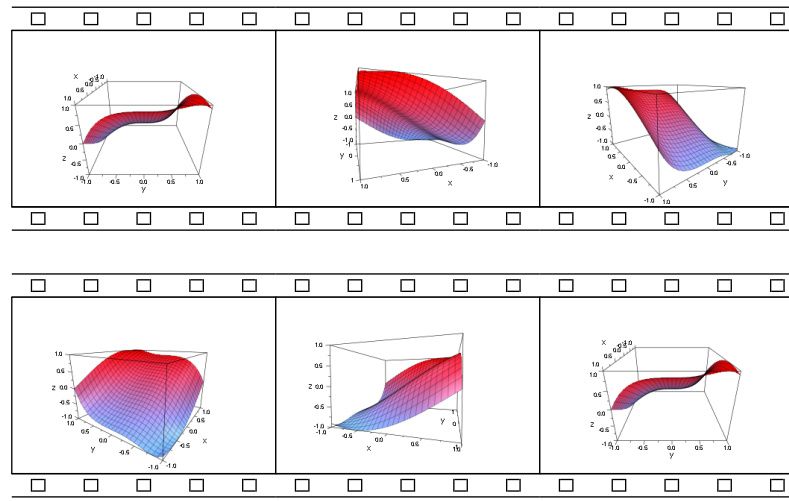


Camera objects can be animated:

```
camera := plot::Camera([3*cos(a), 3*sin(a), 1 + cos(2*a)],
                       [0, 0, 0], PI/3, a = 0..2*PI,
                       Frames = 100):
```

Inserting the animated camera in a graphical scene, we obtain an animated plot simulating a “flight around the scene:”

```
plot(plot::Function3d(sin(x + y^3), x = -1..1, y = -1..1),
     camera)
```



In fact, several cameras can be installed simultaneously in a scene. Per default, the first camera produces the view rendered. After clicking on another camera in the object browser of the viewer (page 11-50), the selected camera takes over and the new view is shown.

Next, we have a look at a more appealing example: the so-called "Lorenz attractor." The Lorenz ODE is the system

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} p\cdot(y-x) \\ -x\cdot z + r\cdot x - y \\ x\cdot y - b\cdot z \end{pmatrix}$$

with fixed parameters p, r, b. As a dynamical system for $Y = [x, y, z]$, we have to solve the ODE $\mathrm{d}Y/\mathrm{d}t = f(t, Y)$ with the following vector field:

```
f := proc(t, Y)
     local x, y, z;
     begin
        [x, y, z] := Y:
        [p*(y - x), -x*z + r*x - y, x*y - b*z]
     end_proc:
```

Consider the following parameters and the following initial condition `Y0`:

```
p := 10: r := 28: b := 1: Y0 := [1, 1, 1]:
```

The routine `plot::Ode3d` serves for generating a graphical 3D solution of a dynamical system. It solves the ODE numerically and generates graphical data from the numerical mesh. The plot data are specified by the user via "generators" (procedures) that map a solution point (t, Y) to a point (x, y, z) in 3D.

The following generator `Gxyz` produces a 3D phase plot of the solution. The generator `Gyz` projects the solution curve to the (y, z) plane with $x = -20$; the generator `Gxz` projects the solution curve to the (x, z) plane with $y = -20$; the generator `Gxy` projects the solution curve to the (x, y) plane with $z = 0$:

```
Gxyz := (t, Y) -> Y:
Gyz := (t, Y) -> [-20, Y[2], Y[3]]:
Gxz := (t, Y) -> [Y[1], -20, Y[3]]:
Gxy := (t, Y) -> [Y[1], Y[2], 0]:
```

With these generators, we create a 3D plot object consisting of the phase curve and its projections:

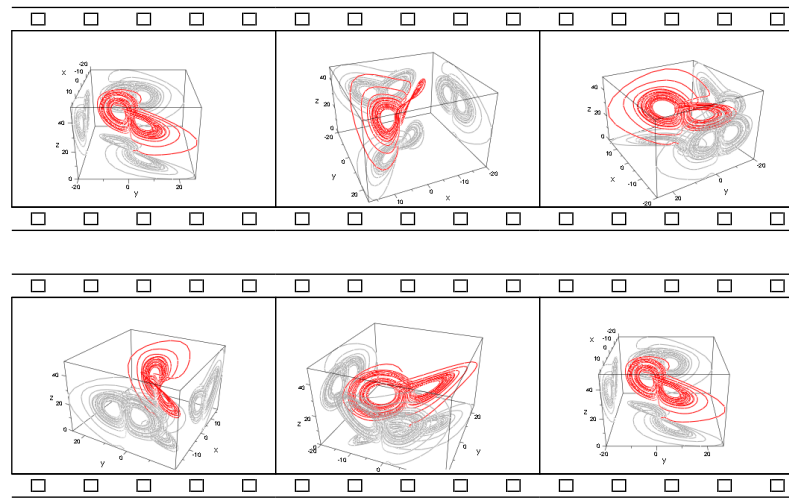
```
object := plot::Ode3d(f, [$ 1..50 step 1/10], Y0,
    [Gxyz, Style = Splines, Color = RGB::Red],
    [Gyz, Style = Splines, Color = RGB::LightGray],
    [Gxz, Style = Splines, Color = RGB::LightGray],
    [Gxy, Style = Splines, Color = RGB::LightGray]):
```

We define an animated camera moving around the scene:

```
camera := plot::Camera(
    [-1 + 100*cos(a), 6 + 100*sin(a), 120],
    [-1, 6, 25], PI/6, a = 0..2*PI, Frames = 250):
```

The following `plot` call takes about half a minute on a 3 GHz computer:

```
plot(object, camera, Axes = Boxed, TicksNumber = Low)
```



Next, we wish to fly *along the Lorenz attractor*. We cannot use `plot::Ode3d`, because we need access to the numerical data of the attractor to build a suitable animated camera object. We use the numerical ODE solver `numeric::odesolve2` and compute a list of numerical sample points on the Lorenz attractor. This takes about twenty seconds on a 3 GHz computer:

```
Y := numeric::odesolve2(f, 0, Y0, RememberLast):
timemesh :=  [$ 0..500 step 1/50]:
Y := [Y(t) $ t in timemesh]:
```

Similar to the picture above, we define a box around the attractor with the projections of the solution curve:

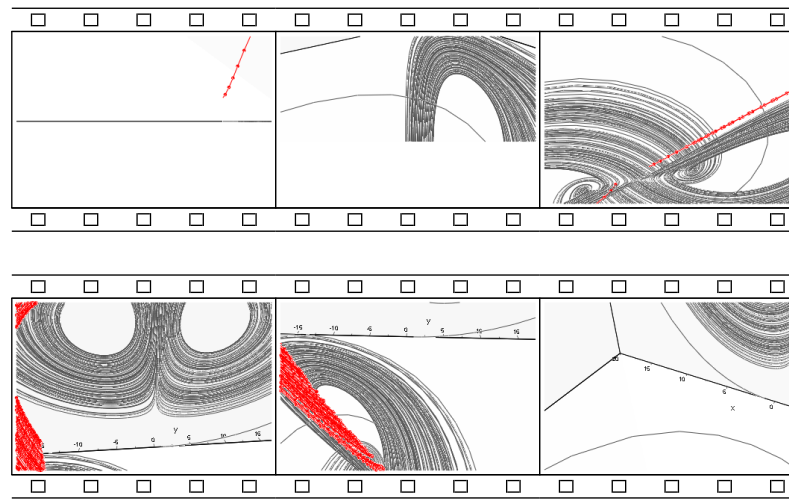
```
box := [-15, 20, -20, 26, 1, 50]:
Yyz := map(Y, pt -> [box[1], pt[2], pt[3]]):
Yxy := map(Y, pt -> [pt[1], pt[2], box[5]]):
Yxz := map(Y, pt -> [pt[1], box[3], pt[3]]):
```

We create an animated camera using an animation parameter *a* that corresponds to the index of the list of numerical sample points. The following procedure returns the i-th coordinate ($i = 1, 2, 3$) of the a-th point in the list of sample points (see Chapter 17, especially the section starting on page 17-7 for details):

```
Point := proc(a, i)
begin
  if domtype(float(a)) <> DOM_FLOAT then
       procname(args());
  else Y[round(a)][i];
  end_if;
end_proc:
```

In the a-th frame of the animation, the camera is positioned at the a-th sample point of the Lorenz attractor, pointing toward the next sample point. Setting `TimeRange=0..n/10`, the camera visits about 10 points per second:

```
n := nops(timemesh) - 1:
plot(plot::Scene3d(
       plot::Camera([Point(a, i) $ i = 1..3],
                    [Point(a + 1, i) $ i = 1..3],
                     PI/4, a = 1..n, Frames = n,
                     TimeRange = 0..n/10),
       plot::Polygon3d(Y, LineColor = RGB::Red,
                          PointsVisible = TRUE),
       plot::Polygon3d(Yxy, LineColor = RGB::DimGray),
       plot::Polygon3d(Yxz, LineColor = RGB::DimGray),
       plot::Polygon3d(Yyz, LineColor = RGB::DimGray),
       plot::Box(box[1]..box[2], box[3]..box[4],
                 box[5]..box[6], Filled = TRUE,
                 LineColor = RGB::Black,
                 FillColor = RGB::Gray.[0.1]),
       BackgroundStyle = Flat))
```



Strange Effects in 3D? Accelerated OpenGL® library?

The rendering engine for 3D plots uses the OpenGL® library. OpenGL library is a widely used graphics standard and (almost) any computer has appropriate drivers installed in its system.

By default, MuPAD® under Windows® uses a software OpenGL driver provided by the Windows operating system. Depending on the graphics card of your machine, you may also have further OpenGL drivers on your system, maybe using hardware support to accelerate your OpenGL library.

The MuPAD 3D graphics was written and tested using certain standard OpenGL drivers. The numerous drivers available on the market have different rendering quality and differ slightly. This may, in rare cases, lead to some unexpected graphical effects on your machine.

After double clicking on a 3D plot, the 'Help' menu contains an entry 'OpenGL Info ...' providing the information which OpenGL driver you are currently using. You also get the information how many light sources and how many clipping planes this driver supports.

The entry 'Configure ...' of the 'View' menu opens a dialog, which contains a section 'User Interface.' This section contains two check boxes labeled 'Accelerate OpenGL' and 'Disable OpenGL,' with the first one selecting between the divers optimized for the hardware installed (which, as outlined above, may cause problems due to bad drivers). The second checkbox should only be used in the extremely unlikely event that 3D graphics do not work at all; on some systems, opening any OpenGL program freezes the computer. Checking this box disables MuPAD 3D graphics.

Thus, if you encounter strange graphical effects in 3D, we recommend to use this configuration dialog to toggle hardware acceleration.

Exercise 11.1: Find alternative display styles for the plots in this chapter. Especially, modify the example from page 11-43 to use `plot::Ode2d` and/or `plot::Streamlines2d`.

12

Input and Output

Output of Expressions

MuPAD® does not display all computed results on the screen. Typical examples are the commands within `for` loops (Chapter 15) or procedures (Chapter 17): only the final result (i.e., the result of the last command) is printed and the output of intermediate results is suppressed. Nevertheless, you can let MuPAD print intermediate results or change the output format.

Printing Expressions on the Screen

The function `print` outputs MuPAD objects on the screen:

```
for i from 4 to 5 do
  print("The ", i, "th prime is ", ithprime(i))
end_for
```

```
    "The ", 4, "th prime is ", 7

    "The ", 5, "th prime is ", 11
```

We recall that `ithprime(i)` computes the i-th prime. MuPAD encloses the text in double quotes. Use the option `Unquoted` to suppress this:

```
for i from 4 to 5 do
  print(Unquoted,
        "The ", i, "th prime is ", ithprime(i))
end_for
```

```
    The , 4, th prime is , 7

    The , 5, th prime is , 11
```

Note that the option `Unquoted` breaks typesetting:

```
print(x^2); print(Unquoted, x^2)
```

$$x^2$$

```
    2
   x
```

Furthermore, you can eliminate the commas from the output by means of the tools for manipulating strings that are presented in Section 4.11:

```
for i from 4 to 5 do
  print(Unquoted,
        "The " . expr2text(i) . "th prime is " .
        expr2text(ithprime(i)) . ".")
end_for
```

```
    The 4th prime is 7.

    The 5th prime is 11.
```

Here, the function `expr2text` is used to convert the value of `i` and the prime returned by `ithprime(i)` to strings. Then, the concatenation operator `.` combines them with the other strings to a single string. Note that converting an expression to a string breaks typesetting as well as the "pretty print" format explained below:

```
print(Unquoted, expr2text(x^2))
```

```
    x^2
```

Alternatively you can use the function `fprint`, which writes data to a file or on the screen. In contrast to `print`, it does not output its arguments as individual expressions. Instead, `fprint` combines them to a single string (if you use the option `Unquoted`):

```
a := one: b := string:
```

```
fprint(Unquoted, 0, "This is ", a, " ", b)
  This is one string
```

The second argument `0` tells `fprint` to direct its output to the screen.

Modifying the Output Format

Usually, MuPAD will format outputs in a way similar to how mathematical formulas are usually written in books or on a blackboard ("typeset expressions"): integral signs are displayed as $\int$, symbolic sums appear as $\sum$ etc. This is the format used in most "result regions" of this book. There are two other output formats available. The following text refers to these ASCII-based output only, which are used if the typesetting is switched off via the corresponding menu of the notebook interface or when `print` with the option `Plain` is used.

Without typesetting, MuPAD usually prints expressions in a two-dimensional form with simple (ascii) characters:

```
diff(sin(x)/cos(x), x)
          2
    sin(x)
    ------- + 1
          2
    cos(x)
```

This format is known as *pretty print*. It resembles the usual mathematical notation. Therefore, it is often easier to read than a single line output. However, MuPAD only uses pretty print for output and it is not a valid input format: in a graphical user interface you cannot copy some output text with the mouse and paste it as input somewhere else. (This is possible with typeset formulas, although not for parts of a larger formula.)

The environment variable `PRETTYPRINT` controls the output format. The default value of the variable is `TRUE`, i.e., the pretty print format is used for the output. If you set this variable to `FALSE`, you obtain a one-dimensional output form which can often also be used as input:

```
PRETTYPRINT := FALSE: diff(sin(x)/cos(x), x)
  1/cos(x)^2*sin(x)^2 + 1
```

If an output would exceed the line width, the system automatically breaks the lines:

```
PRETTYPRINT := TRUE: taylor(sin(x), x = 0, 16)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} - \frac{x^{11}}{39916800} + \frac{x^{13}}{6227020800} - \frac{x^{15}}{1307674368000} + O(x^{17})$$

You can set the environment variable TEXTWIDTH to the desired line width. Its default value is 75 (characters), and you can assign any integer between 10 and $2^{31} - 1$ to it. For example, if you compute $(\ln \ln x)''$, then you obtain the following output:

```
diff(ln(ln(x)), x, x)
```

$$-\frac{1}{x^2 \ln(x)} - \frac{1}{x^2 \ln(x)^2}$$

If you reduce the value of TEXTWIDTH, the system breaks the output across two lines:

```
TEXTWIDTH := 20: diff(ln(ln(x)),x,x)
```

$$-\frac{1}{x^2 \ln(x)} - \frac{1}{x^2 \ln(x)^2}$$

The default value is restored by deleting TEXTWIDTH:

```
delete TEXTWIDTH:
```

You can also control the output by user-defined preferences. This is discussed in the section starting on page 13-2.

Reading and Writing Files

You can save the values of identifiers or a complete MuPAD® session to a file and read the file later into another MuPAD session.

The Functions `write` and `read`

The function `write` stores the values of identifiers in a file, so that you can reuse the computed results in another MuPAD session. In the following example, we save the values of the identifiers `a` and `b` to the file `ab.mb`:

```
a := 2/3: b := diff(sin(cos(x)), x):
write("ab.mb", a, b)
```

You pass the file name as a string (Section 4.11) enclosed in double quotes `"`. The system then creates a file with this name (without `"`). If you read this file into another MuPAD session via the function `read`, you can access the values of the identifiers `a` and `b` without recomputing them:

```
reset():
read("ab.mb"): a, b
```

$$\frac{2}{3},\ -\cos(\cos(x))\ \sin(x)$$

If you use the function `write` as in the above example, it creates a file in the MuPAD binary format. By convention, a file in this format should have the file name extension "`.mb`". You can call the function `write` with the option `Text`. This generates a file in a readable text format:[1]

```
a := 2/3: b := exp(x+1):
write(Text, "ab.mu", a, b)
```

[1] Usually the file name extension for MuPAD text files should be "`.mu`". The MuPAD editor allows editing such files in a comfortable way.

The file *ab.mu* now contains the following two syntactically correct MuPAD commands:

```
a := 2/3:
b := hold(exp)(hold(_plus)(hold(x), 1)):
```

You can use the function `read` to read this file:

```
a := 1: b := 2: read("ab.mu"): a, b
```

$$\frac{2}{3},\ e^{x+1}$$

The text format files generated by `write` contain valid MuPAD commands. Of course, you can use any editor to generate such a text file “by hand” and read it into a MuPAD session, using the command `read` or the entry ‘Read Commands ...’ from the ‘Notebook’ menu. In fact this is a natural way to proceed when you develop more complex MuPAD procedures.

Saving a MuPAD® Session

If you call the function `write` without supplying any identifiers as arguments, the system writes the values of *all* identifiers having a value to a file, except for those defined by the MuPADsystem itself. Thus, it is possible to restore the state of the current session via `read` at a later time:

```
result1 := ...; result2 := ...; ...
```

```
write("results.mb")
```

Reading Data from a Text File

Often you want to use data in MuPAD that are generated by other software (for example, you might want to read in statistical values for further processing), or access all files in some directory automatically. This is possible with the help of the function `import::readdata` from the library `import`. This function converts the contents of a file to a nested MuPAD list. You may regard the file as a “matrix” with line breaks indicating the beginning of a new row. Note that the rows may

have different length. You may pass an arbitrary character to `import::readdata` as a column separator.

Suppose you have a file `numericalData` with the following 4 lines:

```
1   1.2   12
2.34  234
  34  345.6
4    44      444
```

By default, blank characters are assumed as column separators. So you can read this file into a MuPAD session as follows:

```
data := import::readdata("numericalData"):
```

```
data[1]; data[2]; data[3]; data[4]
```

$$[1, 1.2, 12]$$

$$[2.34, 234]$$

$$[34, 345.6]$$

$$[4, 44, 444]$$

The help page for `import::readdata` provides further information. Also see `import::csv` for another important data exchange format.

13

Utilities

In this chapter we present some useful functions. Due to space limitations, we do not explain their complete functionality and refer to the help pages for more detailed information.

User-Defined Preferences

You can customize MuPAD®'s behavior by using *preferences*. The following command lists all preferences:

```
Pref()
  Pref::abbreviateOutput  : TRUE
  Pref::alias             : TRUE
  Pref::autoExpansionLimit: 1000
  Pref::autoPlot          : FALSE
  Pref::callBack          : NIL
  Pref::callOnExit        : NIL
  Pref::dbgAutoDisplay    : TRUE
  Pref::dbgAutoList       : TRUE
  Pref::floatFormat       : "g"
  Pref::ignoreNoDebug     : FALSE
  Pref::keepOrder         : DomainsOnly
  Pref::kernel            : [5, 6, 0]
  Pref::maxMem            : 0
  Pref::maxTime           : 0
  Pref::output            : NIL
  Pref::outputDigits      : UseDigits
  Pref::postInput         : NIL
  Pref::postOutput        : NIL
  Pref::report            : 0
  Pref::trailingZeroes    : FALSE
  Pref::typeCheck         : Interactive
  Pref::unloadableModules : FALSE
  Pref::userOptions       : ""
  Pref::verboseRead       : 0
  Pref::warnDeadProcEnv   : FALSE
```

We refer to the help page `?Pref` for a complete description of all preferences. Only a few options are discussed below.

You can use the `report` preference to request regular information on MuPAD's allocated memory, the memory really used, and the elapsed computing time. Valid arguments for `report` are integers between 0 and 9. The default value 0 means that no information is displayed. If you choose the value 9, you permanently obtain information about MuPAD's current state. You also get this information in the status bar of the notebook, but the output of `Pref::report` will

stay in the output, allowing you to review the development over time.

```
Pref::report(9): int(sin(x)^2/x, x = 0..1)
```

```
  [used=2880k, reserved=3412k, seconds=1]
  [used=3955k, reserved=4461k, seconds=1]
  [used=7580k, reserved=8203k, seconds=2]
```

$$\frac{\gamma}{2} - \frac{\mathrm{Ci}\,(2)}{2} + \frac{\ln\,(2)}{2}$$

The preference `floatFormat` controls the output of floating-point numbers. For example, if you supply the argument `"e"`, floating-point numbers are printed with mantissa and exponent (e.g., `1.234e-7` $= 1.234 \cdot 10^{-7}$). If you use the argument `"f"`, MuPAD prints them in fixed point representation:

```
Pref::floatFormat("e"): float(exp(-50))
```

$$1.928749848 \cdot 10^{-22}$$

```
Pref::floatFormat("f"): float(exp(-50))
```

$$0.00000000000000000000001928749848$$

You can use preferences to control the screen output in other ways as well. For example, after calling `Pref::output(F)`, MuPAD passes every result computed by the kernel to the function `F` before printing it on the screen. The screen output is then the result of the function `F` instead of the result originally computed by the kernel. In the following example, we use this to compute and output the normalization of every requested expression. We define a procedure `F` (Chapter 17) and pass it to `Pref::output`:

```
Pref::output(x -> (x, normal(x))):
```

```
1 + x/(x + 1) - 2/x
```

$$\frac{x}{x+1} - \frac{2}{x} + 1,\ -\frac{-2\,x^2 + x + 2}{x^2 + x}$$

The library `generate` contains functions for converting MuPAD expressions to the input format of other programming languages (C, Fortran, or TEX). In the following example, the MuPAD output is converted to a string. You might then write this string to a text file for further processing with TEX:

```
Pref::output(generate::TeX): diff(f(x),x)
    "\frac{\partial}{\partial x} f\!\left(x\right)"
```

The following command resets the output routine to its original state:

```
Pref::output(NIL):
```

Some users want to obtain information on certain characteristics of all computations, such as computing times. This can be achieved with the functions `Pref::postInput` and `Pref::postOutput`. Both take MuPAD procedures as arguments, which are then called after each input or output, respectively. In the following example, we use a procedure that assigns the system time returned by `time()` to the global identifier `Time`. This starts a "timer" before each computation:

```
Pref::postInput(proc() begin Time := time() end_proc):
```

We define a procedure `myInformation` which—among other things—uses this timer to determine the time taken by the computation. It employs `expr2text` (Section 4.11) to convert the numerical time value to a string and concatenates it with some other strings. Moreover, the procedure uses `domtype` to find the domain type of the object and converts it to a string as well. Finally it concatenates the time information, some blanks `" "`, and the type information via `_concat`. The function `length` is used to determine the precise number of blanks so that the domain type appears flushed right on the screen:

```
myInformation := proc() begin
    "domain type: ". expr2text(domtype(args()));
    "time: ". expr2text(time() - Time). " msec";
    _concat(%1,
            " " $ TEXTWIDTH-1-length(%1)-length(%2),
            %2)
end_proc:
```

We pass this procedure as argument to `Pref::postOutput`:

```
Pref::postOutput(myInformation):
```

After each computed result, the system now prints the string generated by the procedure `myInformation` on the screen:

```
factor(x^3 - 1)
```

$$(x-1)\cdot\left(x+x^2+1\right)$$

```
time: 80 msec                    domain type: Factored
```

You can reset a preference to its default value by specifying `NIL` as argument. For example, the command `Pref::report(NIL)` resets the value of `Pref::report` to 0. Similarly, `Pref(NIL)` resets *all* preferences to their default values.

Exercise 13.1: The MuPAD function `bytes` returns the amount of logical and physical memory used by the current MuPAD session. Let this information appear on the screen after each output, using `Pref::postOutput`.

The History Mechanism

Every input to MuPAD® yields a result after evaluation by the system. The computed objects are stored internally in a *history table*. Note that the result of every statement is stored, even if it is not printed on the screen. You can use it later by means of the function `last`. The command `last(1)` returns the previous result, `last(2)` the last but one, and so on. Instead of `last(i)`, you may use the shorter notation `%i`. Moreover, `%` is short for `%1` or `last(1)`. Thus the input

```
f := diff(ln(ln(x)), x): int(f, x)
```

can be passed to the system in the following equivalent form:

```
diff(ln(ln(x)), x): int(%, x)
```

This enables you to access intermediate results that have not been assigned to an identifier. It is remarkable that the use of `last` may speed up certain interactive evaluations when compared to the use of identifiers to store intermediate results. In the following example, we first try to compute a definite integral symbolically. After recognizing that MuPAD does not compute a symbolic value, we ask for a floating-point approximation:

```
f := int(sin(x)*exp(x^3)+x^2*cos(exp(x)), x=0..1)
```

$$\int_0^1 x^2 \cos(\mathrm{e}^x) + \mathrm{e}^{x^3} \sin(x)\, \mathrm{d}x$$

```
startingTime := time():
float(f);
(time() - startingTime)*msec
```

$$0.5356260737$$

$$750\,\mathrm{msec}$$

The function `time` returns the total computing time (in milliseconds) used by the system since the beginning of the session. Thus the printed difference is the time for computing the floating-point approximation.

In this example, we can reduce the computing time dramatically by employing `last`:

```
f := int(sin(x)*exp(x^3)+x^2*cos(exp(x)), x = 0..1)
```

$$\int_0^1 x^2 \cos(\mathrm{e}^x) + \mathrm{e}^{x^3} \sin(x)\, \mathrm{d}x$$

```
startingTime := time():
float(%2);
(time() - startingTime)*msec
```

$$0.5356260737$$

$$40\,\text{msec}$$

In this case, the reason for the gain in speed is that MuPAD does *not re-evaluate* the objects that `last(i)`, `%i`, or `%` refer to.[1] Thus calls to `last` form an exception to the usual complete evaluation at interactive level (Section 5.2):

```
delete x: sin(x): x := 0: %2
```

$$\sin(x)$$

You can enforce complete evaluation by using `eval`:

```
delete x: sin(x): x := 0: eval(%2)
```

$$0$$

Please note that the value of `last(i)` may differ from the i-th but last *visible* output if you have suppressed the screen output of some intermediate results by terminating the corresponding commands with a colon. Also note that the value of the expression `last(i)` changes permanently during a computation:

```
1: last(1) + 1; last(1) + 1
```

$$2$$

$$3$$

[1] Note that this gain in speed is only achieved when working interactively, since identifiers are evaluated with level 1 within procedures anyway (Section 17.11).

The environment variable HISTORY determines the number of results that MuPAD stores in a session and that can be accessed via last:

```
HISTORY
```

$$20$$

This default means that MuPAD stores the previous 20 expressions. Of course you can change this default by assigning a different value to HISTORY. This may be appropriate when MuPAD has to handle huge objects (such as very large matrices) that fill up a significant part of the main memory of your computer. Copies of these objects are stored in the history table, requiring additional storage space. In this case, you would reduce the memory load by choosing small values in HISTORY. Note that HISTORY only yields the value of the interactive "history depth." Inside a procedure, last only accepts the arguments 1, 2 and 3.

We strongly recommend to use last only interactively. The use of last within procedures is considered bad programming style and should be avoided.

Information on MuPAD® Algorithms

Some MuPAD® system functions may provide runtime information. The following command makes all such procedures produce additional information on the screen:

```
setuserinfo(Any, 1):
```

As an example, we invert the following matrix (Section 4.15) over the ring of integers modulo 11:

```
M := Dom::Matrix(Dom::IntegerMod(11)):
A := M([[1, 2, 3], [2, 4, 7], [0, 7, 5]]):
A^(-1)
```

```
 Info: using Gaussian elimination (LR decomposition)
```

$$\begin{pmatrix} 1 \bmod 11 & 0 \bmod 11 & 6 \bmod 11 \\ 3 \bmod 11 & 4 \bmod 11 & 8 \bmod 11 \\ 9 \bmod 11 & 1 \bmod 11 & 0 \bmod 11 \end{pmatrix}$$

You obtain more detailed information by increasing the second argument of `setuserinfo` (the "information level"):

```
setuserinfo(Any, 3): A^(-1)
```

```
 Info: using Gaussian elimination (LR decomposition)
 Info: searching for pivot element in column 1
 Info: choosing pivot element Dom::IntegerMod(11)(2) (row 2)
 Info: searching for pivot element in column 2
 Info: choosing pivot element Dom::IntegerMod(11)(7) (row 3)
 Info: searching for pivot element in column 3
 Info: choosing pivot element Dom::IntegerMod(11)(5) (row 3)
```

$$\begin{pmatrix} 1 \bmod 11 & 0 \bmod 11 & 6 \bmod 11 \\ 3 \bmod 11 & 4 \bmod 11 & 8 \bmod 11 \\ 9 \bmod 11 & 1 \bmod 11 & 0 \bmod 11 \end{pmatrix}$$

If you enter

```
setuserinfo(Any, 0):
```

the system stops printing additional information:

```
A^(-1)
```

$$\begin{pmatrix} 1 \bmod 11 & 0 \bmod 11 & 6 \bmod 11 \\ 3 \bmod 11 & 4 \bmod 11 & 8 \bmod 11 \\ 9 \bmod 11 & 1 \bmod 11 & 0 \bmod 11 \end{pmatrix}$$

The first argument of `setuserinfo` may be an arbitrary procedure name or library name. Then the corresponding procedure(s) provide additional information. Programmers of the system functions have built output commands into the code via `userinfo`. These commands are activated by `setuserinfo`. You can use this in your own procedures as well (`?userinfo`).

Restarting a MuPAD® Session

The command `reset()` resets a MuPAD® session to its initial state. Afterwards, all identifiers that you defined previously have no value and all environment variables are reset to their default values:

```
a := hello: DIGITS := 100: reset(): a, DIGITS
```

$$a,\ 10$$

Executing Commands of the Operating System

You can use the function `system` (or the exclamation symbol `!` for short) to execute a command of the operating system. On UNIX® platforms, the following command lists the contents of the current directory:

```
!ls
  bin  Contributions  doc  fonts  lib  mmg
```

You can neither use the output of such a command for further computation nor save it to a file.[2] `system` returns the error status of the operating system to the MuPAD® session. When called with the `!` syntax, the output of this value is suppressed.

[2] If this is desired, you can use another command of the operating system to write the output to a file and read this file into a MuPAD session via `import::readdata`.

14

Type Specifiers

The data structure of a MuPAD® object is its domain type, which can be requested by means of the function `domtype`. The domain type reflects the structure that the MuPAD kernel uses internally to manage the objects. The type concept also leads to a classification of the objects according to their mathematical meaning: numbers, sets, expressions, series expansions, polynomials, etc.

In this section, we describe how to obtain detailed information about the mathematical structure of objects. For example, how can you find out efficiently whether an integer of domain type `DOM_INT` is *positive* or *even*, or whether all elements of a set are equations?

Such type checks are barely relevant when using MuPAD interactively: you can control the mathematical meaning of an object by direct inspection. Type checks are mainly used for implementing mathematical algorithms, i.e., when programming MuPAD procedures (Chapter 17). For example, a procedure for differentiating expressions has to decide whether its input is a product, a composition of functions, a symbolic call of a known function, etc. Each case requires a different action, such as supplying the product rule, the chain rule, etc.

The Functions type and testtype

For most MuPAD® objects, the function type returns, like domtype, the domain type:

```
type([a, b]), type({a, b}), type(array(1..1))
```

DOM_LIST, DOM_SET, DOM_ARRAY

For expressions of domain type DOM_EXPR, the function type yields a finer distinction according to the mathematical meaning of the expression: sums, products, function calls, etc.:

```
type(a + b), type(a*b), type(a^b), type(a(b))
```

"_plus", "_mult", "_power", "function"

```
type(a = b), type(a < b), type(a <= b)
```

"_equal", "_less", "_leequal"

For most of these (with the exception of a(b)), the result returned by type is the name of the function that generates the expression (internally, a symbolic sum or product is represented by a call of the system functions _plus or _mult, respectively). More generally, the result for most symbolic calls of system functions is the identifier of the function as a string:

```
type(ln(x)), type(diff(f(x), x)), type(fact(x))
```

"ln", "diff", "fact"

You can use both the domain types DOM_INT, DOM_EXPR, etc. and the strings returned by type as *type specifiers*. There exists a variety of other type specifiers in addition to the "standard typing" of MuPAD objects given by type. An example is Type::Numeric. This type comprises all "numerical" objects (of domain type DOM_INT, DOM_RAT, DOM_FLOAT, or DOM_COMPLEX).

The call testtype(object, typeSpecifier) checks whether an object complies with the specified type. The result is either TRUE or FALSE. Several type specifiers may correspond to an object:

```
testtype(2/3, DOM_RAT), testtype(2/3, Type::Numeric)
```

TRUE, TRUE

```
testtype(2 + x, "_plus"), testtype(2 + x, DOM_EXPR)
```

TRUE, TRUE

```
testtype(f(x), "function"), testtype(f(x), DOM_EXPR)
```

TRUE, TRUE

Exercise 14.1: Consider the expression

$$f(i) = \frac{i^{5/2} + i^2 - i^{1/2} - 1}{i^{5/2} + i^2 + 2\,i^{3/2} + 2\,i + i^{1/2} + 1}$$

How can MuPAD decide whether the set

```
S := {f(i) $ i = -1000..-2} union {f(i) $ i=0..1000}:
```

contains only rational numbers? Hint: For a specific integer `i`, use the function `normal` to simplify subexpressions of `f(i)` containing square roots.

Exercise 14.2: Consider the expressions $\sin(i\,\pi/200)$ with integer values of i between 0 and 100. Which of them are simplified by the MuPAD `sin` function, which are returned as symbolic values `sin(·)`?

Comfortable Type Checking: the Type Library

The type specifiers presented above are useful only for checking relatively simple structures. For more advanced type checking, more flexible type specifications are needed. For example, how can you check without direct inspection whether the object `[1, 2, 3, ...]` is a list of positive integers?

For that purpose, the `Type` library provides further type specifiers and constructors. You can use them to create your own type specifiers, which are recognized by `testtype`:

```
info(Type)
  Library 'Type': type expressions and properties
  -- Interface:
  Type::AlgebraicConstant, Type::AnyType,
  Type::Arithmetical,      Type::Boolean,
  Type::Complex,           Type::Condition,
  Type::Constant,          Type::ConstantIdents,
  Type::ElementOf,         Type::Equation,
  Type::Even,              Type::Function,
  Type::Imaginary,         Type::IndepOf,
  Type::Indeterminate,     Type::Integer,
  Type::Intersection,      Type::Interval,
  Type::ListOf,            Type::ListProduct,
  Type::NegInt,            Type::NegRat,
  Type::Negative,          Type::NonNegInt,
  Type::NonNegRat,         Type::NonNegative,
  Type::NonZero,           Type::Numeric,
  Type::Odd,               Type::PolyExpr,
  Type::PolyOf,            Type::PosInt,
  Type::PosRat,            Type::Positive,
  Type::Predicate,         Type::Prime,
  Type::Product,           Type::Property,
  Type::RatExpr,           Type::Rational,
  Type::Real,              Type::Relation,
  Type::Residue,           Type::SequenceOf,
  Type::Series,            Type::Set,
  Type::SetOf,             Type::Singleton,
  Type::TableOf,           Type::TableOfEntry,
  Type::TableOfIndex,      Type::Union,
  Type::Unknown,           Type::Zero
```

For example, the type specifier `Type::PosInt` represents the set of positive integers $n > 0$, `Type::NonNegInt` corresponds to the nonnegative integers $n \geq 0$, `Type::Even` and `Type::Odd` represent the even and odd integers, respectively. These type specifiers are of domain type `Type`:

```
domtype(Type::Even)
```

Type

You can use such type specifiers to query the mathematical structure of MuPAD objects via `testtype`. In the following example, we extract all even integers from a list of integers via `select` (Section 4.6):

```
select([i $ i = 1..20], testtype, Type::Even)
```

$[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$

You can use constructors such as `Type::ListOf` or `Type::SetOf` to perform type checking for lists or sets. For example, a list of integers is matched by the type `Type::ListOf(DOM_INT)`, a set of equations corresponds to the type specifiers `Type::SetOf(Type::Equation())` (or `Type::SetOf("_equal")`), a set of odd integers is matched by the type `Type::SetOf(Type::Odd)`:

```
T := Type::ListOf(DOM_INT):
```

```
testtype([-1, 1], T), testtype({-1, 1}, T),
testtype([-1, 1.0], T)
```

TRUE, FALSE, FALSE

The constructor `Type::Union` generates type specifiers corresponding to the union of simpler types. For example, the following type specifier

```
T := Type::Union(DOM_FLOAT, Type::NegInt, Type::Even):
```

matches real floating-point numbers as well as negative integers as well as even integers:

```
testtype(-0.123, T), testtype(-3, T),
testtype(2, T), testtype(3, T)
```

TRUE, TRUE, TRUE, FALSE

We describe an application of type checking for the implementation of MuPAD procedures in Section 17.7.

Exercise 14.3: How can you compute the intersection of a set with the set of positive integers?

Exercise 14.4: Use `?Type::ListOf` to consult the help page for this type constructor. Construct a type specifier corresponding to a list of two elements such that each element is again a list with three arbitrary elements.

15

Loops

Loops are important elements of the MuPAD® programming language. The following example illustrates the simplest form of a `for` loop:

```
for i from 1 to 4 do
    x := i^2;
    print("The square of", i, "is", x)
end_for:
```

```
    "The square of", 1, "is", 1

    "The square of", 2, "is", 4

    "The square of", 3, "is", 9

    "The square of", 4, "is", 16
```

The loop variable `i` automatically runs through the values $1, 2, 3, 4$. For each value of `i`, all commands between `do` and `end_for` are executed. There may be arbitrarily many commands, separated by semicolons or colons. *The system does not print the results computed in each loop iteration on the screen, even if you terminate the commands by semicolons.* For that reason, we used the `print` command to generate an output in the above example.

The following variant counts backwards. We use the tools from Section 4.11 to make the output look more appealing:

```
for j from 4 downto 2 do
   print(Unquoted,
         "The square of ".expr2text(j)." is ".
         expr2text(j^2))
end_for:
```

```
    The square of 4 is 16

    The square of 3 is 9

    The square of 2 is 4
```

You can use the keyword `step` to increment or decrement the loop variable in bigger or smaller steps:

```
for x from 3 to 8 step 2 do print(x, x^2) end_for:
```

```
    3, 9

    5, 25

    7, 49
```

Note that at the end of the iteration with $x = 7$ the value of x is incremented to 9. This exceeds the upper bound 8, and the loop terminates. Here is another variant of the `for` loop:

```
for i in [5, 27, y] do print(i, i^2) end_for:
```

```
    5, 25

    27, 729

        2
    y, y
```

The loop variable only runs through the values from the list $[5, 27, y]$. As you can see, such a list may contain symbolic elements such as the variable y.

In a `for` loop, a loop variable changes according to fixed rules (typically, it is incremented or decremented). The `repeat` loop is a more flexible alternative, where you can arbitrarily modify many variables in each step. In the following example, we compute the squares of the integers $i = 2, 2^2, 2^4, 2^8, \ldots$ until $i^2 > 100$ holds for the first time:

```
x := 2:
```

```
repeat
    i := x; x := i^2; print(i, x)
until x > 100 end_repeat:
```

```
    2, 4

    4, 16

    16, 256
```

The system executes the commands between `repeat` and `until` repeatedly. The loop terminates when the condition between `until` and `end_repeat` holds true. In the above example, we have $i = 4$ and $x = 16$ at the end of the second step. Hence the third step is executed, and afterwards we have $i = 16$, $x = 256$. Now the termination condition $x > 100$ is satisfied and the loop terminates.

Another loop variant is the `while` loop:

```
x := 2:
```

```
while x <= 100 do
    i := x; x := i^2; print(i, x)
end_while:
```

```
    2, 4

    4, 16

    16, 256
```

In a `repeat` loop, the system checks the termination condition *after* each loop iteration. In a `while` loop, this condition is checked *before* each iteration. As soon as the condition evaluates to `FALSE`, the system terminates the `while` loop.

You can use `break` to abort a loop explicitly. Typically this is done within an `if` construction (Chapter 16):

```
for i from 3 to 100 do
    print(i);
    if i^2 > 20 then break end_if
end_for:
```

```
    3

    4

    5
```

After a call to `next`, the system skips all commands up to `end_for`. It returns immediately to the beginning of the loop and starts the next iteration with the next value of the loop variable:

```
for i from 2 to 5 do
    x := i;
    if i > 3 then next end_if;
    y := i;
    print(x, y)
end_for:
```

```
    2, 2

    3, 3
```

For $i > 3$, only the first assignment `x:=i` is executed:

```
x, y
```

$$5,\ 3$$

We recall that every MuPAD command returns an object. For a loop, this is the return value of the most recently executed command. If you do not terminate the loop command with a colon as in all of the above examples, then MuPAD displays this value:

```
delete x: for i from 1 to 3 do x.i := i^2 end_for
   9
```

You may process this value further. In particular, you can assign it to an identifier or use it as the return value of a MuPAD procedure (Chapter 17):

```
factorial := proc(n)
  local result;
  begin
    result := 1;
    for i from 2 to n do
      result := result * i
    end_for
  end_proc:
```

The return value of the above procedure is the return value of the `for` loop, which in turn is the value of the last assignment to `result`.

Internally, loops are system function calls. For example, MuPAD processes a `for` loop by evaluating the function `_for`:

```
_for(i, first_i, last_i, increment, command):
```

This is equivalent to

```
for i from first_i to last_i step increment do
    command
end_for:
```

16

Branching: `if-then-else` and `case`

Branching instructions are an important element of every programming language. Depending on the value or the meaning of variables, different commands are executed. The simplest variant in MuPAD® is the `if` statement:

```
for i from 2 to 4 do
  if isprime(i)
     then print(expr2text(i)." is prime")
     else print(expr2text(i)." is not prime")
  end_if
end_for:
```

```
    "2 is prime"

    "3 is prime"

    "4 is not prime"
```

Here, the primality test `isprime(i)` returns either `TRUE` or `FALSE`. If the value is `TRUE`, the system executes the commands between `then` and `else` (in this case, only one `print` command). If it is `FALSE`, the commands between `else` and `end_if` are executed. The `else` branch is optional:

```
for i from 2 to 4 do
  if isprime(i)
     then text := expr2text(i)." is prime";
          print(text)
  end_if
end_for:
```

```
    "2 is prime"

    "3 is prime"
```

Here, the then branch comprises two commands separated by a semicolon (or, alternatively, a colon). You may nest commands, loops, and branching statements arbitrarily:

```
primes := []: evenNumbers := []:
```

```
for i from 30 to 50 do
  if isprime(i)
     then primes := primes.[i]
     else if testtype(i,Type::Even)
             then evenNumbers := evenNumbers.[i]
          end_if
  end_if
end_for:
```

In this example, we inspect the integers between 30 and 50. If we encounter a prime, then we append it to the list primes. Otherwise, we use testtype to check whether i is even (see Sections 14.1 and 14.2). In that case, we append i to the list evenNumbers. Upon termination, the list primes contains all prime numbers between 30 and 50, whilst evenNumbers contains all even integers in this range:

```
primes, evenNumbers
```

$$[31, 37, 41, 43, 47]\,,\ [30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50]$$

You can create more complex conditions for the if statement by using the Boolean operators and, or, and not (Section 4.10). The following for loop prints all prime twins $[i, i+2]$ with $i \leq 100$. The alternative condition not (i>3) yields the additional pair $[2, 4]$:

```
for i from 2 to 100 do
  if (isprime(i) and isprime(i+2)) or not (i>3)
     then print([i,i+2])
  end_if
end_for:
```

```
                          [2, 4]

                          [3, 5]

                          [5, 7]

                            ...
```

Internally, an `if` statement is just a call of the system function `_if`:

```
_if(condition, command1, command2):
```

is equivalent to

```
if condition then command1 else command2 end_if:
```

The return value of an `if` statement is, as for any MuPAD procedure, the result of the last executed command:[1]

```
x := -2: if x > 0 then x else -x end_if
```

$$2$$

For example, you can use the arrow operator `->` (Section 4.12) to implement the absolute value of numbers as follows:

```
Abs := y -> (if y > 0 then y else -y end_if):
```

```
Abs(-2), Abs(-2/3), Abs(3.5)
```

$$2,\ \frac{2}{3},\ 3.5$$

As you can see, you can use `if` commands both at the interactive level and within procedures. The typical application is in programming MuPAD procedures, where `if` statements and loops control the flow of the algorithm. A simple example is the above function `Abs`. You find more examples in Chapter 17.

If you have several nested `if` ... `else` `if` ... constructions, you can abbreviate this by using the `elif` statement:

```
if condition1 then
  statements1
elif condition2 then
  statements2
elif ...
else
  statements
end_if:
```

[1] If no command is executed then the result is `null()`.

This is equivalent to the following nested `if` statement:

```
if condition1 then
  statements1
else if condition2 then
       statements2
     else if ...
          else
            statements
          end_if
     end_if
end_if:
```

A typical application is for type checking within procedures (Chapter 17). The following version of `Abs` computes the absolute value if the input is an integer, a rational number, a real floating-point number, or a complex numbera. Otherwise, it raises an error:

```
Abs := proc(y) begin
  if (domtype(y) = DOM_INT) or (domtype(y) = DOM_RAT)
     or (domtype(y) = DOM_FLOAT) then
       if y > 0 then y else -y end_if;
  elif (domtype(y) = DOM_COMPLEX) then
       sqrt(Re(y)^2 + Im(y)^2);
  else "Invalid argument type" end_if:
end_proc:
```

```
delete x: Abs(-3), Abs(5.0), Abs(1+2*I), Abs(x)
```

$3,\ 5.0,\ \sqrt{5},$ "Invalid argument type"

In our example, we distinguish several cases according to the evaluation of a single expression. We can also implement this by using a case statement, which is often easier to read:

```
case domtype(y)
of DOM_INT do
of DOM_RAT do
of DOM_FLOAT do
  if y > 0 then y else -y end_if;
  break;
of DOM_COMPLEX do
  sqrt(Re(y)^2 + Im(y)^2);
  break;
otherwise
  "Invalid argument type";
end_case:
```

The keywords case and end_case indicate the beginning and the end of the statement, respectively. MuPAD evaluates the expression after case. If the result matches one of the expressions between of and do, the system executes all commands from the first matching of on until it encounters either a break or the keyword end_case.

Warning: Note that, if no break statement used in a branch, the following branches are entered and executed, too. This is in the same style as the switch statement in the C programming language. It allows several branches to share the same code.

If none of the of branches applies and there is an otherwise branch, the code between otherwise and end_case is executed. The return value of a case statement is the value of the last executed command. We refer to the corresponding help page ?case for a more detailed description.

As for loops and if statements, there is a functional equivalent for a case statement: the system function _case. Internally, MuPAD converts the above case statement to the following equivalent form:

```
_case(domtype(y),
      DOM_INT, NIL,
      DOM_RAT, NIL,
      DOM_FLOAT,
        (if y > 0 then y else -y end_if; break),
      DOM_COMPLEX, (sqrt(Re(y)^2 + Im(y)^2); break),
      "Invalid argument type"):
```

Exercise 16.1: In if statements or termination conditions of while and repeat loops, the system evaluates composite conditions with Boolean operators one after the other. The evaluation routine stops prematurely if it can decide whether the final result is TRUE or FALSE ("lazy evaluation"). Are there problems with the following statements? What happens when the conditions are evaluated?

```
A := x/(x - 1) > 0: x := 1:
```

```
(if x <> 1 and A then right else wrong end_if),
(if x = 1 or A then right else wrong end_if)
```

17

MuPAD® Procedures

MuPAD® provides the essential constructs of a programming language. The user can implement complex algorithms comfortably in MuPAD. Indeed, most of MuPAD's mathematical intelligence is not implemented in C or C++ within the kernel, but in the MuPAD programming language at the library level. The programming features are more extensive than in other languages such as C, Pascal, or Fortran, since the MuPAD language offers more general and more flexible constructs.

We have already presented basic structures such as loops (Chapter 15), branching instructions (Chapter 16), and "simple" functions (page 4-52).

In this chapter, we regard "programming" as writing complex MuPAD procedures. In principle the user recognizes no differences between "simple functions" generated via `->` (page 4-52) and "more complex procedures" as presented in this chapter. Procedures, like functions, return values. Only the way of generating such procedure objects via `proc` ... `end_proc` is a little more complicated. Procedures provide additional functionality: there is a distinction between local and global variables, you can use arbitrarily many commands in a clear and convenient way etc.

As soon as a procedure is implemented and assigned to an identifier, you may call it in the form `procedureName(arguments)` like any other MuPAD function. After executing the implemented algorithm, it returns an output value.

You can define and use MuPAD procedures within an interactive session, like any other MuPAD object. Typically, however, you want to use these procedures again in later sessions, in particular when they implement more complex algorithms. Then it is useful to write the procedure definition into a text file using your favorite text editor (such as, e.g., the MuPAD source code editor), and read it into

a MuPAD session via read (page 12-5) or with the entry 'Read commands ...' from the 'Notebook' menu. Apart from the evaluation level, the MuPAD kernel processes the commands in the file exactly in the same way as if they were entered interactively. MuPAD includes an editor with syntax highlighting for MuPAD programs.

Defining Procedures

The following function, which compares two numbers and returns their maximum, is an example of a procedure definition via `proc` ... `end_proc`:

```
maximum := proc(a, b) /* comment: maximum of a and b */
           begin
             if a<b then return(b) else return(a) end_if;
           end_proc:
```

The text enclosed between `/*` and `*/` is a comment[1] and completely ignored by the system. This is a useful tool for documenting the source code when you write the procedure definition in a text file.

The above sample procedure contains an `if` statement as the only command. More realistic procedures contain many commands (separated by colons or semicolons). The command `return` terminates a procedure and passes its argument as output value to the system.

A MuPAD® object generated via `proc` ... `end_proc` is of domain type `DOM_PROC`:

```
domtype(maximum)
```

```
DOM_PROC
```

You can decompose and manipulate a procedure like any other MuPAD object. In particular, you may assign it to an identifier, as above. The syntax of the function call is the same as for other MuPAD functions:

```
maximum(3/7, 0.4)
```

$$\frac{3}{7}$$

The statements between `begin` and `end_proc` may be arbitrary MuPAD commands. In particular, you may call system functions or other procedures from within a procedure. A procedure may even call itself, which is helpful for implementing recursive algorithms.

[1] Alternatively, you can start a comment by `//`. A comment started with `//` automatically ends at the end of the line.

The favorite example for a recursive algorithm is the computation of the factorial $n! = 1 \cdot 2 \cdot \cdots \cdot n$ of a nonnegative integer, which may be defined by the rule $n! = n \cdot (n-1)!$ together with the initial condition $0! = 1$. The realization as a recursive procedure might look as follows:

```
factorial := proc(n) begin
  if n = 0 then
       return(1)
  else return(n*factorial(n - 1))
  end_if
end_proc:
```

```
factorial(10)
```

$$3628800$$

The environment variable MAXDEPTH determines the maximal number of nested procedure calls. Its default value is 500. With this value, the above factorial function works only for $n \leq 500$. For larger values, after MAXDEPTH steps MuPAD assumes that there is an infinite recursion and aborts with an error message. It is advisable to avoid deep recursions, to save on memory, increase efficiency and to avoid these problems.

The Return Value of a Procedure

When you call a procedure, the system executes its body, i.e., the sequence of statements between `begin` and `end_proc`. Every procedure returns some value, either explicitly via `return` or otherwise *the value of the last command executed within the procedure*.[2] Thus, you can implement the above factorial function without using `return`:

```
factorial := proc(n) begin
  if n = 0 then 1 else n*factorial(n - 1) end_if;
end_proc:
```

The `if` statement returns either 1 or $n\,(n-1)!$. Since the end of the procedure is reached directly after the `if` statement, this is the return value of the call `factorial(n)`.

As soon as the system encounters a `return` statement, it terminates the procedure:

```
factorial := proc(n) begin
  if n = 0 then return(1) end_if;
  n*factorial(n - 1);
end_proc:
```

For $n = 0$, MuPAD® does not execute the last statement (the recursive call of `n * factorial(n - 1)`) after returning 1. For $n \neq 0$, the most recently computed value is `n * factorial(n - 1)`, which is then the return value of the call `factorial(n)`.

[2] If there is no command to execute, the return value is `NIL`.

A procedure may return an arbitrary MuPAD object, such as an expression, a sequence, a set, a list, or even a procedure. However, if the returned procedure uses local variables of the outer procedure, you have to declare the latter with the option `escape`. Otherwise, this leads to a MuPAD warning message or to unwanted effects. The following procedure returns a function that uses the parameter `power` of the outer procedure:

```
generatePowerFunction := proc(power)
      option escape;
      begin
        x -> (x^power)
      end_proc:
```

```
f := generatePowerFunction(2):
```

```
g := generatePowerFunction(5):
```

```
f(a), g(b)
```

$$a^2,\ b^5$$

Returning Symbolic Function Calls

Many system functions return "themselves" as symbolic function calls if they cannot find a simple representation of the requested result:

```
sin(x), max(a, b), int(exp(sin(x)), x)
```

$$\sin(x)\,,\ \max(a, b)\,,\ \int \mathrm{e}^{\sin(x)}\,\mathrm{d}x$$

You achieve the same behavior in your own procedures when you encapsulate the procedure name in a `hold` upon return. The `hold` (page 5-4) prevents the function from calling itself recursively and ending up in an infinite recursion. The following function computes the absolute value for numerical inputs (integers, rational numbers, real floating-point numbers, and complex numbers). For all other kinds of inputs, it returns itself symbolically:

```
Abs := proc(x) begin
  if testtype(x, Type::Numeric) then
     if domtype(x) = DOM_COMPLEX then
          return(sqrt(Re(x)^2 + Im(x)^2))
     else if x >= 0 then
               return(x)
          else return(-x)
          end_if
     end_if
  end_if;
  hold(Abs)(x)
end_proc:
```

```
Abs(-1), Abs(-2/3), Abs(1.234), Abs(2 + I/3),
Abs(x + 1)
```

$$1,\ \frac{2}{3},\ 1.234,\ \frac{\sqrt{37}}{3},\ \mathrm{Abs}(x+1)$$

A more elegant way is to use the MuPAD® object `procname`, which returns the name of the calling procedure:

```
Abs := proc(x) begin
  if testtype(x, Type::Numeric) then
     if domtype(x) = DOM_COMPLEX then
          return(sqrt(Re(x)^2 + Im(x)^2))
     else if x >= 0 then
               return(x)
          else return(-x)
          end_if
     end_if
  end_if;
  procname(args())
end_proc:
```

```
Abs(-1), Abs(-2/3), Abs(1.234), Abs(2 + I/3),
Abs(x + 1)
```

$$1,\ \frac{2}{3},\ 1.234,\ \frac{\sqrt{37}}{3},\ \mathrm{Abs}(x+1)$$

Here, we use the expression `args()`, which returns the sequence of arguments passed to the procedure (cf. page 17-21).

Local and Global Variables

You can use arbitrary identifiers in procedures. They are also called *global variables*:

```
a := b: f := proc() begin a := a^2 + 1 end_proc:
```

```
f(); f(); f()
```

$$b^2 + 1$$

$$\left(b^2 + 1\right)^2 + 1$$

$$\left(\left(b^2 + 1\right)^2 + 1\right)^2 + 1$$

The procedure `f` modifies the value of `a`, which has been set outside the procedure. When the procedure terminates, `a` has a new value, which is again changed by further calls to `f`.

The keyword `local` declares identifiers as *local variables* that are only valid within the procedure:

```
a := b: f := proc() local a; begin a := 2 end_proc:
```

```
f(): a
```

$$b$$

Despite the equal names, the assignment `a:=2` of the local variable does not affect the value of the global identifier `a` that has been defined outside the procedure. You can declare an arbitrary number of local variables by specifying a sequence of identifiers after `local`:

```
f := proc(x, y, z)
     local A, B, C;
     begin
        A:= 1; B:= 2; C:= 3; A*B*C*(x + y + z)
     end_proc:
```

```
f(A, B, C)
```

$$6\,A + 6\,B + 6\,C$$

Local variables of a procedure have a special domain type DOM_VAR. They do not get mixed up with global variables which are identifiers of type DOM_IDENT. Note that also local variables declared by the same name in the source code of different functions have no reference to one another. Also, when calling a function several times, a local variable refers to a different value in each call:

```
f := proc(x) local a, b;
begin
  a := x;
  if x > 0 then b := f(x - 1);
  else          b := 1;
  end_if;
  print(a, x);
  b + a;
end:
f(2)
```

```
   0, 0

   1, 1

   2, 2

   4
```

We recommend to take into account the following rule of thumb:

> Using global variables is generally considered bad programming style. Use local variables whenever possible.

The reason for this principle is that procedures implement mathematical functions, which should return a unique output value for a given set of input values. If you use global variables then, depending on their values, the same procedure call may lead to different results:

```
a := 1: f := proc(b) begin a := a + 1; a + b end_proc:
```

```
f(1), f(1), f(1)
```

```
   3, 4, 5
```

Moreover, a procedure call can change the calling environment in a subtle way by redefining global variables (“side effect”). In more complex programs, this may lead to unwanted effects that are difficult to debug.

An important difference between global and local variables is that an uninitialized global variable is regarded as a *symbol*, whose value is its own name, while the value of an uninitialized local variable is `NIL`. Using a local variable without initialization leads to a warning message in MuPAD® and should be avoided:

```
IAmGlobal + 1
```

$$\text{IAmGlobal} + 1$$

```
f := proc()
  local IAmLocal;
  begin
    IAmLocal + 1
  end_proc:
```

```
f()
  Warning: Uninitialized variable 'IAmLocal' used;
  during evaluation of 'f'
  Error: Illegal operand [_plus];
  during evaluation of 'f'
```

The reason for the error is that MuPAD cannot add the value `NIL` of the local variable to the number 1.

We now present a realistic example of a meaningful procedure. If we use arrays of domain type `DOM_ARRAY` to represent matrices, we are faced with the problem that there is no direct way to perform matrix multiplication with such arrays.[3] The following procedure solves this problem: you can compute the matrix product $C = A \cdot B$ with the command `C:=MatrixProduct(A, B)`. We want the procedure to work for arbitrary dimensions of the matrices A and B, provided the result is defined mathematically. If A is an $m \times n$ matrix, then B may be an $n \times r$ matrix, where m, n, r are arbitrary positive integers. The result is the $m \times r$ matrix C with the entries

$$C_{ij} = \sum_{k=1}^{n} A_{ik}\, B_{kj}\,, \quad i = 1, \ldots, m, \quad j = 1, \ldots, r.$$

The multiplication procedure below automatically extracts the dimension parameters m, n, r from the arguments, namely from the 0-th operands of the input arrays (page 4-44). If B is a $q \times r$ matrix with $q \neq n$, the multiplication is not defined mathematically. In this case, the procedure terminates with an error

[3] If you use the data type `Dom::Matrix()` instead, you can directly use the standard operators +, -, *, ^, / for arithmetic with matrices (cf. page 4-70 ff.).

message. For that purpose, we employ the function `error`, which aborts the calling procedure and writes the string passed as argument on the screen. We store the result component by component in the local variable `C`. We initialize this variable as an array of dimension $m \times r$, so that the result of our procedure is of the desired data type `DOM_ARRAY`. We might implement the sum over k in the computation of C_{ij} as a loop of the form `for k from 1 to n do ..`. Instead, we use the system function `_plus` which returns the sum of its arguments. We generally recommend to use these system functions, if possible, since they work quite efficiently. The return value of `MatrixProduct` is the final expression `C`:

```
MatrixProduct := /* multiplication C=AB of an m x n */
proc(A, B)       /* matrix A by an n x r Matrix B   */
local m, n, r, i, j, k, C; /* with arrays A, B of   */
begin                      /* domain type DOM_ARRAY */
    m := op(A, [0, 2, 2]);
    n := op(A, [0, 3, 2]);
    if n <> op(B, [0, 2, 2]) then
      error("incompatible matrix dimensions")
    end_if;
    r := op(B, [0, 3, 2]);
    C := array(1..m, 1..r);       /* initialization */
      for i from 1 to m do
        for j from 1 to r do
          C[i, j] := _plus(A[i, k]*B[k, j] $ k = 1..n)
        end_for
      end_for;
      C
  end_proc:
```

A general remark about MuPAD programming style: you should always perform argument checking in procedures meant for interactive use. If you implement a procedure, you usually know which types of inputs are valid (such as `DOM_ARRAY` in the above example). If somebody passes parameters of the wrong type by mistake, this usually leads to system functions calls with invalid arguments, and your procedure aborts with an error message originating from a system function. In the above example, the function `op` returns the value `FAIL` when accessing the 0-th operand of `A` or `B` and one of them is not of type `DOM_ARRAY`. Then this value is assigned to `m`, `n` or `r`, and the following `for` loop aborts with an error message, since `FAIL` is not allowed as a value for the endpoint of the loop.

In such a situation, it is often difficult to locate the source of the error. However, an even worse scenario might happen: if the procedure does not abort, the result is likely to be wrong! Thus, type checking helps to avoid errors.

In the above example, we might add a type check of the form

```
if domtype(A) <> DOM_ARRAY or domtype(B) <> DOM_ARRAY
  then error("arguments must be of type DOM_ARRAY")
end_if
```

to the procedure body. Starting on page 17-20, we discuss a simpler type checking concept.

Subprocedures

Often tasks occur frequently within a procedure and you want to implement them again in the form of a procedure. This structures and simplifies the program code. In many cases, such a procedure is used only from within a single procedure. Then it is reasonable to define this procedure locally as a subprocedure only in the scope of the calling procedure. In MuPAD® you can use local variables to implement subprocedures. If you want to make

```
g := proc() begin ... end_proc:
```

a local procedure of

```
f := proc() begin ... end_proc:
```

define `f` as follows:

```
f := proc()
local g;
begin
  g := proc() begin ... end_proc;   /* subprocedure */

  /* main part of f, which calls g(..): */
  ...
end_proc:
```

Now, `g` is a *local* procedure of `f` and you can use it only from within `f`.

We give an example. You can implement matrix multiplication by means of suitable column×row multiplications:

$$\begin{pmatrix} 2 & 1 \\ 5 & 3 \end{pmatrix} \cdot \begin{pmatrix} 4 & 6 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} (2,1) \cdot \begin{pmatrix} 4 \\ 2 \end{pmatrix} & (2,1) \cdot \begin{pmatrix} 6 \\ 3 \end{pmatrix} \\ (5,3) \cdot \begin{pmatrix} 4 \\ 2 \end{pmatrix} & (5,3) \cdot \begin{pmatrix} 6 \\ 3 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 10 & 15 \\ 26 & 39 \end{pmatrix}.$$

More generally, if we partition the input matrices by rows a_i and columns b_j, respectively, then

$$\begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix} \cdot (b_1, \ldots, b_n) = \begin{pmatrix} a_1 \cdot b_1 & \ldots & a_1 \cdot b_n \\ \vdots & \ddots & \vdots \\ a_m \cdot b_1 & \ldots & a_m \cdot b_n \end{pmatrix},$$

with the inner product

$$a_i \cdot b_j = \sum_r (a_i)_r \, (b_j)_r \; .$$

We now write a procedure `MatrixMult` that expects input arrays A and B of the form `array(1..m, 1..k)` and `array(1..k, 1..n)`, and returns the $m \times n$ matrix product $A \cdot B$. A call of the subprocedure `RowTimesColumn` with arguments i, j extracts the i-th row and the j-th column from the input matrices A and B, respectively, and computes the inner product of the row and the column. The subprocedure uses the arrays `A`, `B` as well as the locally declared dimension parameters `m`, `n`, and `k` as "global" variables:

```
MatrixMult := proc(A, B)
local m, n, k, K,     /* local variables */
      RowTimesColumn; /* local subprocedure */
begin
  /* subprocedure */
  RowTimesColumn := proc(i, j)
  local row, column, r;
  begin
    /* ith row of A: */
    row := array(1..k, [A[i,r] $ r=1..k]);
    /* jth column of B: */
    column := array(1..k, [B[r,j] $ r=1..k]);
    /* row times column */
    _plus(row[r]*column[r] $ r=1..k)
  end_proc;

  /* main part of the procedure MatrixMult: */
  m := op(A, [0, 2, 2]); /* number of rows of A */
  k := op(A, [0, 3, 2]); /* number of columns of A */
  K := op(B, [0, 2, 2]); /* number of rows of B */
  n := op(B, [0, 3, 2]); /* number of columns of B */

  if k <> K then
     error("# of columns of A <> # of rows of B")
  end_if;

  /* matrix A*B: */
  array(1..m, 1..n,
        [[RowTimesColumn(i, j) $ j=1..n] $ i=1..m])
end_proc:
```

The following example returns the desired result:

```
A := array(1..2, 1..2, [[2, 1], [5, 3]]):
```

```
B := array(1..2, 1..2, [[4, 6], [2, 3]]):
```

```
MatrixMult(A, B)
```

$$\begin{pmatrix} 10 & 15 \\ 26 & 39 \end{pmatrix}$$

Scope of Variables

The MuPAD® programming language implements *lexical scoping*. This essentially means that the scope of a procedure's local variables and parameters can already be determined when the procedure is defined. We start with a simple example to explain this concept.

```
p := proc() begin x end_proc:
```

```
x := 3: p(); x := 4: p()
```

$$3$$

$$4$$

```
q := proc() local x; begin x := 5; p(); end_proc:
```

```
q()
```

$$4$$

First, a procedure `p` without arguments is defined. It uses a variable `x`, which is not declared as a local variable of `p`. Thus, the call `p()` returns the value of the global variable `x`, as shown in the two subsequent calls. In the procedure `q`, however, the variable `x` is declared local, and the value 5 is assigned to it. The global variable `x` is not visible from within the procedure `q`, only the local variable `x` can be accessed. Nevertheless, the call `q()` returns the value of the *global* variable `x`, and *not* the current value of the local variable `x` within `p`. We might, for example, define a subprocedure within `q` to achieve the latter behavior:

```
x := 4:
```

```
q := proc()
    local x, p;
  begin
    x := 5;
    p := proc() begin x; end_proc;
    x := 6;
    p()
  end_proc:
```

```
q(), p()
```

$$6,\ 4$$

The previously defined global procedure p is not accessed from within q because a local variable p is declared. The call of the local procedure p from within q now indeed returns the current value of the local variable x, as shown by the call q(). The last command p(), however, executes the global procedure p defined before, which still returns the current value 4 of the global variable x.

Here is another example:

```
p := proc(x) begin 2 * cos(x) + 1; end_proc:
```

```
q := proc(y)
    local cos;
  begin
    cos := proc(z) begin z + 1; end_proc;
    p(y) * cos(y)
  end_proc:
```

```
p(PI), q(PI)
```

$$-1,\ -\pi - 1$$

The procedure p uses MuPAD's globally defined cosine function. Within q, we have defined a local subprocedure cos. Calling q, the internal call cos(y) refers to the local function cos. Even when called from within q, the procedure p still uses the MuPAD global cosine function.

When using the option escape, a local variable can be accessed even when it has escaped the scope of the procedure that it was local to. In the following example, the "counter" returned by f refers to the local variable x of f. The procedures counter1, counter2 etc. are not local to f anymore but still use the reference to x to synchronize subsequent calls of each counter. Note that both counters refer to *different* independent instances of x, so that they count independently:

```
f := proc() local x;
       option escape;
     begin
       x := 1;
       // The following procedure
       // is the return value of f:
       proc() begin x := x + 1 end;
     end:
```

```
counter1 := f(): counter2 := f():
```

```
counter1(), counter1(), counter1();
counter2(), counter2();
counter1(), counter2(), counter1();
```

```
   2, 3, 4

   2, 3

   5, 4, 6
```

Type Declaration

MuPAD® provides easy-to-use type checking for procedure arguments. For example, you can restrict the arguments of `MatrixProduct`, a procedure from the Section starting on page 17-9, to the domain type `DOM_ARRAY` as follows:

```
MatrixProduct := proc(A: DOM_ARRAY, B: DOM_ARRAY)
                   local m, n, r, i, j, k, C;
                 begin ...
```

If you declare the type of the parameters of a procedure in the form used above, *argument*`:` *typeSpecifier*, a call of the procedure with parameters of an incompatible type leads to an error message. In the example above, we used the domain type `DOM_ARRAY` as a type specifier.

We have discussed MuPAD's type concept in Chapter 14. The `Type` library offers `Type::NonNegInt` to represent the set of nonnegative integers. If we use it in the following variant of the factorial function

```
factorial := proc(n: Type::NonNegInt) begin
  if n = 0 then
       return(1)
  else n*factorial(n - 1)
  end_if
end_proc:
```

then only nonnegative integers are permitted for the argument `n`:

```
factorial(4)
```

$$24$$

```
factorial(4.0)
```

```
  Error: Wrong type of 1. argument (type 'Type::NonNeg\
  Int' expected,
         got argument '4.0');
  during evaluation of 'factorial'
```

```
factorial(-4)
```

```
  Error: Wrong type of argument 'n' (type 'Type::NonNeg\
  Int' expected,
         got argument '-4');
  during evaluation of 'factorial'
```

Procedures with a Variable Number of Arguments

The system function max computes the maximum of its arguments. You may call it with arbitrarily many arguments:

```
max(1), max(3/7, 9/20), max(-1, 3, 0, 7, 3/2, 7.5)
```

$$1,\ \frac{9}{20},\ 7.5$$

You can implement this behavior in your own procedures as well. The function args returns the arguments passed to the calling procedure:

`args(0)`	:	the number of arguments,
`args(i)`	:	the i-th argument, $1 \le i \le$ `args(0)`,
`args(i..j)`	:	the sequence of arguments from i to j, $1 \le i \le j \le$ `args(0)`,
`args()`	:	the sequence `args(1)`, `args(2)`, ... of all arguments.

The following function simulates the behavior of the system function max:

```
maximum := proc() local m, i; begin
  m := args(1);
  for i from 2 to args(0) do
    if m < args(i) then m := args(i) end_if
  end_for:
  m
end_proc:
```

```
maximum(1), maximum(3/7, 9/20), maximum(-1, 3, 0, 7, 3/2, 7.5)
```

$$1,\ \frac{9}{20},\ 7.5$$

Here, we initialize m with the first argument. Then, we test for each of the remaining arguments whether it is greater than m, and if so, replace m by the corresponding argument. Thus, m contains the maximum at the end of the loop. Note that if you call maximum with only one argument (so that `args(0)=1`), then the loop `for i from 2 to 1 do ...` is not executed at all.

You may use both formal parameters and accesses via `args` in a procedure:

```
f := proc(x, y) begin
       if args(0) = 3 then
            x^2 + y^3 + args(3)^4
       else x^2 + y^3
       end_if
     end_proc:
```

```
f(a, b), f(a, b, c)
```

$$a^2 + b^3,\ a^2 + b^3 + c^4$$

The following example is a trivial function returning itself symbolically for any number of arguments that are actually passed:

```
f := proc() begin procname(args()) end_proc:
```

```
f(1), f(1, 2), f(1, 2, 3), f(a1, b2, c3, d4)
```

$$f(1),\ f(1,2),\ f(1,2,3),\ f(\mathrm{a1}, \mathrm{b2}, \mathrm{c3}, \mathrm{d4})$$

Options: the Remember Table

When declaring MuPAD® procedures, you can specify *options* that affect the execution of a procedure call. Besides the option `escape` already mentioned, the option `remember` may be of interest to the user. In this section, we take a closer look at this option and demonstrate its effect with an example. The sequence of Fibonacci numbers is defined by the recursion

$$F_n = F_{n-1} + F_{n-2}\,, \quad F_0 = 0, \quad F_1 = 1.$$

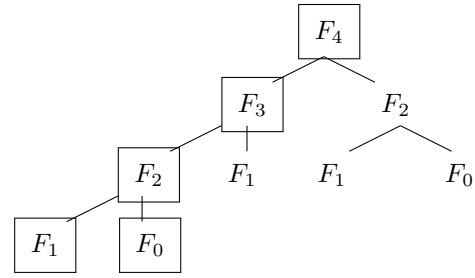
It is easy to translate this into a MuPAD procedure:

```
F := proc(n: Type::NonNegInt)
     begin
       if n < 2 then n
       else F(n - 1) + F(n - 2) end_if
     end_proc:
```

```
F(i) $ i = 0..10
```

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55$$

This way of computing F_n is highly inefficient for larger values of n. To see why, let us trace the recursive calls of `F` when computing F_4. You may regard this as a tree structure: `F(4)` calls `F(3)` and `F(2)`, `F(3)` calls `F(2)` and `F(1)` etc.:



One can show that the call `F(n)` leads to about $1.45.. \cdot (1.618..)^n$ calls of `F` for large n. These "costs" grow dramatically fast for increasing values of n:

```
time(F(10)), time(F(15)), time(F(20)), time(F(25))
```

$$80,\ 910,\ 10290,\ 113600$$

We recall that the function `time` returns the time in milliseconds used to evaluate its argument.

We see that many calls (such as, for example, `F(1)`) are executed several times. For evaluating `F(4)`, it is sufficient to execute only the boxed function calls `F(0)`, ..., `F(4)` in the above figure and to store these values. All other computations of `F(0)`, `F(1)`, `F(2)` are redundant since their results are already known. This is precisely what MuPAD does when you declare `F` with the `option remember`:

```
F := proc(n: Type::NonNegInt)
       option remember;
     begin
       if n < 2 then n
       else F(n - 1) + F(n - 2) end_if
     end_proc:
```

The system internally creates a remember table for the procedure `F`, which initially is empty. At each call to `F`, MuPAD checks whether there is an entry for the current argument sequence in this table. If this is the case, the procedure is not executed at all, and the result is taken from the table. If the current arguments do not appear in the table, the system executes the procedure body as usual and returns its result. Then it appends the argument sequence and the return value to the remember table. This ensures that a procedure is not unnecessarily executed twice with the same arguments.

In the Fibonacci example, the call `F(n)` now leads to only $n+1$ calls to compute `F(0)`, ..., `F(n)`. In addition, the system searches the remember table $n-2$ times. However, this happens very quickly. In this example, the benefit in speed from using `option remember` is quite dramatic:

```
time(F(10)), time(F(15)), time(F(20)), time(F(25)),
time(F(500))
```

$$0,\ 10,\ 0,\ 10,\ 390$$

The real running times are so small that the system cannot measure them exactly. This explains the (rounded) times of 0 milliseconds for `F(10)` and `F(20)`.

> Using `option remember` in a procedure is promising whenever a procedure is called frequently with the same arguments.

Of course, you can implement this method for computing the Fibonacci numbers directly by computing F_n iteratively instead of recursively and storing already computed values in a table:[4]

```
F := proc(n: Type::NonNegInt)
       local i, F;
     begin
       F[0] := 0: F[1] := 1:
       for i from 2 to n do
         F[i] := F[i - 1] + F[i - 2]
       end_for
     end_proc:
```

```
time(F(10)), time(F(15)), time(F(20)), time(F(25)),
time(F(500))
```

$$10,\ 0,\ 10,\ 0,\ 400$$

The function `numlib::fibonacci` from the number theory library is yet faster for large arguments, since it uses more elaborate algorithms and direct formulas.

Warning: The remember mechanism recognizes only previously processed inputs, but does not consider the values of possibly used global variables. When the values of these global variables change, then the remembered return values are usually wrong. In particular, this is the case for global environment variables such as `DIGITS`:

```
floatexp := proc(x) option remember;
            begin float(exp(x)) end_proc:
```

```
DIGITS := 20: floatexp(1);
```

$$2.7182818284590452354$$

```
DIGITS := 40: floatexp(1); float(exp(1))
```

$$2.7182818284590452353602874713449239278 42$$

$$2.7182818284590452353602874713526624977 57$$

Here, the system outputs the remembered value of `floatexp(1)` with higher precision after switching from 20 to 40 `DIGITS`. Nevertheless, this is still the value computed with `DIGITS=20`; the output only shows all digits that were used

[4] This procedure is not properly implemented: what happens when you call `F(0)`?

internally[5] in this computation. The last of the three numbers is the true value of `exp(1)` computed with 40 digits. It differs from the wrongly remembered value at the 30-th decimal digit.

You can explicitly add new entries to the remember table of a procedure. In the following example, f is the function $x \mapsto \sin(x)/x$, which has a removable singularity at $x = 0$. The limit is $f(0) := \lim_{x \to 0} \sin(x)/x = 1$:

```
f := proc(x) begin sin(x)/x end_proc: f(0)
```

```
  Error: Division by zero;
  during evaluation of 'f'
```

You can easily add the value for $x = 0$:

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

$$1$$

The assignment `f(0):=1` creates a remember table for `f`, so that a later call of `f(0)` does not try to evaluate the value of $\sin(x)/x$ for $x = 0$. Now you can use `f` without running into danger at $x = 0$.

Warning: The call:

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

does *not* generate the function $f : x \mapsto x^2$, but rather creates a remember table for the identifier `f` with the entry `x^2` *only for the symbolic identifier* `x`. Any other call to `f` returns a symbolic function call:

```
f(x), f(y), f(1)
```

$$x^2,\ f(y)\,,\ f(1)$$

[5] Internally, MuPAD uses a certain number of additional "guard digits" exceeding the number of digits requested via `DIGITS`. However, for the output, the system truncates the internally computed value to the requested number of digits.

Input Parameters

The declared formal arguments of a procedure can be used like additional local variables:

```
f := proc(a, b) begin a := 1; a + b end_proc:
```

```
a := 2: f(a, 1): a
```

$$2$$

Modifying *a* within this procedure does not affect the identifier *a* that is used outside the procedure. You should be cautious when you access the calling arguments via `args` (page 17-21) in a procedure after changing some input parameter. Assignment to a formal parameter changes the return value of `args`:

```
f := proc(a) begin a := 1; a, args(1) end_proc: f(2)
```

$$1,\ 1$$

In principle, arbitrary MuPAD® objects may be input parameters of a procedure. Thus you can use sets, lists, expressions, or even procedures and functions:

```
p := proc(f) begin
       [f(1), f(2), f(3)]
     end_proc:
```

```
p(g)
```

$$[g(1), g(2), g(3)]$$

```
p(proc(x) begin x^2 end_proc)
```

$$[1, 4, 9]$$

```
p(x -> x^3)
```

$$[1, 8, 27]$$

In general, user-defined procedures evaluate their arguments ("call by value"): when you call `f(x)`, the procedure `f` knows only the value of the identifier `x`. If you declare a procedure with the `option hold`, then the "call by name" scheme is used instead: the expression or the name of the identifier passed as the actual argument is seen by the procedure. In this case, you can use `context` to obtain the value of the expression:

```
f := proc(x) option hold;
     begin x = context(x) end_proc:
```

```
x := 2:
```

```
f(x), f(sin(x)), f(sin(PI))
```

$$x = 2,\ \sin(x) = \sin(2)\,,\ \sin(\pi) = 0$$

Evaluation Within Procedures

In Chapter 5, we have discussed MuPAD®'s evaluation strategy: complete evaluation at interactive level. Thus all identifiers are replaced by their values *recursively* until only symbolic identifiers without a value remain (or the evaluation depth given by the environment variable `LEVEL` is reached):

```
delete a, b, c: x := a + b: a := b + 1: b := c:
```

```
x
```

$$2\,c + 1$$

In contrast, within procedures, the system performs evaluation not completely but only with evaluation depth 1. This is similar to internally replacing each identifier by `level(identifier, 1)`: every identifier is replaced by its value, but *not recursively*. We recall from page 5-1 the distinction between the *value* of an identifier (the evaluation at the time of assignment) and its *evaluation* (the "current value," where symbolic identifiers that have been assigned a value in the meantime are replaced by their values as well). In interactive mode, calling an object yields its complete evaluation, while in procedures only the object's value is returned. This explains the difference between the interactive result above and the following result:

```
f := proc() begin
       delete a, b, c:
       x := a + b: a := b + 1: b := c:
       x
     end_proc:
```

```
f()
```

$$a + b$$

The reason why two different behaviors are implemented is that the strategy of incomplete evaluation makes the evaluation of procedures faster and increases the efficiency of MuPAD procedures considerably. For a beginner in programming MuPAD procedures, this evaluation concept has its pitfalls. However, after some practice you acquire an appropriate programming style so that you can work with the restricted evaluation depth without problems.

Warning: If you do not work interactively with MuPAD, but use an editor to write your MuPAD commands into a text file which is read into a MuPAD session via `read` (page 12-5), these commands are executed within a procedure (namely, `read`). Consequently, the evaluation depth is 1.

You can use the system function `level` (page 5-4) to control the evaluation depth and to enforce complete evaluation if necessary:

```
f := proc() begin
       delete a, b, c:
       x := a + b: a := b + 1: b := c:
       level(x)
     end_proc:
```

```
f()
```

$$2c+1$$

Warning: `level` does not evaluate local variables.

Moreover, you cannot use local variables as symbolic identifiers; you must initialize them before use. The following procedure, in which a symbolic identifier `x` is passed to the integration function, is invalid:

```
f := proc(n) local x; begin int(exp(x^n), x) end_proc:
```

You can pass the name of the integration variable as additional argument to the procedure. Thus, the following variants are valid:

```
f := proc(n, x) begin int(exp(x^n), x) end_proc:
```

```
f := proc(n, x) local y; begin
     y := x; int(exp(y^n), y) end_proc:
```

If you need symbolic identifiers for intermediate results, you can generate an identifier without a value via `genident()` (page 4-8) and assign it to a local variable.

Function Environments

MuPAD® provides a variety of tools for handling built-in mathematical standard functions such as `sin`, `cos`, `exp`. These tools implement the mathematical properties of these functions. Typical examples are the `float` conversion routine, the differentiation function `diff`, or the function `expand`, which you use to manipulate expressions:

```
float(sin(1)), diff(sin(x), x, x, x),
expand(sin(x + 1))
```

$$0.8414709848,\ -\cos(x),\ \cos(1)\ \sin(x) + \sin(1)\ \cos(x)$$

In this sense, the mathematical knowledge about the standard functions is distributed over several system functions: the function `float` has to know how to compute numerical approximations of the sine function, `diff` must know its derivative, and `expand` has to know the addition theorems of the trigonometric functions.

You can invent arbitrary new functions as symbolic names or implement them in procedures. How can you pass the knowledge about the mathematical meaning and the rules of manipulation for the new functions to the other system functions? For example, how can you tell the differentiation routine `diff` what the derivative of your newly created function is? If the function is composed of standard functions known to MuPAD, such as, for example, $f : x \mapsto x\ \sin(x)$, then this is no problem. The call

```
f := x -> (x*sin(x)): diff(f(x), x)
```

$$\sin(x) + x\ \cos(x)$$

immediately yields the desired answer. However, often there are situations where the newly implemented function cannot be composed from standard objects. Thus, our goal is to hand the rules of manipulation (floating-point approximation, differentiation etc.) for *symbolic* function calls to the MuPAD functions `float`, `diff` etc. This is the actual challenge when you "implement a new mathematical function in MuPAD:" to distribute the knowledge about the mathematical meaning of the symbol to MuPAD's standard tools. Indeed, this is a necessary task: for example, if you want to differentiate a more complex expression containing both the new function and some standard functions, then this is only

possible via the system's differentiation routine. Thus, the latter has to learn how to handle the new symbols.

For that purpose, MuPAD provides the domain type DOM_FUNC_ENV (short for: function environment). Indeed, all built-in mathematical standard functions are of this type in order to enable float, diff, expand etc. to handle them:

```
domtype(sin)
```

DOM_FUNC_ENV

You can call a function environment like any "normal" function or procedure:

```
sin(1.7)
```

0.9916648105

A function environment consists of three operands. The first operand is a procedure that computes the return value of a function call. The second operand is a procedure for printing a symbolic function call on the screen. The third operand is a table containing information how the system functions float, diff, expand, etc. should handle symbolic function calls.

You can look at the procedure for evaluating a function call with the function expose, just like for normal functions:

```
expose(sin)
```

```
  proc(x)
    name sin;
    local f, y;
    option noDebug;
  begin
    if args(0) = 0 then
      error("no arguments given")
    else
    ...
  end_proc
```

To keep the example manageable, we will choose two closely related functions and act as if they were not implemented in MuPAD yet. Our example functions are the complete elliptic integral functions of the first and second kind, $K(z)$ and $E(z)$. Since MuPAD already has a predefined identifier E, we will implement these functions as ellipE and, for consistency, ellipK. (The more obvious names

`ellipticE` and `ellipticK` are already used by the MuPAD versions of the same functions.) These functions appear in such different contexts as calculating the perimeter of an ellipsis, the gravitational or electrostatic potential of a uniform ring or the probability that a random walk in three dimensions ever goes through the origin. For this presentation, let us concentrate on the following properties of the functions E and K:

$$E'(z) = \frac{E(z) - K(z)}{2\,z},$$
$$K'(z) = \frac{E(z) - (1-z)K(z)}{2\,(1-z)z},$$

$$E(0) = K(0) = \frac{\pi}{2}, \qquad E(1) = 1,$$
$$K\left(\frac{1}{2}\right) = \frac{8\,\pi^{3/2}}{\Gamma\left(-\frac{1}{4}\right)^2}, \qquad K(-1) = \frac{\Gamma\left(\frac{1}{4}\right)^2}{4\,\sqrt{2\pi}}.$$

That is, we are going to implement the derivatives of E and K with the above relations and make the functions evaluate at special points. Additionally, we will implement the functions in such a way that they are written as E and K in the output.

The basic functions are easy to write:

```
ellipE :=
 proc(x) begin
   if x = 0 then PI/2
   elif x = 1 then 1
   else procname(x) end_if
 end_proc:
```

```
ellipK :=
 proc(x) begin
   if x = 0 then PI/2
   elif x = 1/2 then 8*PI^(3/2)/gamma(-1/4)^2
   elif x = -1 then gamma(1/4)^2/4/sqrt(2*PI)
   else procname(x) end_if
 end_proc:
```

Since the values of the functions are known only at specific places, we use `procname` (page 17-7) to return the symbolic expressions `ellipE(x)` and `ellipK(x)`, respectively, for all arguments where the function values are not known. This yields:

```
ellipE(0), ellipE(1/2),
ellipK(12/17), ellipK(x^2+1)
```

$$\frac{\pi}{2},\ \mathrm{ellipE}\left(\frac{1}{2}\right),\ \mathrm{ellipK}\left(\frac{12}{17}\right),\ \mathrm{ellipK}\left(x^2+1\right)$$

You generate a new function environment by means of `funcenv`:

```
output_E := f -> hold(E)(op(f)):
ellipE := funcenv(ellipE, output_E):
```

```
output_K := f -> hold(K)(op(f)):
ellipK := funcenv(ellipK, output_K):
```

These commands convert the procedures `ellipE` and `ellipK` to function environments. The first arguments are the procedures for evaluation. The (optional) second argument of `funcenv` is the procedure for screen output. We want a symbolic expression `ellipE(x)` displayed as `E(x)` and likewise `ellipK(x)` shall be displayed as `K(x)`. This is achieved by the second argument of `funcenv`, which you should interpret as a conversion routine. On input `ellipE(x)`, it returns the MuPAD object to print on the screen instead of `ellipE(x)`. The argument `f`, which represents `ellipE(x)`, is converted to the unevaluated function call `E(x)` (note that `x = op(f)` for `f = ellipE(x)` and that you need `hold(E)` to prevent evaluation of the identifier `E`). The system outputs this expression instead of `ellipE(x)` on the screen:[6]

```
ellipE(0), ellipE(1/2),
ellipK(12/17), ellipK(x^2+1)
```

$$\frac{\pi}{2},\ E\left(\frac{1}{2}\right),\ K\left(\frac{12}{17}\right),\ K\left(x^2+1\right)$$

The (optional) third argument to `funcenv` is a table of *function attributes*. It tells the system functions `float`, `diff`, `expand` etc. how to handle symbolic calls of the

[6] Note that you should avoid returning strings from such procedures. Using strings breaks both pretty-printing and typesetting of the output.

form `ellipK(x)` and `ellipE(x)`. In the example above, we did not provide any such function attributes. Hence, the system functions do not yet know how to proceed and, by default, return the expression or themselves symbolically:

```
float(ellipE(1/3)), expand(ellipE(x + y)),
diff(ellipE(x), x), diff(ellipK(x), x)
```

$$E\left(\frac{1}{3}\right),\ E(x+y),\ \frac{\partial}{\partial x}E(x),\ \frac{\partial}{\partial x}K(x)$$

By assigning to the `"diff"` slot of our function environments, we set the attributes for the differentiation routine `diff`:

```
ellipE::diff :=
 proc(f,x)
  local z;
 begin
  z := op(f);
  (ellipE(z) - ellipK(z))/(2*z) * diff(z, x)
 end_proc:
```

```
ellipK::diff :=
 proc(f,x)
  local z;
 begin
  z := op(f);
  (ellipE(z) - (1-z)*ellipK(z))/
     (2*(1-z)*z) * diff(z, x)
 end_proc:
```

These commands tell `diff` that `diff(f, x)` with a symbolic function call `f = ellipE(z)`, where `z` depends on `x`, should apply the procedure assigned to `ellipE::diff`. The well-known chain rule yields

$$\frac{\mathrm{d}}{\mathrm{d}x}E(z) = E'(z)\,\frac{\mathrm{d}z}{\mathrm{d}x}\,.$$

The specified procedure implements this rule, where the inner function in the expression `f = ellipE(z)` is given by `z = op(f)`.

Now MuPAD knows the derivatives of the functions represented by the identifiers `ellipE` *and* `ellipK`. We have already implemented the screen outputs:

```
diff(ellipE(z), z), diff(ellipE(y(x)), x);
diff(ellipE(x*sin(x)), x)
```

$$\frac{E(z)-K(z)}{2\,z},\ \frac{(E(y(x))-K(y(x)))\ \frac{\partial}{\partial x}y(x)}{2\,y(x)}$$

$$\frac{(E(x\ \sin(x))-K(x\ \sin(x)))\ (\sin(x)+x\ \cos(x))}{2\,x\ \sin(x)}$$

As far as `diff` is concerned, the implementation of our two elliptic integrals is now complete:

```
diff(ellipE(x), x, x)
```

$$\frac{\frac{E(x)-K(x)}{2\,x}+\frac{E(x)+K(x)\,(x-1)}{2\,x\,(x-1)}}{2\,x}-\frac{E(x)-K(x)}{2\,x^2}$$

```
normal(diff(ellipK(2*x + 3), x, x, x))
  - (73 E(2 x + 3) + 86 K(2 x + 3) + 115 x E(2 x + 3) +

                                 2
     228 x K(2 x + 3) + 46 x  E(2 x + 3) +

          2                    3
     202 x  K(2 x + 3) + 60 x  K(2 x + 3)) /

          6         5         4          3          2
     (32 x  + 240 x  + 744 x  + 1220 x  + 1116 x  +

     540 x + 108)
```

As an application, we now want MuPAD to compute the first terms of the Taylor expansion of the complete elliptic integral of the first kind around $x = 0$. We can use the function `taylor` since it calls `diff` internally:

```
taylor(ellipK(x), x = 0, 6)
```

$$\frac{\pi}{2}+\frac{\pi\,x}{8}+\frac{9\,\pi\,x^2}{128}+\frac{25\,\pi\,x^3}{512}+\frac{1225\,\pi\,x^4}{32768}+\frac{3969\,\pi\,x^5}{131072}+O\left(x^6\right)$$

But there is more: With functions such as the elliptic integrals, which appear in their own derivatives, the integration routine has a good chance of finding symbolic integrals once the `diff` attributes have been implemented:

```
int(ellipE(x), x)
```

$$\frac{2\,E(x)}{3} - \frac{2\,K(x)}{3} + x\left(\frac{2\,E(x)}{3} + \frac{2\,K(x)}{3}\right)$$

Exercise 17.1: Extend the definitions of `ellipE` and `ellipK` by a `"float"` slot. Use `hypergeom::float` and the following equivalences:

$$E(z) = \frac{\pi}{2}\,\mathsf{hypergeom}\left(\left[-\frac{1}{2}, \frac{1}{2}\right], [1], z\right)$$

$$K(z) = \frac{\pi}{2}\,\mathsf{hypergeom}\left(\left[\frac{1}{2}, \frac{1}{2}\right], [1], z\right)$$

Also, extend the definitions of the functions such that your new float evaluation is automatically used when a floating point value is given as input.

Exercise 17.2: Implement an absolute value function `Abs` as a function environment. The call `Abs(x)` should return the absolute value for real numbers `x` of domain type `DOM_INT`, `DOM_RAT`, or `DOM_FLOAT`. For all other arguments, the symbolic output `|x|` should appear on the screen. The absolute value is differentiable on $\mathbb{R}\backslash\{0\}$. Its derivative is

$$\frac{\mathrm{d}\,|y|}{\mathrm{d}\,x} = \frac{|y|}{y}\,\frac{\mathrm{d}y}{\mathrm{d}x}\,.$$

Set the `"diff"` attribute accordingly and compute the derivative of `Abs(x^3)`. Compare your result to the corresponding derivative of the system function `abs`.

A Programming Example: Differentiation

In this section, we discuss an example demonstrating the typical mode of operation of a symbolic MuPAD® procedure. We implement a symbolic differentiation routine that computes the derivatives of algebraic expressions composed of additions, multiplications, exponentiations, some mathematical functions (exp, ln, sin, cos, ...), constants, and symbolic identifiers.

This example is only for illustration purposes, since MuPAD already provides such a function: the `diff` routine. This function is implemented in the MuPAD kernel and therefore very fast. A user-defined function that is written in the MuPAD programming language can hardly achieve the efficiency of `diff`.

The following algebraic differentiation rules are valid for the class of expressions that we consider:

(1) $\dfrac{\mathrm{d}f}{\mathrm{d}x} = 0$, if f does not depend on x,

(2) $\dfrac{\mathrm{d}x}{\mathrm{d}x} = 1$,

(3) $\dfrac{\mathrm{d}(f+g)}{\mathrm{d}x} = \dfrac{\mathrm{d}f}{\mathrm{d}x} + \dfrac{\mathrm{d}g}{\mathrm{d}x}$ (linearity),

(4) $\dfrac{\mathrm{d}a\,b}{\mathrm{d}x} = \dfrac{\mathrm{d}a}{\mathrm{d}x}\,b + a\,\dfrac{\mathrm{d}b}{\mathrm{d}x}$ (product rule),

(5) $$\frac{\mathrm{d}\,a^b}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}x}\,e^{b\,\ln(a)} = e^{b\,\ln(a)}\,\frac{\mathrm{d}}{\mathrm{d}x}\,(b\,\ln(a))$$
$$= a^b\,\ln(a)\,\frac{\mathrm{d}b}{\mathrm{d}x} + a^{b-1}\,b\,\frac{\mathrm{d}a}{\mathrm{d}x},$$

(6) $\dfrac{\mathrm{d}}{\mathrm{d}x}\,F(y(x)) = F'(y)\,\dfrac{\mathrm{d}y}{\mathrm{d}x}$ (chain rule).

Moreover, for some functions F, the derivative is known, and we want to take this into account in our implementation. For an unknown function F, we return the symbolic function call of the differentiation routine.

The procedure `Diff` in Table 17.1 implements the above properties in the stated order. Its calling syntax is `Diff(expression, identifier)`.

```
Diff := proc(f, x : DOM_IDENT)                    // (0)
local a, b, F, y; begin
   if not has(f, x) then return(0) end_if;        // (1)
   if f = x then return(1) end_if;                // (2)
   if type(f) = "_plus" then
      return(map(f, Diff, x)) end_if;             // (3)
   if type(f) = "_mult" then
      a := op(f, 1); b := subsop(f, 1 = 1);
      return(Diff(a, x)*b + a*Diff(b, x))         // (4)
   end_if;
   if type(f) = "_power" then
      a := op(f, 1); b := op(f, 2);
      return(f*ln(a)*Diff(b, x)
             + a^(b - 1)*b*Diff(a, x))            // (5)
   end_if;
   if op(f, 0) <> FAIL then
      F := op(f, 0); y := op(f, 1);               // (6)
      if F = hold(exp) then
         return( exp(y)*Diff(y, x)) end_if;       // (6)
      if F = hold(ln)  then
         return(  1/y  *Diff(y, x)) end_if;       // (6)
      if F = hold(sin) then
         return( cos(y)*Diff(y, x)) end_if;       // (6)
      if F = hold(cos) then
         return(-sin(y)*Diff(y, x)) end_if;       // (6)
      /* specify further known functions here */
   end_if;
   procname(args())                               // (7)
end_proc:
```

Table 17.1: A symbolic differentiation routine

In (0), we implemented an automatic type check of the second argument, which must be a symbolic identifier of domain type `DOM_IDENT`.

In (1), the MuPAD function `has` checks whether the expression f to be differentiated depends on x.

Linearity of differentiation is implemented in (3) by means of the MuPAD

function `map`:

```
map(f1(x) + f2(x) + f3(x), Diff, x)
```

$$\text{Diff}(\text{f1}(x), x) + \text{Diff}(\text{f2}(x), x) + \text{Diff}(\text{f3}(x), x)$$

In (4), we handle a product expression $f = f_1 \cdot f_2 \cdot \ldots$: the command `a:=op(f,1)` determines the first factor $a = f_1$, then `subsop(f, 1=1)` replaces this factor by 1, such that b assumes the value $f_2 \cdot f_3 \cdot \ldots$. Then, we call `Diff(a, x)` and `Diff(b, x)`. If $b = f_2 \cdot f_3 \cdot \ldots$ is itself a product, this leads to another execution of step (4) at the next recursion level. In this way, (4) handles products of arbitrary length.

Step (5) differentiates powers. For $f = a^b$, the call `op(f, 1)` returns the base a and `op(f, 2)` the exponent b. In particular, this covers all monomial expressions of the form $f = x^n$ for constant n. The recursive calls to `Diff` for $a = x$ and $b = n$ then yield `Diff(a, x)=1` and `Diff(b, x)=0`, respectively, and the expression returned in (5) simplifies to the correct result $n\,x^{n-1}$.

If the expression f is a symbolic function call of the form $f = F(y)$, we extract the "outer" function F in (6) via `F:=op(f, 0)` (otherwise, `F` gets the value `FAIL`). Next, we handle the case where F is a function with one argument y and extract the "inner" function by `y:=op(f, 1)`. If F is the name of a function with known derivative (such as $F = \exp, \ln, \sin, \cos$), then we apply the chain rule. It is easy to extend this list of functions F with known derivatives. In particular, you can add a formula for differentiating symbolic expressions of the form `int(·, ·)`. Extensions to functions F with more than one argument are also possible.

Finally, step (7) returns `Diff(f, x)` symbolically if no simplifications of the expression f happen in steps (1) through (6).

`Diff`'s mode of operation is adopted from the system function `diff`. Compare the following results to those returned by `diff`:

```
Diff(x*ln(x + 1/x), x)
```

$$\ln\left(x + \frac{1}{x}\right) - \frac{x\,\left(\frac{1}{x^2} - 1\right)}{x + \frac{1}{x}}$$

```
Diff(f(x)*sin(x^2), x)
```

$$\sin\left(x^2\right)\ \text{Diff}(f(x), x) + 2\,x\,\cos\left(x^2\right)\ f(x)$$

Programming Exercises

Exercise 17.3: Write a short procedure `date` that takes three integers `month`, `day`, `year` as input and prints the date in the usual way. For example, the call `date(5, 3, 1990)` should yield the screen output `5/3/1990`.

Exercise 17.4: We define the function $f : \mathbb{N} \to \mathbb{N}$ by

$$f(x) = \begin{cases} 3\,x + 1 \text{ for odd } x\,, \\ x/2 \text{ for even } x\,. \end{cases}$$

The "$(3\,x + 1)$ problem" asks whether for an arbitrary initial value $x_0 \in \mathbb{N}$, the sequence recursively defined by $x_{i+1} := f(x_i)$ contains the value 1. Write a program that on input x_0 returns the smallest index i with $x_i = 1$.

Exercise 17.5: Implement a function `Gcd` to compute the greatest common divisor of two positive integers. Of course, you should not use the system functions `gcd` and `igcd`. Hint: the Euclidean Algorithm for computing the gcd is based on the observation

$$\text{gcd}(a,\, b) \;=\; \text{gcd}(a \;mod\; b,\, b) \;=\; \text{gcd}(b,\, a \;mod\; b)$$

and the facts $\text{gcd}(0,\, b) = \text{gcd}(b,\, 0) = b$.

Exercise 17.6: Implement a function `Quadrature`. For a function `f` and a MuPAD® list `X` of numerical values

$$x_0 < x_1 < \cdots < x_n,$$

the call `Quadrature(f, X)` should compute a numerical approximation of the integral

$$\int_{x_0}^{x_n} f(x)\,dx \;\approx\; \sum_{i=0}^{n-1} (x_{i+1} - x_i)\, f(x_i).$$

Exercise 17.7: Newton's method for finding a numerical root of a function $f : \mathbb{R} \mapsto \mathbb{R}$ employs the iteration $x_{i+1} = F(x_i)$, where $F(x) = x - f(x)/f'(x)$. Write a procedure `Newton`. The call `Newton(f, x0, n)`, with an expression `f`, should return the first $n + 1$ elements $x_0, \ldots, x_n$ of the Newton sequence.

For bonus points, extend the example on page 11-34 to display the sequence just found.

Exercise 17.8: The *Sierpinski triangle* is a well-known fractal. We define a variant of it as follows. The Sierpinski triangle is the set of all points $(x, y) \in \mathbb{N} \times \mathbb{N}$ with the following property: there exists at least one position in the binary expansions of x and y such that both have a 1-bit at this position. Write a program `Sierpinski` that on input `xmax`, `ymax` plots the set of all such points with integer coordinates in the range $1 \leq x \leq$ `xmax`, $1 \leq y \leq$ `ymax`. Hint: the function `numlib::g_adic` computes the binary expansion of an integer. The graphical primitive `plot::PointList2d` can be used to create a plot of the points.

Exercise 17.9: A *logical formula* is composed of identifiers and the operators `and`, `or`, and `not`. For example:

```
formula := (x and y) or
           ((y or z) and (not x) and y and z)
```

Such a formula is called *satisfiable* if it is possible to assign the values `TRUE` and `FALSE` to all identifiers in such a way that the formula can be evaluated to `TRUE`. Write a program that checks whether an arbitrary logical formula is satisfiable.

A

Solutions to Exercises

Exercise 2.1: The help page `?diff` tells you how to compute higher order derivatives:

```
diff(sin(x^2), x, x, x, x, x)
```

$$32\,x^5\,\cos\left(x^2\right) - 120\,x\,\cos\left(x^2\right) + 160\,x^3\,\sin\left(x^2\right)$$

You can also use the longer command `diff(diff(...,x),x)`.

Exercise 2.2: The exact representations are:

```
sqrt(27) - 2*sqrt(3), cos(PI/8)
```

$$\sqrt{3},\ \frac{\sqrt{\sqrt{2}+2}}{2}$$

The numerical approximations are:

```
DIGITS := 5:
```

```
float(sqrt(27) - 2*sqrt(3)), float(cos(PI/8))
```

$$1.7321,\ 0.92388$$

They are correct to within 5 digits.

Exercise 2.3:

```
expand((x^2 + y)^5)
```

$$x^{10} + 5\,x^8\,y + 10\,x^6\,y^2 + 10\,x^4\,y^3 + 5\,x^2\,y^4 + y^5$$

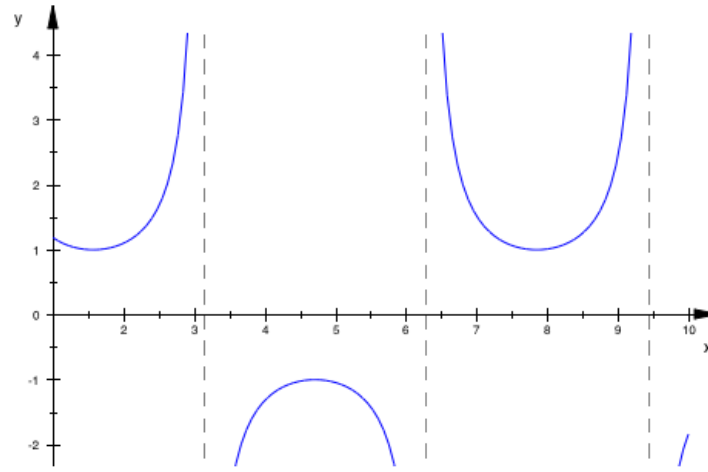
Exercise 2.4:

```
normal((x^2 - 1)/(x + 1))
```

$$x - 1$$

Exercise 2.5: You can plot the singular function $f(x) = 1/\sin(x)$ on the interval $[1, 10]$ without any problems:

```
plot(1/sin(x), x = 1..10)
```



Exercise 2.6: MuPAD® immediately returns the claimed limits:

```
limit(sin(x)/x, x = 0),
limit((1 - cos(x))/x, x = 0),
limit(ln(x), x = 0, Right)
```

$$1,\ 0,\ -\infty$$

```
limit(x^sin(x), x = 0),
limit((1 + 1/x)^x, x = infinity),
limit(ln(x)/exp(x), x = infinity)
```

$$1,\ \mathrm{e},\ 0$$

```
limit(x^ln(x), x = 0, Right),
limit((1 + PI/x)^x, x = infinity),
limit(2/(1 + exp(-1/x)), x = 0, Left)
```

$$\infty,\ e^{\pi},\ 0$$

The result `undefined` denotes a non-existing limit:

```
limit(exp(cot(x)), x = 0)
```

undefined

Exercise 2.7: You obtain the first result in the desired form by factoring:

```
sum(k^2 + k + 1 , k = 1..n): % = factor(%)
```

$$\frac{n^3}{3} + n^2 + \frac{5\,n}{3} = \left(\frac{1}{3}\right) \cdot n \cdot \left(n^2 + 3\,n + 5\right)$$

```
sum((2*k - 3)/((k + 1)*(k + 2)*(k + 3)),
    k = 0..infinity)
```

$$-\frac{1}{4}$$

```
sum(k/(k - 1)^2/(k + 1)^2, k = 2..infinity)
```

$$\frac{5}{16}$$

Exercise 2.8:

```
A := matrix([[1, 2, 3], [4, 5, 6], [7, 8, 0]]):
```

```
B := matrix([[1, 1, 0], [0, 0, 1], [0, 1, 0]]):
```

```
2*(A + B), A*B
```

$$\begin{pmatrix} 4 & 6 & 6 \\ 8 & 10 & 14 \\ 14 & 18 & 0 \end{pmatrix},\ \begin{pmatrix} 1 & 4 & 2 \\ 4 & 10 & 5 \\ 7 & 7 & 8 \end{pmatrix}$$

```
(A - B)^(-1)
```

$$\begin{pmatrix} -\frac{5}{2} & \frac{3}{2} & -\frac{5}{7} \\ \frac{5}{2} & -\frac{3}{2} & \frac{6}{7} \\ -\frac{1}{2} & \frac{1}{2} & -\frac{2}{7} \end{pmatrix}$$

Exercise 2.9: a) The function `numlib::mersenne` returns a list of values for p yielding the 47 currently known Mersenne primes that have been found on supercomputers and/or huge networks. The actual computation for $1 < p \leq 1000$ can be easily performed in MuPAD:

```
select([$ 1..1000], isprime):
```

```
select(%, p -> (isprime(2^p - 1)))
```

After some time you obtain the desired list of values of p:

```
select(%, p -> (isprime(2^p - 1)))
```

$$[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607]$$

The corresponding Mersenne primes are:

```
map(%, p -> (2^p-1))
  [3, 7, 31, 127, 8191, 131071, 524287, 2147483647,

     2305843009213693951, 618970019642690137449562111,

     162259276829213363391578010288127, ... ]
```

b) Depending on your computer's speed, you can test only the first 13 or 14 Fermat numbers in a reasonable amount of time. Note that the 12-th Fermat number already has 1234 decimal digits.

```
Fermat := n -> 2^(2^n) + 1: isprime(Fermat(10))
```

```
   FALSE
```

The only known Fermat primes are the first five Fermat numbers (including `Fermat(0)`). Indeed, if MuPAD tests the first 12 Fermat numbers, then after some time it returns the following five values:

```
select([Fermat(i) $ i = 0..11], isprime)
```

$$[3, 5, 17, 257, 65537]$$

Exercise 4.1: The first operand of a power is the base, the second is the exponent. The first and second operand of an equation is the left and the right-hand side, respectively. The operands of a function call are its arguments:

```
op(a^b, 1), op(a^b, 2)
```

$$a,\ b$$

```
op(a = b, 1), op(a = b, 2)
```

$$a,\ b$$

```
op(f(a, b), 1), op(f(a, b), 2)
```

$$a,\ b$$

Exercise 4.2: The list with the two equations is `op(set, 1)`. Its second operand is the equation `y = ...`, whose second operand is the right-hand side:

```
set := solve({x+sin(3)*y = exp(a),
              y-sin(3)*y = exp(-a)}, {x,y})
```

$$\left\{\left[x = \frac{e^a\ \sin(3) - e^a + \frac{\sin(3)}{e^a}}{\sin(3) - 1}, y = -\frac{1}{e^a\ (\sin(3) - 1)}\right]\right\}$$

```
y := op(set, [1, 2, 2])
```

$$-\frac{1}{e^a\ (\sin(3) - 1)}$$

Use `assign(op(set))` to perform assignments of both unknowns `x` and `y` simultaneously.

Exercise 4.3: If at least one number in a numerical expression is a floating-point number, then the result is a floating-point number:

```
1/3 + 1/3 + 1/3, 1.0/3 + 1/3 + 1/3
```

$$1,\ 1.0$$

Exercise 4.4: You obtain the desired floating-point numbers immediately by applying `float`:

```
float(PI^(PI^PI)), float(exp(PI*sqrt(163)/3))
```

$$1.340164183 \cdot 10^{18},\ 640320.0$$

Note that only the first 10 digits of these values are reliable since this is the default precision. Indeed, for larger values of `DIGITS`, you find:

```
DIGITS := 100:
```

```
float(PI^(PI^PI)), float(exp(PI*sqrt(163)/3))
```

```
  1340164183006357435.297449129640131415099374974573499\
  237787927516586034092619094068148269472611301142

     ,

  640320.000000000604863735049016039471741818818539475 7\
  714857603665918194652218258286942536340815822646
```

We compute 235 decimal digits of `PI` to obtain the correct 234-th digit after the decimal point. After setting `DIGITS:=235`, the result is the last shown digit of `float(PI)`. A more elegant way is to multiply by 10^{234}. Then the desired digit is the first digit before the decimal point, and we obtain it by truncating the digits after the decimal point:

```
DIGITS := 235: trunc(10^234*PI) - 10*trunc(10^233*PI)
```

$$6$$

Exercise 4.5: a) Internally, MuPAD computes with some additional digits not shown in the output.

```
DIGITS := 10: x := 10^50/3.0; floor(x)
```

$$3.333333333 \cdot 10^{49}$$

$$33333333333333333307484730568084080731669827420160$$

b) After increasing DIGITS, MuPAD displays the additional digits. However, not all of them are correct:

```
DIGITS := 40: x
```

$$3.33333333333333333074847305680840807316 7 \cdot 10^{49}$$

Restart the computation with the increased value of DIGITS to obtain the desired precision:

```
DIGITS := 40: x := 10^50/3.0
```

$$3.33333333333333333333333333333333333333 3 \cdot 10^{49}$$

Exercise 4.6: The names caution!-!, x-y, and Jack&Jill are invalid since they contain the special characters !, -, and &, respectively. Since an identifier's name must not start with a number, 2x is not valid either. The names diff and exp are valid names of identifiers. However, you cannot assign values to them since they are protected names of MuPAD functions. The name #1 is a valid name of identifier. However, you cannot assign values to it because, starting with a hash mark, it cannot be assigned to.

Exercise 4.7: We use the sequence operator $ (page 4-24) to generate the set of equations and the set of unknowns. Then a call to solve returns a set of simpler equations:

```
equations := {(x.i + x.(i+1) = 1) $ i = 1..19,
              x20 = PI}:
```

```
unknowns := {x.i $ i = 1..20}:
```

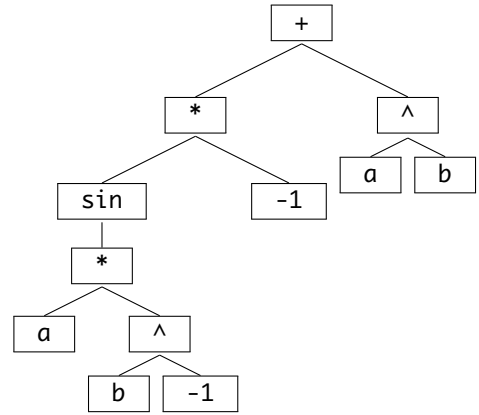
```
solutions := solve(equations, unknowns)
  {[x1 = 1 - PI, x10 = PI, x11 = 1 - PI, x12 = PI,

     x13 = 1 - PI, x14 = PI, x15 = 1 - PI, x16 = PI,

     x17 = 1 - PI, x18 = PI, x19 = 1 - PI, x2 = PI,

     x20 = PI, x3 = 1 - PI, x4 = PI, x5 = 1 - PI,

     x6 = PI, x7 = 1 - PI, x8 = PI, x9 = 1 - PI]}
```

We use the function `assign` to assign the computed values to the identifiers:

```
assign(op(solutions, 1)): x1, x2, x3, x4, x5, x6
```

$$1-\pi,\ \pi,\ 1-\pi,\ \pi,\ 1-\pi,\ \pi$$

Exercise 4.8: MuPAD stores the expression `a^b - sin(a/b)` in the form `sin(a*b^(-1))*(-1)+a^b`. Its expression tree is:



Exercise 4.9: We observe that:

```
op(2/3); op(x/3)
```

$$2,\ 3$$

$$x,\ \frac{1}{3}$$

The reason is that `2/3` is of domain type `DOM_RAT`, whose operands are the numerator and the denominator. The domain type of the symbolic expression `x/3` is `DOM_EXPR` and its internal representation is `x*(1/3)`. The situation is similar for `1+2*I` and `x+2*I`:

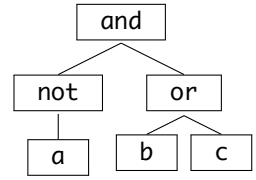
```
op(1 + 2*I); op(x + 2*I)
```

$$1,\ 2$$

$$x,\ 2\mathrm{i}$$

The first object is of domain type DOM_COMPLEX. Its operands are the real and the imaginary part. The operands of the symbolic expression x+2*I are the first and the second term of the sum.

Exercise 4.10: The expression tree of condition=(not a) and (b or c) is:



Thus op(condition, 1)=not a and op(condition, 2)=b or c. We obtain the atoms a, b, c as follows:

```
op(condition, [1, 1]), op(condition, [2, 1]),
op(condition, [2, 2])
```

$$a,\ b,\ c$$

Exercise 4.11: You can use both the assignment function _assign presented on page 4-8 and the assignment operator :=.

```
for i from 1 to 100 do _assign(x.i, i); end:
```

```
for i from 1 to 100 do x.i := i; end:
```

You can also pass a set of assignment equations to the assign function:

```
assign({x.i = i $ i = 1..100}):
```

Exercise 4.12: Since a sequence is a valid argument of the sequence operator, you may achieve the desired result as follows:

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

```
  x1, x2, x2, x3, x3, x3, x4, x4, x4, x4, ...
```

Exercise 4.13: We use the addition function `_plus` and generate its argument sequence via `$`:

```
_plus(((i+j)^(-1) $ j = 1.. i) $ i=1..10)
```

$$\frac{1464232069}{232792560}$$

Exercise 4.14:

```
L1 := [a, b, c, d]: L2 := [1, 2, 3, 4]:
```

```
L1.L2, zip(L1, L2, _mult)
```

$$[a, b, c, d, 1, 2, 3, 4]\,,\ [a, 2\,b, 3\,c, 4\,d]$$

Exercise 4.15: The function `_mult` multiplies its arguments:

```
map([1, x, 2], _mult, multiplier)
```

$$[\text{multiplier}, \text{multiplier}\,x, 2\,\text{multiplier}]$$

We use `map` to apply the function `sublist -> map(sublist, _mult, 2)` to the sublists in a nested list:

```
L := [[1, x, 2], [PI], [2/3, 1]]:
map(L, map, _mult, 2)
```

$$\left[[2, 2\,x, 4]\,,[2\,\pi]\,,\left[\frac{4}{3}, 2\right]\right]$$

Exercise 4.16: For

```
X := [x1, x2, x3]: Y := [y1, y2, y3]:
```

the products are given immediately by

```
_plus(op(zip(X, Y, _mult)))
```

$$\mathrm{x1}\,\mathrm{y1} + \mathrm{x2}\,\mathrm{y2} + \mathrm{x3}\,\mathrm{y3}$$

The following function `f` multiplies each element of the list `Y` by its input parameter `x` and returns the resulting list:

```
f := x -> (map(Y, _mult, x)):
```

The next command replaces each element of `X` by the list returned by `f`:

```
map(X, f)
```

$$[[x1\,y1, x1\,y2, x1\,y3], [x2\,y1, x2\,y2, x2\,y3], [x3\,y1, x3\,y2, x3\,y3]]$$

Exercise 4.17: For each m, we use the sequence generator `$` to create a list of all integers to be checked. Then we extract all primes from the list via `select(·, isprime)`. The number of primes is just `nops` of the resulting list. We compute this value for all m between 0 and 41:

```
nops(select([(n^2 + n + m) $ n = 1..100], isprime))
  $ m = 0..41
  1, 32, 0, 14, 0, 29, 0, 31, 0, 13, 0, 48, 0, 18, 0,

     11, 0, 59, 0, 25, 0, 14, 0, 28, 0, 28, 0, 16, 0,

     34, 0, 35, 0, 11, 0, 24, 0, 36, 0, 17, 0, 86
```

There is a simple explanation for the zero values for even $m > 0$. Since $n^2 + n = n\,(n + 1)$ is always even, $n^2 + n + m$ is an even integer greater than 2 and hence not a prime.

Exercise 4.18: We store the children in a list `C` and remove the one that was counted out at the end of each round. We represent the positions $1, 2, \ldots, n$ of n children by a list with the corresponding integers. Let `out` $\in \{1, \ldots, n\}$ be the position of the last child that was counted out. After deleting the `out`-th element, the next round begins at position `out` in the shortened list. At the end of the round, after m words, the child at position `out+m-1` in the current list is counted out. Since we are counting cyclically, we take this value modulo the number of remaining children. Note, however, that `a mod b` produces numbers in the range $0, 1, \ldots, b - 1$ rather than $1, \ldots, b$. This is overcome by using `((a-1) mod b)+1` instead of `a mod b`:

```
m := 8: n := 12: C := [$ 1..n]: out := 1:
```

```
out := ((out + m - 2) mod nops(C)) + 1:
```

```
C[out]
```

$$8$$

```
delete C[out]:
```

```
out := ((out + m - 2) mod nops(C)) + 1:
```

```
C[out]
```

$$4$$

etc. It is useful to implement the complete counting by a loop (Chapter 15):

```
m := 8: n := 12: C := [$ 1..n]: out := 1:
```

```
repeat
  out := ((out + m - 2) mod nops(C)) + 1:
  print(C[out]);
  delete C[out];
until nops(C) = 0 end_repeat:
```

```
                              8

                              4

                              1

                              11

                             ...

                              9

                              5
```

Exercise 4.19:

```
set := {op(list)}:  list := [op(set)]:
```

Note that the two conversions $list \mapsto set \mapsto list$ in general change the order of the list elements.

Exercise 4.20:

```
A := {a, b, c}: B := {b, c, d}: C := {b, c, e}:
```

```
A union B union C, A intersect B intersect C,
A minus (B union C)
```

$$\{a, b, c, d, e\},\ \{b, c\},\ \{a\}$$

Exercise 4.21: You obtain the union via `_union`:

```
M := {{2, 3}, {3, 4}, {3, 7}, {5, 3}, {1, 2, 3, 4}}:
```

```
_union(op(M))
```

$$\{1, 2, 3, 4, 5, 7\}$$

and the intersection via `_intersect`:

```
_intersect(op(M))
```

$$\{3\}$$

Exercise 4.22:

```
telephoneDirectory := table(Ford = 1815,
  Reagan = 4711, Bush = 1234, Clinton = 5678):
```

An indexed call returns Ford's number:

```
telephoneDirectory[Ford]
```

$$1815$$

You can use `select` to extract all table entries containing the number 5678:

```
select(telephoneDirectory, has, 5678)
```

Clinton	5678

Exercise 4.23: The command `[op(Table)]` returns a list of all assignment equations. The call `map(·, op, i)` ($i = 1, 2$) extracts the left and the right-hand sides, respectively, of the equations:

```
T := table(a = 1, b = 2,
           1 - sin(x) = "derivative of x + cos(x)" ):
```

```
indices := map([op(T)], op, 1)
```

$$[a, b, 1 - \sin(x)]$$

```
values := map([op(T)], op, 2)
```

$$[1, 2, \text{"derivative of x + cos(x)"}]$$

Exercise 4.24: The following timings (in milliseconds) show that generating a table is more time consuming:

```
n := 100000:
```

```
time((T := table((i=i) $ i=1..n))),
time((L := [i $ i=1..n]))
```

$$422,\ 165$$

However, working with tables is notably faster. The following assignments create an additional table entry and extend the list by one element, respectively:

```
time((T[n + 1] := New)), time((L := L.[New]))
```

$$0,\ 84$$

Exercise 4.25: We use the sequence generator `$` to create a nested list and pass it to `array`:

```
n := 20:
array(1..n, 1..n,
  [[1/(i + j - 1) $ j = 1..n] $ i = 1..n]):
```

Exercise 4.26:

```
TRUE and (FALSE or not (FALSE or not FALSE))
```

$$\text{FALSE}$$

Exercise 4.27: We use the function `zip` to generate a list of comparisons. We pass the system function `_less` as third argument, which generates inequalities of the form `a<b`. Then we extract the sequence of inequalities via `op` and pass it to `_and`:

```
L1 := [10*i^2 - i^3 $ i = 1..10]:
L2 := [i^3 + 13*i $ i = 1..10]:
_and(op(zip(L1, L2, _less)))
  9 < 14 and 32 < 34 and 63 < 66 and 96 < 116 and
     125 < 190 and 144 < 294 and 147 < 434 and
     128 < 616 and 81 < 846 and 0 < 1130
```

Finally, evaluating this expression with `bool` answers the question:

```
bool(%)
   TRUE
```

Exercise 4.28: The function `sort` does not sort the identifiers alphabetically by their names but according to an internal order (page 4-28). Thus, we convert them to strings via `expr2text` before sorting:

```
sort(map(anames(All), expr2text))
  ["Ax", "Axiom", "AxiomConstructor", "C_", "Cat",
     "Category", ... , "zeta", "zip"]
```

Exercise 4.29: We compute the reflection of the palindrome

```
text := "Never odd or even":
```

by passing the reflected sequence of individual characters to the function `_concat`, which converts it to a string again:

```
n := length(text):
_concat(text[n - i + 1] $ i = 1..n)
   "neve ro ddo reveN"
```

This can also be achieved with the call `revert(text)`.

Exercise 4.30:

```
f := x -> (x^2): g := x -> (sqrt(x)):
```

```
(f@f@g)(2), (f@@100)(x)
```

$$4,\ x^{1267650600228229401496703205376}$$

Exercise 4.31: The following function does the job:

```
f := L -> [L[nops(L) + 1 - i] $ i = 1..nops(L)]
```

```
    L -> [L[nops(L) + 1 - i] ...
```

```
f([a, b, c])
```

$$[c, b, a]$$

However, the simplest solution is to use `f:=revert`.

Exercise 4.32: You can use the function `last` (Chapter 13.2) to generate the Chebyshev polynomials as expressions:

```
T0 := 1: T1 := x:
```

```
T2 := 2*x*% - %2; T3:= 2*x*% - %2; T4:= 2*x*% - %2
```

$$2\,x^2 - 1$$

$$2\,x\,\left(2\,x^2 - 1\right) - x$$

$$1 - 2\,x^2 - 2\,x\,\left(x - 2\,x\,\left(2\,x^2 - 1\right)\right)$$

A much more elegant way is to translate the recursive definition into a MuPAD function that works recursively:

```
T := (k, x) ->
     (if k < 2
         then x^k
         else 2*x*T(k - 1, x) - T(k - 2, x)
      end_if):
```

Then we obtain:

```
T(i, 1/3) $ i = 2..5
```

$$-\frac{7}{9},\ -\frac{23}{27},\ \frac{17}{81},\ \frac{241}{243}$$

```
T(i, 0.33) $ i = 2..5
```

$$-0.7822,\ -0.846252,\ 0.22367368,\ 0.9938766288$$

```
T(i, x) $ i = 2..5
     2              2
  2 x  - 1, 2 x (2 x  - 1) - x,

            2                    2
     1 - 2 x  - 2 x (x - 2 x (2 x  - 1)), x -

             2                                 2
     2 x (2 x  - 1) - 2 x (2 x (x - 2 x (2 x  - 1)) +

        2
     2 x  - 1)
```

You can obtain expanded representations of the polynomials by inserting a call to `expand` (page 9-3) in the function definition.

The Chebyshev polynomials are already implemented in `orthpoly`, the library for orthogonal polynomials. The i-th Chebyshev polynomial is returned by the call `orthpoly::chebyshev1(i, x)`.

Exercise 4.33: In principle, you can compute the derivatives of f in MuPAD and substitute $x = 0$. But it is simpler to approximate the function by a Taylor series whose leading terms describe the behavior in the neighborhood of $x = 0$:

```
taylor(tan(sin(x)) - sin(tan(x)), x = 0)
```

$$\frac{x^7}{30} + \frac{29\,x^9}{756} + \frac{1913\,x^{11}}{75600} + O\left(x^{13}\right)$$

Thus, $f(x) = x^7/30 \cdot (1 + O(x^2))$, and hence f has a root of order 7 at $x = 0$.

Exercise 4.34: The reason for the difference between the results

```
taylor(diff(1/(1 - x), x), x);
diff(taylor(1/(1 - x), x), x)
```

$$1 + 2\,x + 3\,x^2 + 4\,x^3 + 5\,x^4 + 6\,x^5 + O\left(x^6\right)$$

$$1 + 2\,x + 3\,x^2 + 4\,x^3 + 5\,x^4 + O\left(x^5\right)$$

is the truncation determined by the environment variable `ORDER` with the default value 6. Both `taylor` calls compute the corresponding series up to $O(x^6)$:

```
taylor(1/(1 - x), x)
```

$$1 + x + x^2 + x^3 + x^4 + x^5 + O\left(x^6\right)$$

The order term `O(x^5)` appears when the term `O(x^6)` is differentiated:

```
diff(%, x)
```

$$1 + 2\,x + 3\,x^2 + 4\,x^3 + 5\,x^4 + O\left(x^5\right)$$

Exercise 4.35: An asymptotic expansion yields:

```
f := sqrt(x + 1) - sqrt(x - 1):
g := series(f, x = infinity)
```

$$\frac{1}{\sqrt{x}} + \frac{1}{8\,x^{\frac{5}{2}}} + \frac{7}{128\,x^{\frac{9}{2}}} + O\left(\frac{1}{x^{\frac{11}{2}}}\right)$$

Thus

$$f \approx \frac{1}{\sqrt{x}}\left(1 + \frac{1}{8\,x^2} + \frac{7}{128\,x^4} + \cdots\right),$$

and hence $f(x) \approx 1/\sqrt{x}$ for all real $x \gg 1$. The next better approximation is $f(x) \approx \frac{1}{\sqrt{x}}\left(1 + \frac{1}{8\,x^2}\right)$.

Exercise 4.36: The command `?revert` requests the corresponding help page.

```
f := taylor(sin(x + x^3), x); g := revert(%)
```

$$x + \frac{5\,x^3}{6} - \frac{59\,x^5}{120} + O\left(x^7\right)$$

$$x - \frac{5\,x^3}{6} + \frac{103\,x^5}{40} + O\left(x^7\right)$$

To check this result, we consider the composition of f and g, whose series expansion is that of the identity function $x \mapsto x$:

```
g@f
```

$$x + O\left(x^7\right)$$

Exercise 4.37: We perform the computation over the standard coefficient ring (page 4-64), which comprises both rational numbers and floating-point numbers:

```
n := 16:
```

```
H := matrix(n, n, (i, j) -> ((i + j -1)^(-1))):
```

```
e := matrix(n, 1, 1): b := H*e:
```

We first compute the solution of the system of equations $H\,\vec{x} = \vec{b}$ with exact arithmetic over the rational numbers. Then we convert all entries of H and $\vec{b}$ to floating-point numbers and solve the system numerically (and in the unstable and slow way, using an explicit inverse instead of calling `numeric::matlinsolve`):

```
exact = H^(-1)*b, numerical = float(H)^(-1)*float(b)
```

$$\text{exact} = \begin{pmatrix} 1 \\ 1 \\ \cdots \\ 1 \\ 1 \\ 1 \end{pmatrix},\ \text{numerical} = \begin{pmatrix} 0.999999993 \\ 1.000000164 \\ \cdots \\ -11.875 \\ 4.036132812 \\ 0.3975830078 \end{pmatrix}$$

The errors in the numerical solution originate from rounding errors. To demonstrate this, we repeat the numerical computation with higher precision:

```
DIGITS := 20:
```

```
numerical = float(H)^(-1)*float(b)
```

$$\text{numerical} = \begin{pmatrix} 1.0000000505004147319 \\ 0.99999992752642441474 \\ \cdots \\ 0.99999889731407165527 \\ 1.0000003278255462646 \\ 0.9999999580904841423 \end{pmatrix}$$

Exercise 4.38: We look for values where the determinant of the matrix vanishes:

```
matrix([[1, a, b], [1, 1, c], [1, 1, 1]]):
```

```
factor(linalg::det(%))
```

$$(c-1)\cdot(a-1)$$

The matrix is invertible unless $a = 1$ or $c = 1$.

Exercise 4.39: We first store the matrix data in arrays. These arrays are used later to generate matrices over different coefficient rings:

```
a := array(1..3, 1..3, [[ 1, 3, 0],
                        [-1, 2, 7],
                        [ 0, 8, 1]]):
```

```
b := array(1..3, 1..2, [[7, -1], [2, 3], [0, 1]]):
```

Now we define the constructor `MQ` for matrices over the rational numbers and convert the arrays to corresponding matrices:

```
MQ := Dom::Matrix(Dom::Rational): A := MQ(a): B := MQ(b):
```

The method "transpose" of the constructor computes the transpose of a matrix B via MQ::transpose(B):

```
(2*A + B*MQ::transpose(B))^(-1)
```

$$\begin{pmatrix} \frac{34}{1885} & \frac{7}{1508} & -\frac{153}{7540} \\ \frac{11}{3770} & -\frac{31}{3016} & \frac{893}{15080} \\ -\frac{47}{3770} & \frac{201}{3016} & -\frac{731}{15080} \end{pmatrix}$$

Computing over the residue class ring modulo 7 we find:

```
Mmod7 := Dom::Matrix(Dom::IntegerMod(7)):
```

```
A := Mmod7(a): B := Mmod7(b):
```

```
C := (2*A + B*Mmod7::transpose(B)): C^(-1)
```

$$\begin{pmatrix} 3 \bmod 7 & 0 \bmod 7 & 1 \bmod 7 \\ 1 \bmod 7 & 3 \bmod 7 & 2 \bmod 7 \\ 4 \bmod 7 & 2 \bmod 7 & 2 \bmod 7 \end{pmatrix}$$

We check this by multiplying the inverse by the original matrix, which yields the identity matrix over the coefficient ring:

```
%*C
```

$$\begin{pmatrix} 1 \bmod 7 & 0 \bmod 7 & 0 \bmod 7 \\ 0 \bmod 7 & 1 \bmod 7 & 0 \bmod 7 \\ 0 \bmod 7 & 0 \bmod 7 & 1 \bmod 7 \end{pmatrix}$$

Exercise 4.40: We compute over the coefficient ring of rational numbers:

```
MQ := Dom::Matrix(Dom::Rational):
```

We consider 3×3 matrices. To define the matrix, we pass a function to the constructor that maps the indices to the corresponding matrix entries:

```
A := MQ(3, 3, (i, j) -> (if i=j then 0 else 1 end_if))
```

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

The determinant of A is

```
linalg::det(A)
```

$$2$$

The eigenvalues are the roots of the characteristic polynomial:

```
p := linalg::charpoly(A, x)
```

$$x^3 - 3\,x - 2$$

```
solve(p, x)
```

$$\{-1, 2\}$$

Alternatively, the `linalg` package provides a function for computing eigenvalues:

```
linalg::eigenvalues(A)
```

$$\{-1, 2\}$$

Let *Id* denote the 3×3 identity matrix. The eigenspace for the eigenvalue $\lambda \in \{-1, 2\}$ is the solution space of the system of linear equations $(A - \lambda \cdot \mathit{Id})\,\vec{x} = \vec{0}$. The solution vectors span the nullspace (the "kernel") of the matrix $A - \lambda \cdot \mathit{Id}$. The function `linalg::nullspace` computes a basis for the kernel of a matrix:

```
Id := MQ::identity(3):
```

```
lambda := -1: linalg::nullspace(A - lambda*Id)
```

$$\left[\begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}\right]$$

There are two linearly independent basis vectors. Hence the eigenspace for the eigenvalue $\lambda = -1$ is two-dimensional. The other eigenvalue is simple:

```
lambda := 2: linalg::nullspace(A - lambda*Id)
```

$$\left[\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\right]$$

Alternatively, `linalg::eigenvectors` computes all eigenspaces simultaneously:

```
linalg::eigenvectors(A)
```

$$\left[\left[2, 1, \left[\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\right]\right], \left[-1, 2, \left[\begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}\right]\right]\right]$$

The return value is a nested list. For each eigenvalue λ, it contains a list of the form

$$[\ \lambda\,,\ \text{multiplicity of } \lambda\,,\ \text{eigenspace basis}\ ].$$

Exercise 4.41:

```
p := poly(x^7 - x^4 + x^3 - 1): q := poly(x^3 - 1):
```

```
p - q^2
```

$$\text{poly}\left(x^7 - x^6 - x^4 + 3\,x^3 - 2, [x]\right)$$

The polynomial p is a multiple of q:

```
p/q
```

$$\text{poly}\left(x^4 + 1, [x]\right)$$

This is confirmed by a factorization:

```
factor(p)
```

$$\text{poly}\left(x - 1, [x]\right) \cdot \text{poly}\left(x^2 + x + 1, [x]\right) \cdot \text{poly}\left(x^4 + 1, [x]\right)$$

```
factor(q)
```

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

Exercise 4.42: We use the identifier R to abbreviate the lengthy type name Dom::IntegerMod(3). With alias, MuPAD also uses R as an alias in the output of the following polynomials.

```
p := 3: alias(R = Dom::IntegerMod(p)):
```

We only need to try the possible remainders $0, 1, 2$ modulo 3 for the coefficients a, b, c in $a\,x^2 + b\,x + c$. We generate a list of all 18 quadratic polynomials with $a \neq 0$ as follows:

```
[((poly(a*x^2 + b*x + c, [x], R) $ a = 1..p-1)
   $ b = 0..p-1) $ c = 0..p-1]:
```

The command select(·, irreducible) extracts the 6 irreducible polynomials:

```
select(%, irreducible)
                                    2
          2                        2
  [poly(x  + 1, [x], R), poly(2 x  + x + 1, [x], R),

             2
    poly(2 x  + 2 x + 1, [x], R),

             2                      2
    poly(2 x  + 2, [x], R), poly(x  + x + 2, [x], R),

          2
    poly(x  + 2 x + 2, [x], R)]
```

Exercise 5.1: The value of x is the identifier a1. The evaluation of x yields the identifier c1. The value of y is the identifier b2. The evaluation of y yields the identifier c2. The value of z is the identifier a3. The evaluation of z yields 10.

The evaluation of u1 leads to an infinite recursion, which MuPAD aborts with an error message. The evaluation of u2 yields the expression v2^2 - 1.

Exercise 6.1: The result of subsop(b+a, 1=c) is b+c and not c+a, as you might have expected. The reason is that subsop evaluates its arguments. The system reorders the sum internally when evaluating it, and thus subsop processes a+b instead of b+a. Upon return, the result c+b is reordered again.

Exercise 6.2: The highest derivative occurring in `g` is the 6-th derivative `diff(f(x), x$6)`. We pass the sequence of replacement equations:

```
diff(f(x), x $ 6) = f6, diff(f(x), x $ 5) = f5, ...,
                  diff(f(x), x) = f1, f(x) = f0
```

to MuPAD's substitution function. Note that, in accordance with the usual mathematical notation, `diff` returns the function itself as the 0-th derivative: `diff(f(x), x$0) = diff(f(x)) = f(x)`.

```
delete f: g := diff(f(x)/diff(f(x), x), x $ 5):
```

```
subs(g, (diff(f(x), x $ 6 - i) = f.(6-i)) $ i = 0..6)
```

$$\frac{60\,f2^4}{f1^4} - \frac{4\,f5}{f1} + \frac{20\,f3^2}{f1^2} - \frac{120\,f0\,f2^5}{f1^6} - \frac{100\,f2^2\,f3}{f1^3} -$$

$$\frac{f0\,f6}{f1^2} + \frac{25\,f2\,f4}{f1^2} + \frac{10\,f0\,f2\,f5}{f1^3} + \frac{20\,f0\,f3\,f4}{f1^3} -$$

$$\frac{90\,f0\,f2^3\,f3}{f1^4} + \frac{240\,f0\,f2^2\,f3}{f1^5} - \frac{60\,f0\,f2^2\,f4}{f1^4}$$

Exercise 7.1: The following commands yield the desired evaluation of the function:

```
f := sin(x)/x: x := 1.23: f
```

$$0.7662510585$$

However, `x` now has a value. The following call `diff(f, x)` would internally lead to the command `diff(0.7662510584, 1.23)`, since `diff` evaluates its arguments.

You can circumvent this problem by preventing a complete evaluation of the arguments via `level` or `hold` (page 5-4):

```
g := diff(level(f, 1), hold(x)); g
```

$$\frac{\cos(x)}{x} - \frac{\sin(x)}{x^2}$$

$$-0.3512303507$$

Here the evaluation of `hold(x)` is the identifier `x` and not its value. Writing `hold(f)` instead of `level(f, 1)` would yield the wrong result `diff(hold(f),hold(x))=0`, since `hold(f)` does not contain `hold(x)`. Using `level(f, 1)` replaces `f` by its value `sin(x)/x` (page 5-4). The next call of `g` returns the evaluation of `g`, namely the value of the derivative at `x=1.23`. Alternatively you can delete the value of `x`:

```
delete x: diff(f, x) | x = 1.23
```

$$-0.3512303507$$

Exercise 7.2: The first three derivatives of the numerator and the denominator vanish at the point $x = 0$:

```
Z := x -> (x^3*sin(x)): N := x -> ((1 - cos(x))^2):
```

```
Z(0), N(0), Z'(0), N'(0), Z''(0), N''(0),
Z'''(0), N'''(0)
```

$$0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0$$

For the fourth derivatives, we have:

```
Z''''(0), N''''(0)
```

$$24,\ 6$$

Thus the limit is $Z''''(0)/N''''(0) = 4$, according to de l'Hospital's rule. The function `limit` computes the same result:

```
limit(Z(x)/N(x), x = 0)
```

$$4$$

Exercise 7.3: The first order partial derivatives of f_1 are:

```
f1 := sin(x1*x2): diff(f1, x1), diff(f1, x2)
```

$$x2\cos(x1\,x2),\ x1\cos(x1\,x2)$$

The second order derivatives are:

```
diff(f1, x1, x1), diff(f1, x1, x2),
diff(f1, x2, x1), diff(f1, x2, x2)
```

```
      2
  - x2  sin(x1 x2), cos(x1 x2) - x1 x2 sin(x1 x2),

                                          2
    cos(x1 x2) - x1 x2 sin(x1 x2), - x1  sin(x1 x2)
```

The total derivative of f_2 with respect to t is:

```
f2 := x^2*y^2: x := sin(t): y := cos(t): diff(f2, t)
```

$$2\cos(t)^3\sin(t) - 2\cos(t)\sin(t)^3$$

Exercise 7.4:

```
int(sin(x)*cos(x), x = 0..PI/2),
int(1/(sqrt(1 - x^2)), x = 0..1),
int(x*arctan(x), x = 0..1),
int(1/x, x = -2..-1)
```

$$\frac{1}{2},\ \frac{\pi}{2},\ \frac{\pi}{4} - \frac{1}{2},\ -\ln(2)$$

Exercise 7.5:

```
int(x/(2*a*x - x^2)^(3/2), x)
```

$$\frac{x}{a\sqrt{2\,a\,x - x^2}}$$

```
int(sqrt(x^2 - a^2), x)
```

$$\frac{x\sqrt{x^2 - a^2}}{2} - \frac{a^2\ln\left(x + \sqrt{x^2 - a^2}\right)}{2}$$

```
int(1/(x*sqrt(1 + x^2)), x)
```

$$\ln(x) - \ln\left(\sqrt{x^2+1}+1\right)$$

Exercise 7.6: The function `intlib::changevar` only performs a change of variables without invoking the integration:

```
intlib::changevar(hold(int)(sin(x)*sqrt(1 + sin(x)),
                  x = -PI/2..PI/2), t = sin(x))
```

$$\int_{-1}^{1} \frac{t\sqrt{t+1}}{\sqrt{1-t^2}}\,\mathrm{d}t$$

A further evaluation activates the integration routine:

```
eval(%): % = float(%)
```

$$\frac{2\sqrt{2}}{3} = 0.9428090416$$

Numerical quadrature returns the same result:

```
numeric::int(sin(x)*sqrt(1 + sin(x)),
             x = -PI/2..PI/2)
```

$$0.9428090416$$

Exercise 8.1: The equation solver returns the general solution:

```
equations := {a +   b +   c +   d +   e = 1,
              a + 2*b + 3*c + 4*d + 5*e = 2,
              a - 2*b - 3*c - 4*d - 5*e = 2,
              a -   b -   c -   d -   e = 3}:
```

```
solve(equations, {a, b, c, d, e})
```

$$\{[a = 2, b = z + 2\,\mathrm{z1} - 3, c = 2 - 3\,\mathrm{z1} - 2\,z, d = z, e = \mathrm{z1}]\}$$

The free parameters are on the right-hand sides of the solved equations. You can determine them in MuPAD by extracting the right hand sides and using `indets` to find the identifiers contained therein:

```
map(%, map, op, 2); indets(%)
```

$$\{[2, z + 2\,\mathrm{z1} - 3, 2 - 3\,\mathrm{z1} - 2\,z, z, \mathrm{z1}]\}$$

$$\{z, \mathrm{z1}\}$$

Exercise 8.2: The symbolic solution is:

```
solution := solve(ode(
  {y'(x) = y(x) + 2*z(x), z'(x) = y(x)}, {y(x), z(x)}))
```

$$\left\{\left[z(x) = \frac{\mathrm{C2}\,\mathrm{e}^{2\,x}}{2} - \frac{\mathrm{C1}}{\mathrm{e}^{x}}, y(x) = \frac{\mathrm{C1}}{\mathrm{e}^{x}} + \mathrm{C2}\,\mathrm{e}^{2\,x}\right]\right\}$$

with free constants `C1`, `C2`. We remove the outer curly braces via `op`:

```
solution := op(solution)
```

$$\left[z(x) = \frac{\mathrm{C2}\,\mathrm{e}^{2\,x}}{2} - \frac{\mathrm{C1}}{\mathrm{e}^{x}}, y(x) = \frac{\mathrm{C1}}{\mathrm{e}^{x}} + \mathrm{C2}\,\mathrm{e}^{2\,x}\right]$$

Now, we set $x = 0$ and substitute `y(0)` and `z(0)`, respectively, for the initial conditions. Then, we solve the resulting linear system of equations for `C1` and `C2`:

```
solve((solution | x = 0) | [y(0) = 1, z(0) = 1],
      {C1, C2})
```

$$\left\{\left[\mathrm{C1} = -\frac{1}{3}, \mathrm{C2} = \frac{4}{3}\right]\right\}$$

Again, we remove the outer curly braces via `op` and assign the solution values to `C1` and `C2` by means of `assign`:

```
assign(op(%)):
```

Thus, the value at $x = 1$ of the symbolic solution for the above initial conditions is:

```
x := 1: solution
```

$$\left[z(1) = \frac{1}{3\,\mathrm{e}} + \frac{2\,\mathrm{e}^{2}}{3}, y(1) = \frac{4\,\mathrm{e}^{2}}{3} - \frac{1}{3\,\mathrm{e}}\right]$$

Finally, we apply `float`:

```
float(%)
```

$$[z(1) = 5.04866388, y(1) = 9.729448318]$$

Exercise 8.3:

```
solve(ode(y'(x)/y(x)^2 = 1/x, y(x)))
```

$$\left\{-\frac{1}{\mathrm{C3} + \ln(x)}\right\}$$

```
solve(ode({y'(x) - sin(x)*y(x) = 0, y'(1)=1}, y(x)))
```

$$\left\{\frac{\mathrm{e}^{\cos(1)}}{\mathrm{e}^{\cos(x)}\ \sin(1)}\right\}$$

```
solve(ode({2*y'(x) + y(x)/x = 0, y'(1) = PI}, y(x)))
```

$$\left\{-\frac{2\,\pi}{\sqrt{x}}\right\}$$

```
solve(ode({diff(x(t),t) = y(t)*z(t),
           diff(y(t),t) = x(t)*z(t),
           diff(z(t),t) = t*z(t)},
          {x(t),y(t),z(t)}))
```

```
  { --
  { |
  { |  x(t) =
  { --

              1/2       1/2      1/2
     C11 exp(2    C10 PI    erf(2    t 1/2 I) (-1/2 I))
     ---------------------------------------------------
                             2

                 1/2       1/2      1/2
       C12 exp(2    C10 PI    erf(2    t 1/2 I) 1/2 I)
      + ----------------------------------------------,
                              2

     y(t) =

              1/2       1/2      1/2
     C11 exp(2    C10 PI    erf(2    t 1/2 I) (-1/2 I))
     ---------------------------------------------------
                             2

                 1/2       1/2      1/2
       C12 exp(2    C10 PI    erf(2    t 1/2 I) 1/2 I)
      - ----------------------------------------------,
                              2

                   /  2 \ -- }
                   | t  |  | }
     z(t) = C10 exp| -- |  | }
                   \  2 / -- }
```

Exercise 8.4: The function `solve` directly yields the solution of the recurrence:

```
solve(rec(F(n) = F(n-1) + F(n-2), F(n),
          {F(0) = 0, F(1) = 1}))
```

$$\left\{ \frac{\sqrt{5}\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)^{n}}{5} - \frac{\sqrt{5}\left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)^{n}}{5} \right\}$$

Exercise 9.1:

You get the answer by applying `combine` to rewrite products of trigonometric functions as sums:

```
combine(cos(x)^2 + sin(x)*cos(x), sincos)
```

$$\frac{\cos(2\,x)}{2} + \frac{\sin(2\,x)}{2} + \frac{1}{2}$$

Exercise 9.2:

```
expand(cos(5*x)/(sin(2*x)*cos(x)^2))
```

$$\frac{\cos(x)^2}{2\,\sin(x)} - 5\,\sin(x) + \frac{5\sin(x)^3}{2\cos(x)^2}$$

```
f := (sin(x)^2 - exp(2*x)) /
     (sin(x)^2 + 2*sin(x)*exp(x) + exp(2*x)):
```

```
normal(expand(f))
```

$$-\frac{\mathrm{e}^{x} - \sin(x)}{\mathrm{e}^{x} + \sin(x)}$$

```
f := (sin(2*x) - 5*sin(x)*cos(x)) /
     (sin(x)*(1 + tan(x)^2)):
```

```
combine(normal(expand(f)), sincos)
```

$$-\frac{3\,\cos(x)}{\tan(x)^2 + 1}$$

```
f := sqrt(14 + 3*sqrt(3 +
                  2*sqrt(5 - 12*sqrt(3 - 2*sqrt(2))))):
```

```
simplify(f, sqrt)
```

$$\sqrt{2}+3$$

Exercise 9.3: As a first step, we perform a normalization:

```
int(sqrt(sin(x) + 1), x): normal(diff(%, x))
```

$$\frac{3\cos(x)^2\,\sin(x)+\cos(x)^2+2\sin(x)^3-2\,\sin(x)}{\cos(x)^2\,\sqrt{\sin(x)+1}}$$

Then we eliminate the cosine terms:

```
rewrite(%, sin)
```

$$\frac{2\,\sin(x)+\sin(x)^2-2\sin(x)^3+3\,\sin(x)\,\left(\sin(x)^2-1\right)-1}{\sqrt{\sin(x)+1}\,\left(\sin(x)^2-1\right)}$$

The final normalization step achieves the desired simplification:

```
normal(%)
```

$$\sqrt{\sin(x)+1}$$

On the other hand, calling `Simplify` is much easier:

```
int(sqrt(sin(x) + 1), x): Simplify(diff(%, x))
```

$$\sqrt{\sin(x)+1}$$

Exercise 9.4: The problem obviously is that the valuation function does not look carefully enough at the term to analyze. To remedy this, we use the function `length` which returns a general, quickly computed "complexity:"

```
noTangent := x -> if has(x, [hold(tan), hold(cot)])
                  then 1000000
                  else length(x) end_if:
Simplify(tan(x) - cot(x), Valuation = noTangent)
```

$$-\frac{2\cos(2\,x)}{\sin(2\,x)}$$

Exercise 9.5: The operator `assuming` (page 9-19) assigns properties to identifiers. These are taken into account by `limit`:

```
limit(Ei(a*x), x = infinity)
```

$$\lim_{x\to\infty} \mathrm{Ei}(a\,x)$$

```
limit(Ei(a*x), x = infinity) assuming a > 0
```

$$\infty$$

```
limit(Ei(a*x), x = infinity) assuming a < 0
```

$$0$$

Exercise 10.1: In analogy to the previous gcd example, we obtain the following experiment:

```
die := random(1..6):
```

```
experiment := [[die(), die(), die()] $ i = 1..216]:
```

```
diceScores := map(experiment,
                  x -> (x[1] + x[2] + x[3])):
```

```
frequencies := Dom::Multiset(op(diceScores)):
```

```
sortingOrder := (x, y) -> (x[1] < y[1]):
```

```
sort([op(frequencies)], sortingOrder)
```

```
  [[4, 4], [5, 9], [6, 8], [7, 9], [8, 16], [9, 20],

     [10, 27], [11, 31], [12, 32], [13, 20], [14, 13],

     [15, 12], [16, 6], [17, 7], [18, 2]]
```

In this experiment, the score 3 did not occur.

Exercise 10.2: a) The command `frandom` is a generator for random floating-point numbers in the interval $[0, 1]$. The call

```
n := 1000:
absValues := [sqrt(frandom()^2+frandom()^2) $ i = 1..n]:
```

returns a list with the absolute values of n random vectors in the rectangle $Q = [0, 1] \times [0, 1]$. The number of values ≤ 1 is the number of random points in the right upper quadrant of the unit circle:

```
m := nops(select(absValues, z -> (z <= 1)))
```

$$821$$

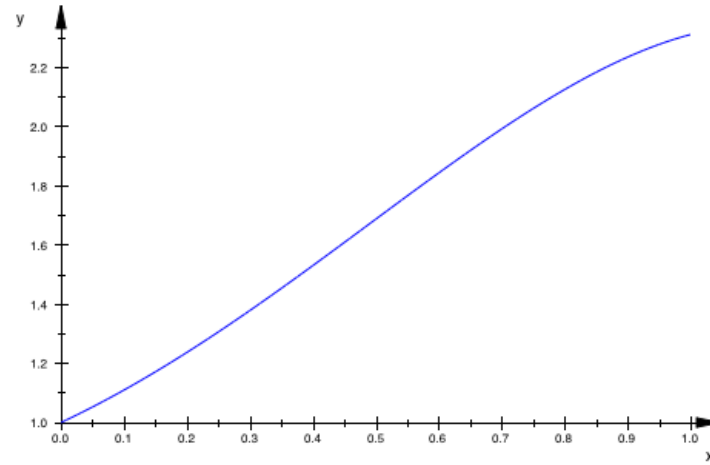
Since `m/n` approximates the area $\pi/4$ of the right upper quadrant of the unit circle, we obtain the following approximation to π:

```
float(4*m/n)
```

$$3.284$$

b) First we determine the maximum of f. The following function plot shows that f is monotonically increasing on the interval $[0, 1]$:

```
f := x -> x*sin(x) + cos(x)*exp(x):
plotfunc2d(f(x), x = 0..1):
```



Thus $f(x)$ assumes its maximal value at the right end of the interval. Therefore, $M = f(1)$ is an upper bound for the function:

```
M := f(1.0)
```

$$2.310164925$$

We use the random number generator defined above to generate random points in the rectangle $[0, 1] \times [0, M]$:

```
n := 1000:
pointlist := [[frandom(), M*frandom()] $ i = 1..n]:
```

We select those points $p = [x, y]$ for which $0 \leq y \leq f(x)$ holds:

```
select(pointlist, p -> (p[2] <= f(p[1]))):
```

```
m := nops(%)
```

$$742$$

Thus the following is an approximation of the integral:

```
m/n*M
```

$$1.714142374$$

The exact value is:

```
float(int(f(x), x = 0..1))
```

$$1.679193292$$

Exercise 13.1: With the following definition of the `postOutput` method of `Pref`, the system prints an additional status line:

```
Pref::postOutput(
  proc()
  begin
    "bytes: " .
    expr2text(op(bytes(), 1)) . " (logical) / " .
    expr2text(op(bytes(), 2)) . " (physical)"
   end_proc):
```

```
float(sum(1/i!, i = 0..100))
```

$$2.718281828$$

```
 bytes: 836918 (logical) / 1005938 (physical)
```

Exercise 14.1: We first generate the set S:

```
f := i -> (i^(5/2)+i^2-i^(1/2)-1) /
          (i^(5/2)+i^2+2*i^(3/2)+2*i+i^(1/2)+1):
```

```
S := {f(i) $ i=-1000..-2} union {f(i) $ i=0..1000}:
```

Then we apply `domtype` to all elements of the set to determine their domain types:

```
map(S, domtype)
```

$$\{\text{DOM\_RAT}, \text{DOM\_INT}, \text{DOM\_EXPR}\}$$

You see the explanation for this result by looking at some elements:

```
f(-2), f(0), f(1), f(2), f(3), f(4)
```

$$\frac{(-2)^{\frac{5}{2}}+3-\sqrt{2}\,\mathrm{i}}{2\,(-2)^{\frac{3}{2}}+(-2)^{\frac{5}{2}}+1+\sqrt{2}\,\mathrm{i}},\ -1,\ 0,\ \frac{3\sqrt{2}+3}{9\sqrt{2}+9},\ \frac{8\sqrt{3}+8}{16\sqrt{3}+16},\ \frac{3}{5}$$

The function `normal` simplifies the expressions containing square roots:

```
map(%, normal)
```

$$3,\ -1,\ 0,\ \frac{1}{3},\ \frac{1}{2},\ \frac{3}{5}$$

Now we apply `normal` to all elements of the set before querying their data type:

```
map(S, domtype@normal)
```

$$\{\text{DOM\_RAT}, \text{DOM\_INT}\}$$

Thus, all numbers in S are indeed rational (in particular there are two integer values `f(0)=-1` and `f(1)=0`). The reason is that `f(i)` can be simplified to `(i-1)/(i+1)`:

```
normal(f(i))
```

$$\frac{i-1}{i+1}$$

Exercise 14.2: We apply `testtype(·, "sin")` to each element of the list:

```
list := [sin(i*PI/200) $ i = 0..100]:
```

to find out whether it is returned in the form `sin(·)`. The following `split` command (page 4-36) decomposes the list accordingly:

```
decomposition := split(list, testtype, "sin"):
```

MuPAD simplified 9 of the 101 calls:

```
map(decomposition, nops); decomposition[2]
```

```
    [92, 9, 0]

  --     1/2              1/2 1/2   1/2       1/2 1/2
  |     5           (2 - 2   )     2    (5 - 5   )
  |  0, ---- - 1/4, -------------, ------------------,
  --      4               2                4

      1/2   1/2          1/2     1/2
     2     5           (2    + 2)
     ----, ---- + 1/4, -------------,
       2     4               2

      1/2   1/2     1/2    --
     2    (5    + 5)        |
     -----------------, 1  |
              4            --
```

Exercise 14.3: You can use `select` (page 4-36) to extract those elements that `testtype` identifies as positive integers. For example:

```
set := {-5, 2.3, 2, x, 1/3, 4}:
```

```
select(set, testtype, Type::PosInt)
```

$$\{2, 4\}$$

Note that this selects only those objects that *are* positive integers, but not those that might *represent* positive integers, such as the identifier `x` in the above example. This is not possible with `testtype`. Instead, you can use `assume` to set this property and query it via `is`:

```
assume(x, Type::PosInt):
```

```
select(set, is, Type::PosInt)
```

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

Exercise 14.4: We construct the desired type specifier and employ it as follows:

```
T := Type::ListOf(Type::ListOf(
       Type::AnyType, 3, 3), 2, 2)
```

$$\text{Type::ListOf}(\text{Type::ListOf}(\text{Type::AnyType}, 3, 3), 2, 2)$$

```
testtype([[a, b, c], [1, 2, 3]], T),
testtype([[a, b, c], [1, 2]], T)
```

$$\text{TRUE}, \text{FALSE}$$

Exercise 16.1: Consider the conditions `x <> 1 and A` and `x = 1 or A`, respectively. After entering:

```
x := 1:
```

it is not possible to evaluate them due to the singularity in $x/(x-1)$:

```
x <> 1 and A
  Error: Division by zero [_power]
```

```
x = 1 or A
  Error: Division by zero [_power]
```

However, this is not a problem within an `if` statement since the Boolean evaluation of `x <> 1` and `x = 1` already tells us that `x <> 1 and A` evaluates to `FALSE` and `x = 1 or A` to `TRUE`, respectively:

```
(if x <> 1 and A then right else wrong end_if),
(if x = 1 or A then right else wrong end_if)
```

$$\text{wrong}, \text{right}$$

On the other hand, evaluation of the following `if` statement still produces an error, since it is necessary to evaluate `A` in order to determine the truth value of `x = 1 and A`:

```
if x = 1 and A then right else wrong end_if
  Error: Division by zero [_power]
```

Exercise 17.1: With the formulas given, implementation of the "float" slots is straightforward:

```
ellipE::float := z -> float(PI/2) *
  hypergeom::float([-1/2, 1/2], [1], z):
ellipK::float := z -> float(PI/2) *
  hypergeom::float([1/2, 1/2], [1], z):
```

This is almost sufficient:

```
ellipE(1/3)
```

$$E\left(\frac{1}{3}\right)$$

```
float(%)
```

$$1.430315257$$

However, one thing is missing: we would like to have calls with floating point arguments evaluated immediately:

```
ellipE(0.1)
```

$$E(0.1)$$

To this end, we must extend the definitions of ellipE and ellipK:

```
ellipE :=
 proc(x) begin
   if domtype(x) = DOM_FLOAT
     or domtype(x) = DOM_COMPLEX
        and (domtype(Re(x)) = DOM_FLOAT
             or domtype(Im(x)) = DOM_FLOAT)
   then
     return(ellipE::float(x));
   end_if;
   if x = 0 then PI/2
   elif x = 1 then 1
   else procname(x) end_if
 end_proc:
```

Extend ellipK analogously, then make these new functions into function environments (the above definition replaced the old one completely) and add the function slots we used for the original definitions and the new float slots, and you get:

```
ellipE(0.1)
```

$$1.530757637$$

Exercise 17.2: The following procedure evaluates the `Abs` function:

```
Abs := proc(x)
begin
   if domtype(x) = DOM_INT or domtype(x) = DOM_RAT
      or domtype(x) = DOM_FLOAT
      then if x >= 0 then x else -x end_if;
      else procname(x);
   end_if
end_proc:
```

We convert `Abs` to a function environment and supply a function producing the desired screen output:

```
Abs := funcenv(Abs,
               proc(f) begin
                  "|" . expr2text(op(f)) . "|"
               end_proc,
               NIL):
```

Then we set the function attribute for differentiation:

```
Abs::diff := proc(f,x) begin
               f/op(f)*diff(op(f), x)
            end_proc:
```

Now we have the following behavior:

```
Abs(-23.4), Abs(x), Abs(x^2 + y - z)
```

$$23.4,\ |x|,\ |x^2 + y - z|$$

The `diff` attribute of the system function `abs` yields a slightly different but equivalent result:

```
diff(Abs(x^3), x), diff(abs(x^3), x)
```

$$\frac{3\,|x^3|}{x},\ 3\,|x|\ \mathrm{sign}(x)$$

Exercise 17.3: We use `expr2text` (page 4-49) to convert the integers passed as arguments to strings. Then we combine them, together with some slashes, via the concatenation operator ".":

```
date := proc(month, day, year) begin
            print(Unquoted, expr2text(month) . "/" .
                            expr2text(day) . "/" .
                            expr2text(year))
        end_proc:
```

Exercise 17.4: We present a solution using a `while` loop. The condition `x mod 2=0` checks whether `x` is even:

```
f := proc(x) local i;
begin
  i := 0;
  userinfo(2, "term " . expr2text(i) . ": " .
              expr2text(x));
  while x <> 1 do
    if x mod 2 = 0 then x := x/2
    else x := 3*x+1 end_if;
    i := i + 1;
    userinfo(2, "term " . expr2text(i) . ": " .
                expr2text(x))
  end_while;
  i
end_proc:
```

```
f(4), f(1234), f(56789), f(123456789)
```

$$2,\ 132,\ 60,\ 177$$

If we set `setuserinfo(f, 2)` (page 13-9), then the `userinfo` command outputs all terms of the sequence until the procedure terminates:

```
setuserinfo(f, 2): f(4)
```

```
 Info: term 0: 4
 Info: term 1: 2
 Info: term 2: 1
```

$$2$$

If you do not believe in the $3\,x + 1$ conjecture, you should insert a stopping condition for the index `i` to ensure termination.

Exercise 17.5: A recursive implementation based on the relation $\gcd(a, b) = \gcd(a \bmod b, b)$ leads to an infinite recursion: we have $a \bmod b \in \{0, 1, \ldots, b-1\}$, and hence

$$(a \bmod b) \bmod b = a \bmod b$$

in the next step. Thus the function gcd would always call itself recursively with the same arguments. However, a recursive call of the form $\gcd(a, b) = \gcd(b, a \bmod b)$ make sense. Since $a \bmod b < b$, the function calls itself recursively for decreasing values of the second argument, which finally becomes zero:

```
Gcd := proc(a, b) begin     /* recursive variant */
           if b = 0
              then a
              else Gcd(b, a mod b)
           end_if
       end_proc:
```

For large values of a and b, you may need to increase the value of the environment variable `MAXDEPTH` if `Gcd` exhausts the valid recursion depth. The following iterative variant avoids this problem:

```
GCD := proc(a, b)
       begin
         while b <> 0 do
           [a, b] := [b, a mod b];
         end_while;
         a
       end_proc:
```

These implementations yield the same results as the functions `igcd` and `gcd` provided by the system:

```
a := 123456: b := 102880:
```

```
Gcd(a, b), GCD(a, b), igcd(a, b), gcd(a, b)
```

$$20576,\ 20576,\ 20576,\ 20576$$

Exercise 17.6: In the following implementation, we generate a shortened copy $Y = [x_1, \ldots, x_n]$ of $X = [x_0, \ldots, x_n]$ and compute the list of differences $[x_1 - x_0, \ldots, x_n - x_{n-1}]$ via `zip` and `_subtract` (`_subtract(y,x)=y-x`). Then we multiply each element of this list with the corresponding numerical value in the list $[f(x_0), f(x_1), \ldots]$. Finally, the function `_plus` adds all elements of the resulting list:

$$[(x_1 - x_0)\, f(x_0), \ldots, (x_n - x_{n-1})\, f(x_{n-1})]\,:$$

```
Quadrature := proc(f, X)
local Y, distances, numericalValues, products;
begin
   Y := X; delete Y[1];
   distances := zip(Y, X, _subtract);
   numericalValues := map(X, float@f);
   products := zip(distances, numericalValues, _mult);
   _plus(op(products))
end_proc:
```

In the following example, we use $n = 1000$ equidistant sample points in the interval $[0, 1]$:

```
f := x -> (x*exp(x)): n := 1000:
```

```
Quadrature(f, [i/n $ i = 0..n])
```

$$0.9986412288$$

This is a (crude) numerical approximation of $\int_0^1 x\, e^x\, dx\ (= 1)$.

Exercise 17.7: The specification of `Newton` requires that the first argument `f` be an *expression* and not a MuPAD function. Thus, to compute the derivative, we first use `indets` to determine the unknown in `f`. We substitute a numerical value for the unknown to evaluate the iteration function $F(x) = x - f(x)/f'(x)$ at a point:

```
Newton := proc(f, x0, n)
  local vars, x, F, sequence, i;
  begin
    vars := indets(float(f)):
    if nops(vars) <> 1
       then error(
     "the function must contain exactly one unknown"
                 )
       else x := op(vars)
    end_if;
    F := x - f/diff(f,x); sequence := x0;
    for i from 1 to n do
        x0 := float(subs(F, x = x0));
        sequence := sequence, x0
    end_for;
    return(sequence)
  end_proc:
```

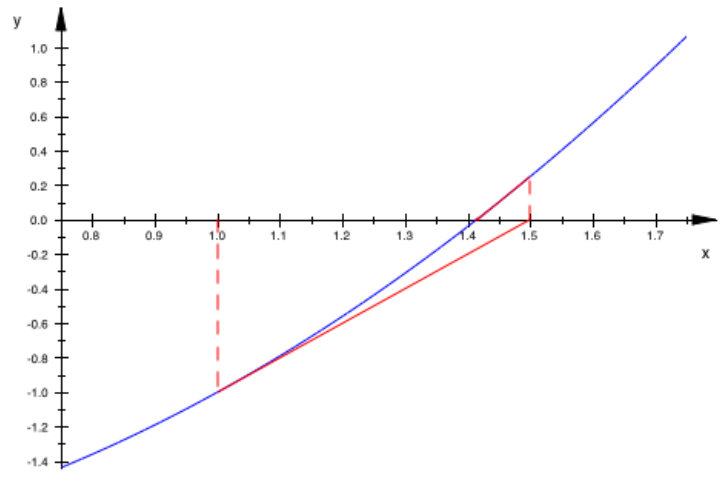
In the following example, `Newton` computes the first terms of a sequence rapidly converging to the solution $\sqrt{2}$:

```
Newton(x^2 - 2, 1, 6)
  1, 1.5, 1.416666667, 1.414215686, 1.414213562,

     1.414213562, 1.414213562
```

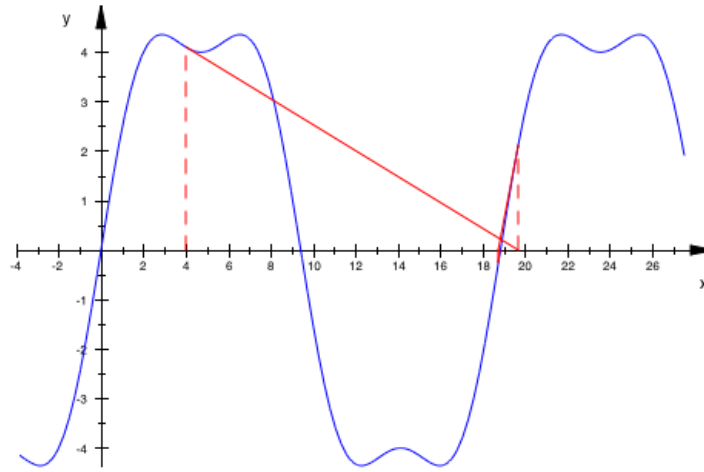
The following procedure uses `Newton` to plot the approximation procedure:

```
plotNewton :=
proc(f, x0, n)
  local x, xvals, xmin, xmax, i, xi;
begin
  xvals := Newton(f, x0, n);
  xmin := min(xvals);
  xmax := max(xvals);
  [xmin, xmax] :=
    [xmin - (xmax - xmin)/2, xmax + (xmax - xmin)/2];
  x := op(indets(f));
  plot(plot::Function2d(f, x=xmin..xmax),
    plot::Line2d([xvals[i], f | x=xvals[i]],
      [xvals[i+1], 0]) $ i = 1..nops(xvals)-1,
    plot::Line2d([xvals[i], 0], [xvals[i], f | x=xvals[i]],
      LineStyle = Dashed) $ i = 1..nops(xvals),
    plot::Line2d::LineColor = RGB::Red)
end_proc:
```

```
plotNewton(x^2 - 2, 1, 3)
```



```
plotNewton(sin(x)+5*sin(x/3), 4, 3)
```



Exercise 17.8: The call `numlib::g_adic(..,2)` yields the binary expansion of an integer as a list of bits:

```
numlib::g_adic(7, 2), numlib::g_adic(16, 2)
```

$$[1,1,1],\ [0,0,0,0,1]$$

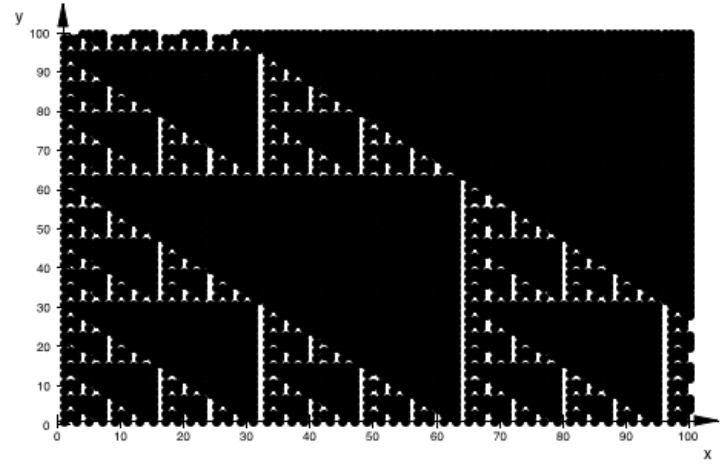
Instead of calling `numlib::g_adic` directly, our solution uses a subprocedure `binary` furnished with the `option remember`. This accelerates the computation notably since `numlib::g_adic` is called frequently with the same arguments. The call `isSPoint([x, y])` returns `TRUE` when the point specified by the list `[x, y]` is a Sierpinski point. To check this, the function multiplies the lists with the bits of the two coordinates. At those positions where both `x` and `y` have a 1 bit, multiplication yields a 1. In all other cases, $0 \cdot 0$, $1 \cdot 0$, $0 \cdot 1$, produce the result 0. If the list of products contains at least one 1, the point is a Sierpinski point.

We use `select` (page 4-28) to extract the Sierpinski points from all points considered. Finally, we create a `plot::PointList2d` with these points and call `plot` (page 11-1):

```
Sierpinski := proc(xmax, ymax)
local binary, isSPoint, allPoints, i, j, SPoints;
begin
  binary := proc(x) option remember; begin
              numlib::g_adic(x, 2)
            end_proc;
  isSPoint := proc(Point) local x, y; begin
                x := binary(Point[1]);
                y := binary(Point[2]);
                has(zip(x, y, _mult), 1)
              end_proc;
  allPoints := [([i, j] $ i = 1..xmax) $ j = 1..ymax];
  SPoints := select(allPoints, isSPoint);
  plot(plot::PointList2d(SPoints, Color = RGB::Black));
end_proc:
```

For `xmax = ymax = 100`, say, you obtain a quite appealing picture:

```
Sierpinski(100, 100)
```



Exercise 17.9: We present a recursive solution. Given an expression `formula(x1, x2, ...)` with the identifiers `x1, x2, ...`, we collect the identifiers in the set `x = {x1, x2, ...}` by means of `indets`. Then we substitute `TRUE` and `FALSE`, respectively, for `x1` (= `op(x, 1)`), and call the procedure recursively with the arguments `formula(TRUE, x2, x3, ...)` and `formula(FALSE, x2, x3, ...)`, respectively. In this way, we test all possible combinations of `TRUE` and `FALSE` for the identifiers until the expression can finally be simplified to `TRUE` or `FALSE` at the bottom of the recursion. This value is returned to the calling procedure. If at least one of the `TRUE`/`FALSE` combinations yields `TRUE`, then the procedure returns `TRUE`, indicating that the formula is satisfiable, and otherwise it returns `FALSE`.

```
satisfiable := proc(formula) local x;
begin
  x := indets(formula);
  if x = {} then return(formula) end_if;
  return(satisfiable(subs(formula, op(x, 1) = TRUE))
      or satisfiable(subs(formula, op(x, 1) = FALSE)))
end_proc:
```

If the number of identifiers in the input formula is n, then the recursion depth is at most n and the total number of recursive calls of the procedure is at most 2^n. We apply this procedure in two examples:

```
F1 := ((x and y) or (y or z)) and (not x) and y and z:
```

```
F2 := ((x and y) or (y or z)) and (not y) and (not z):
```

```
satisfiable(F1), satisfiable(not F1),
satisfiable(F2), satisfiable(not F2)
```

$$\text{TRUE},\ \text{TRUE},\ \text{FALSE},\ \text{TRUE}$$

The call `simplify(·, logic)` (page 9-13) simplifies logical formulae. Formula `F2` can be simplified to false, no matter what the values of x, y, and z are:

```
simplify(F1, logic), simplify(F2, logic)
```

$$\neg x \wedge y \wedge z,\ \text{FALSE}$$

B

Documentation and References

You can find a list of all documents that are available in your installation by choosing “Browse Help” from the “Help” menu of any MuPAD® notebook window and then clicking on “Contents”.

The MuPAD Quick Reference lists all MuPAD data types, functions, and libraries, and provides a survey of its functionality.

You also find links to various libraries such as, for example, `Dom` (the library for pre-installed data types). The corresponding documentation contains a concise description of all domains provided by `Dom`. In a MuPAD session, the command `?Dom` directly opens this document. Moreover, you can access the description of individual data structures from this document, such as `Dom::Matrix`, directly through the call `?Dom::Matrix`. Another example is the documentation for the `linalg` package (linear algebra). It can be requested directly via `?linalg`.

As an introduction to MuPAD, targeted especially at Windows® users, we recommend the following books:

[Maj 05] M. Majewski *Getting Started with MuPAD*. Springer Heidelberg, 2005. ISBN 3-540-28635-7

[Maj 04] M. Majewski *MuPAD Pro Computing Essentials, Second Edition*. Springer Heidelberg, 2004. ISBN 3-540-21943-9

In addition to the MuPAD documentation, we recommend the following books about computer algebra in general and the underlying algorithms:

[Wes 99] M. Wester (ed.), *Computer Algebra Systems. A Practical Guide*. Wiley, 1999.

[GG 99] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*. Cambridge University Press, 1999.

[Hec 93] A. Heck, *Introduction to Maple*. Springer, 1993.

[DTS 93] J.H. Davenport, E. Tournier and Y. Siret, *Computer Algebra: Systems and Algorithms for Algebraic Computation*. Academic Press, 1993.

[GCL 92] K.O. Geddes, S.R. Czapor and G. Labahn, *Algorithms for Computer Algebra*. Kluwer, 1992.

C

Graphics Gallery

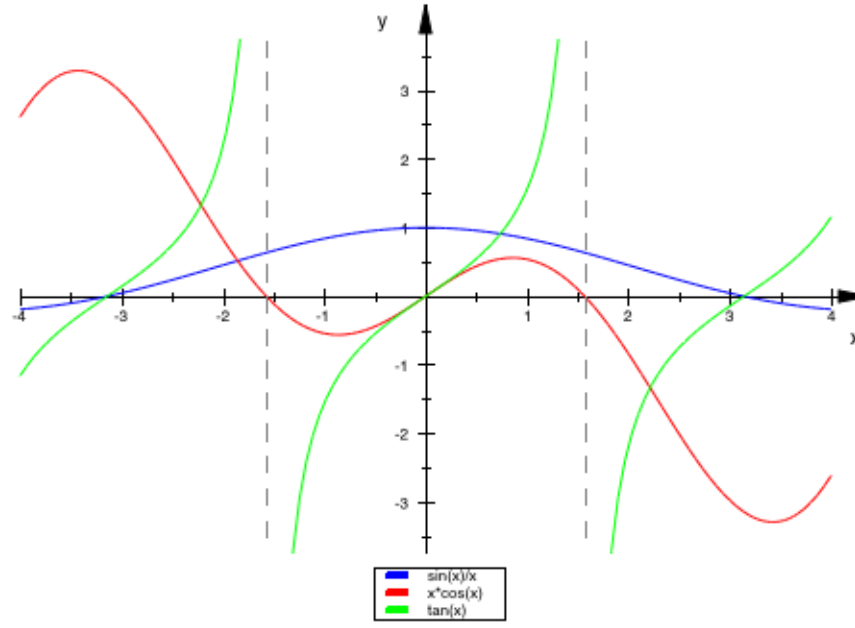
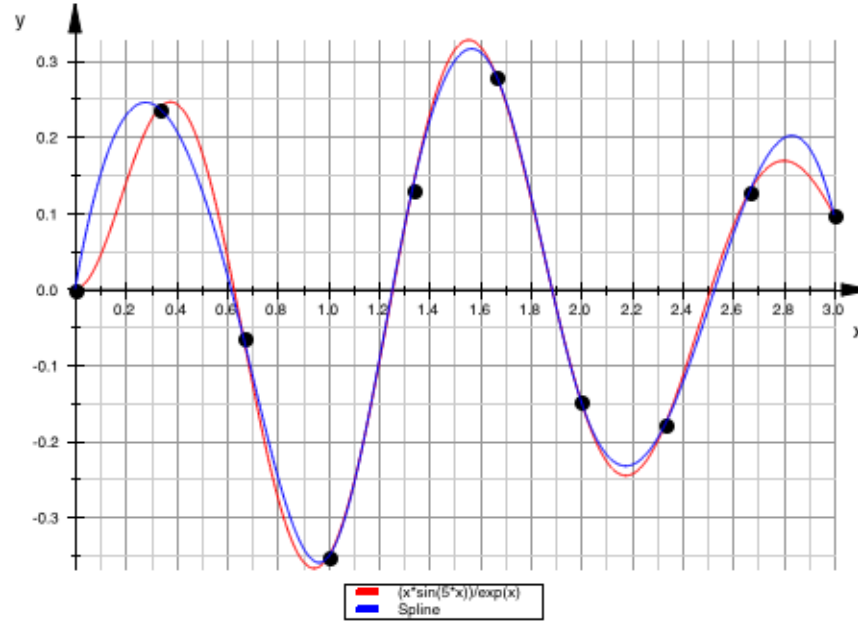


Fig. 1: Functions



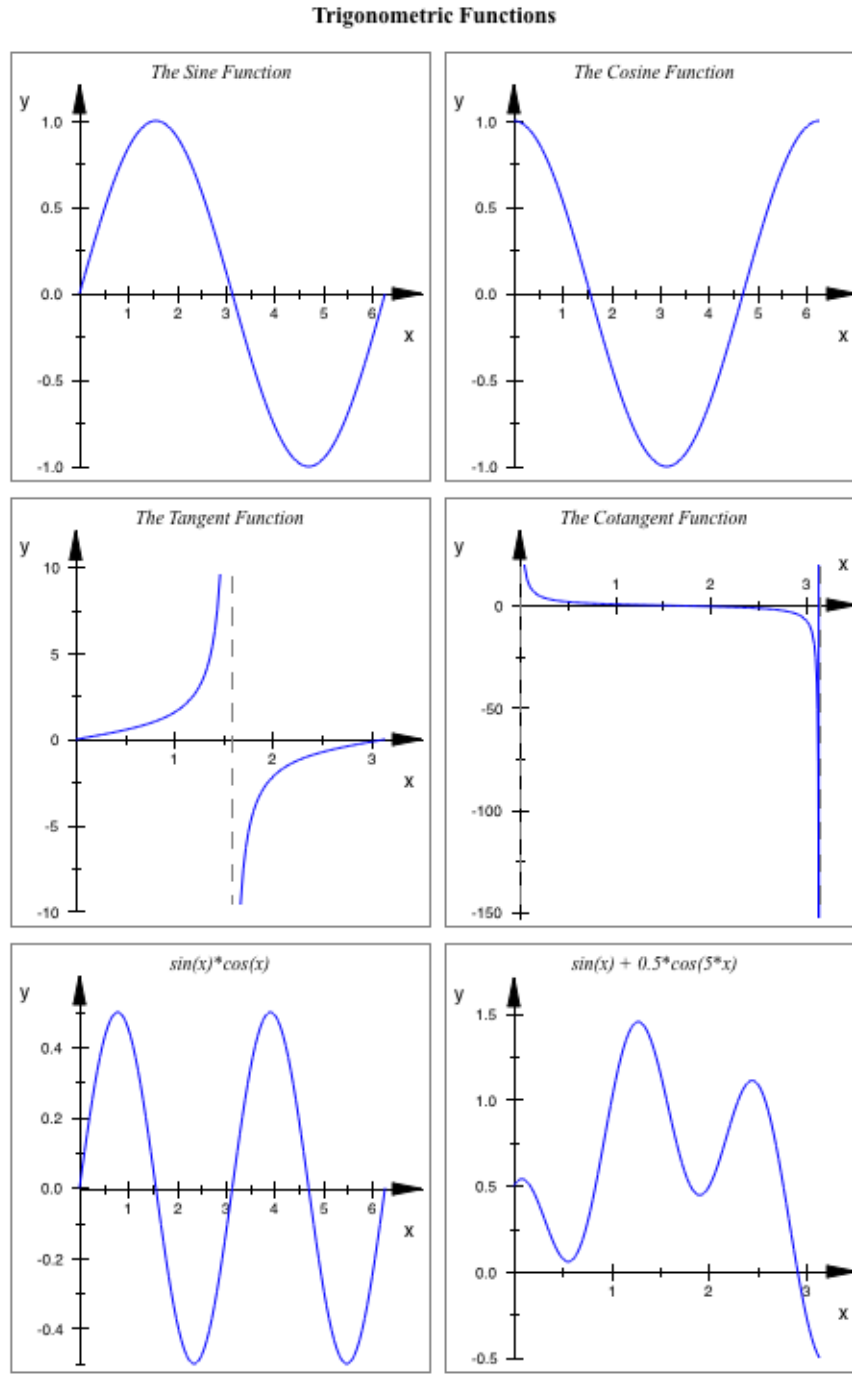


Fig. 3: A Canvas with several Scenes

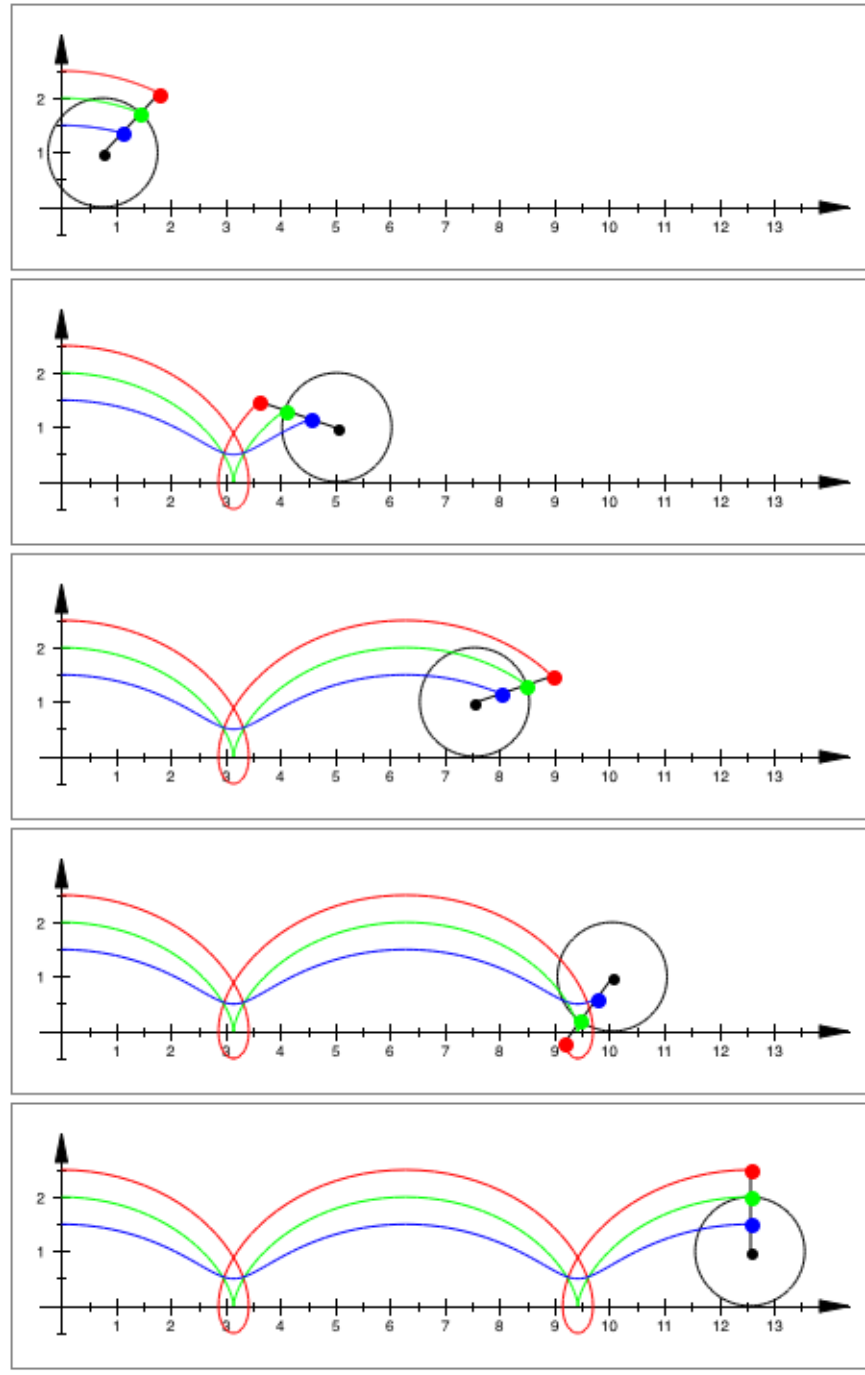


Fig. 4: Construction of Cycloids

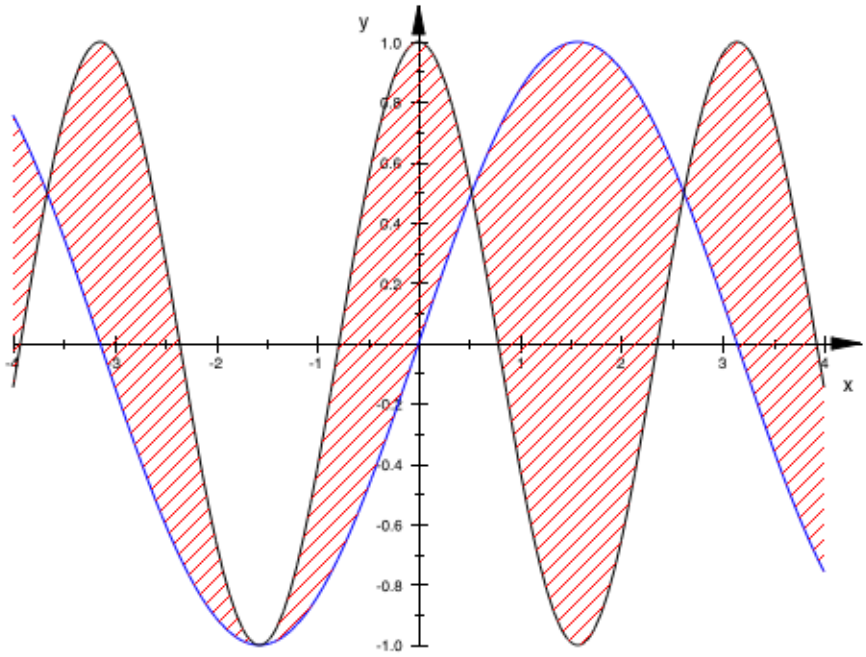


Fig. 5: Hatch between Functions

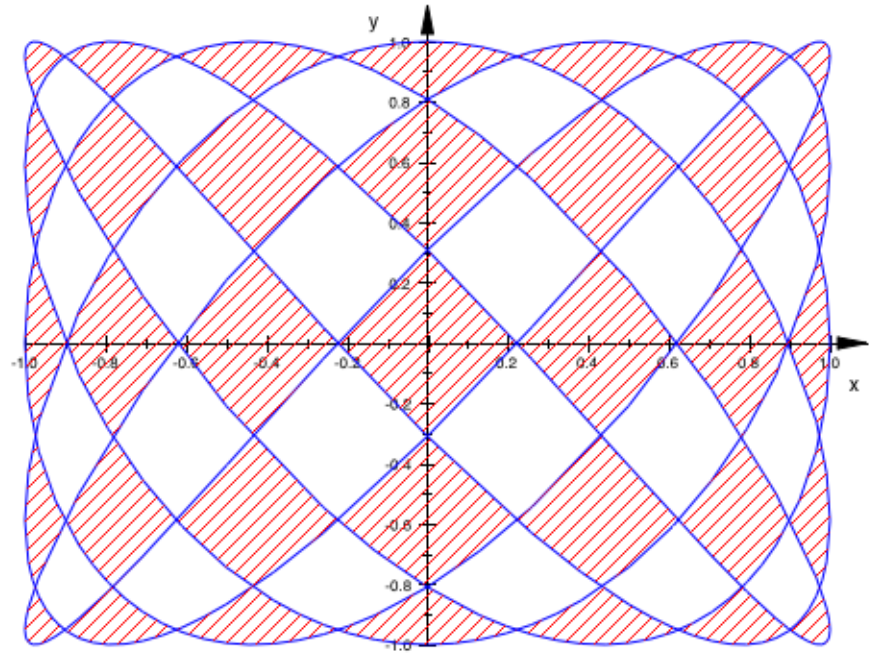


Fig. 6: Hatch inside Curves

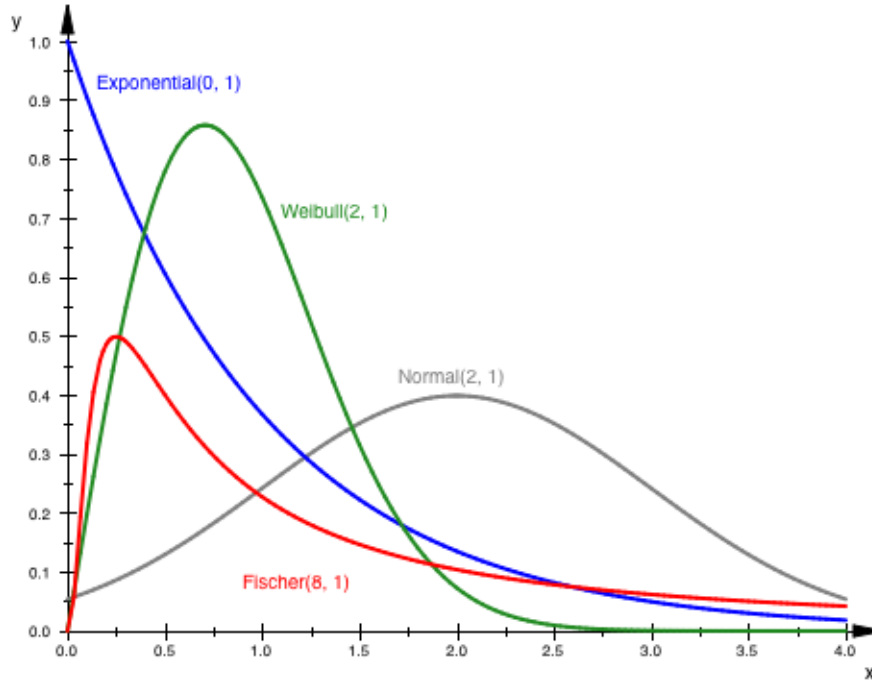


Fig. 7: Statistical Distributions

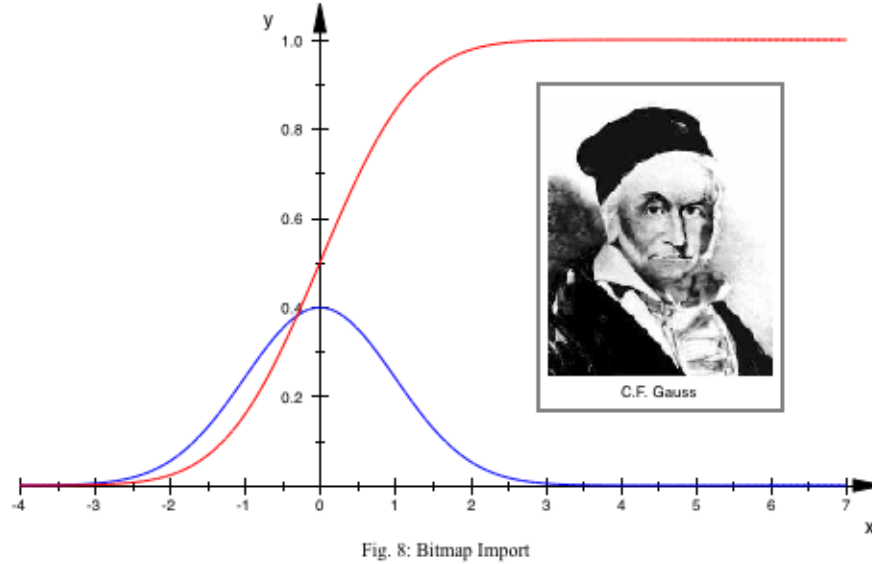


Fig. 8: Bitmap Import

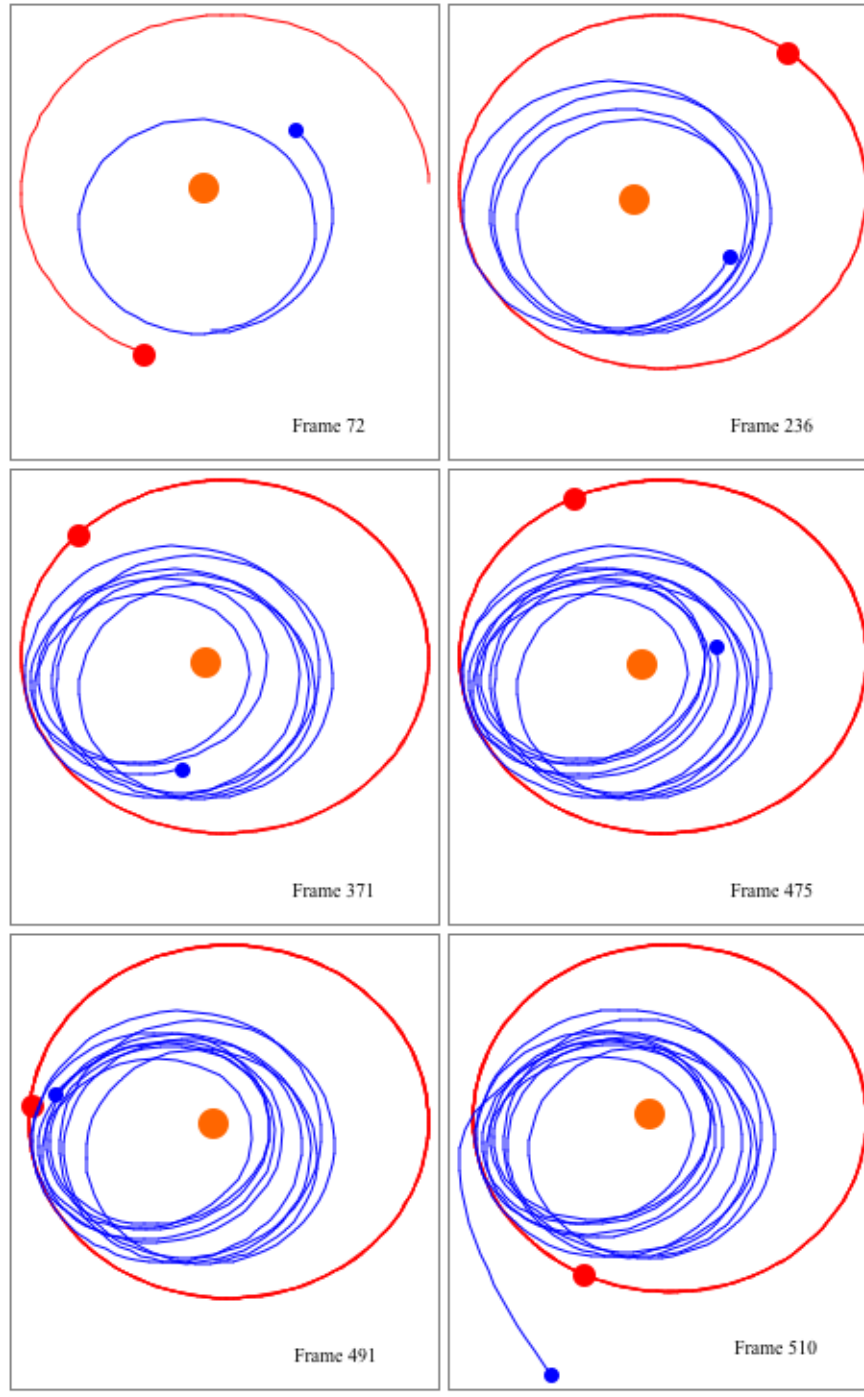


Fig. 9: Three Body Problem

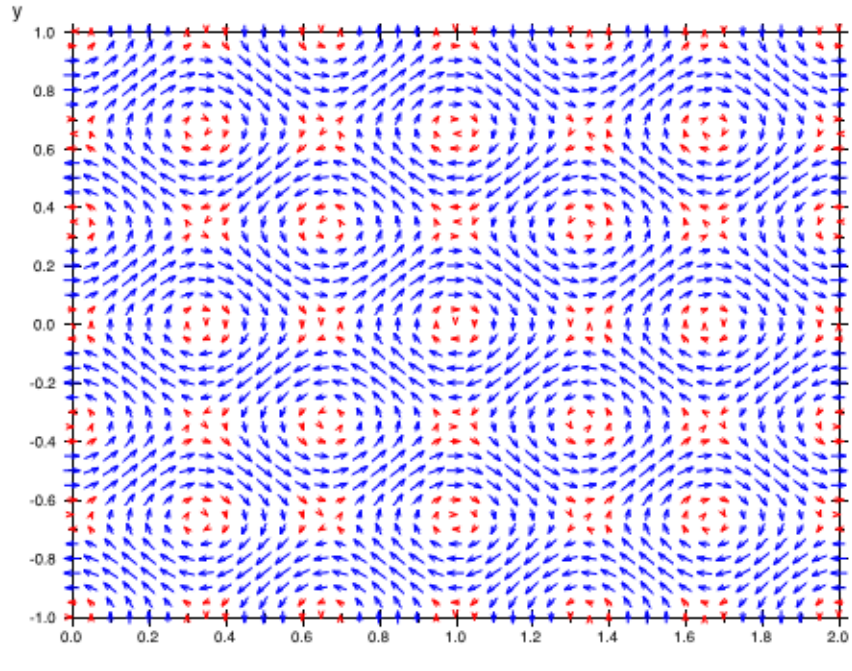


Fig. 10: Visualization of a Vector Field

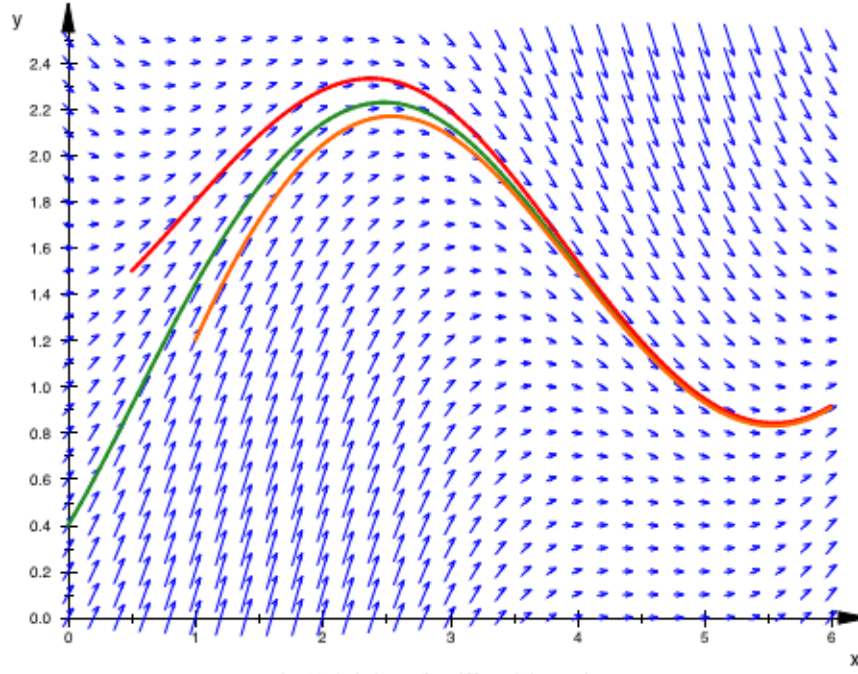


Fig. 11: Solutions of a Differential Equation

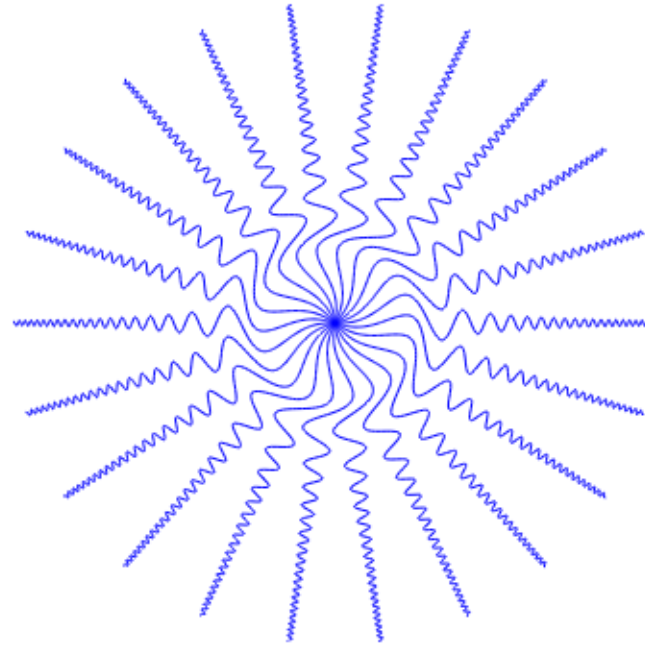


Fig. 12: Rotation of a Function

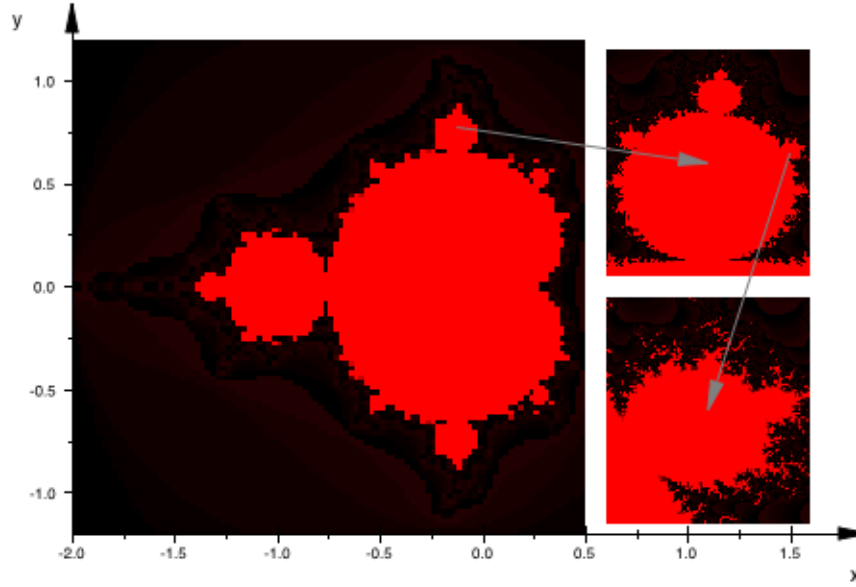


Fig. 13: Mandelbrot Set

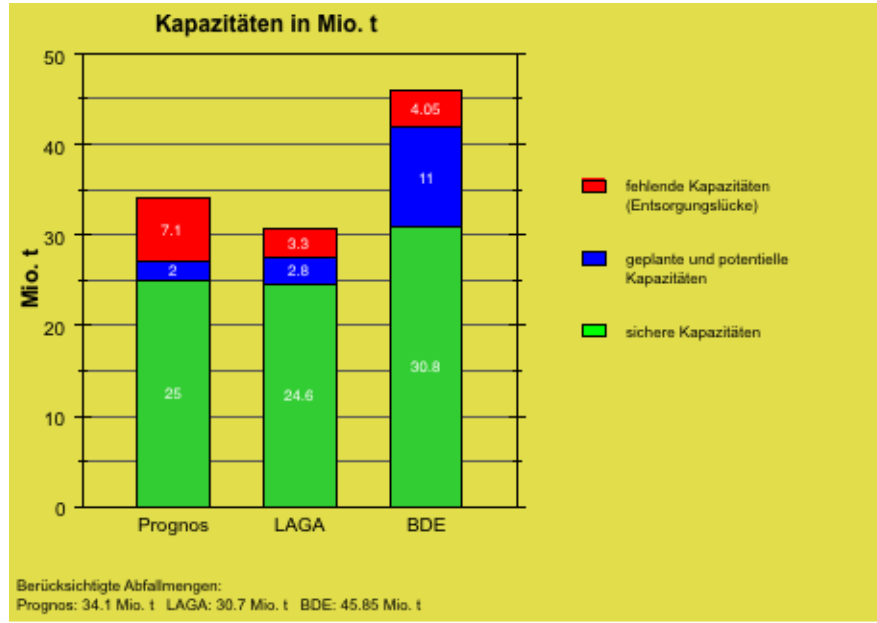


Fig. 14: Advanced bar plot of statistical data

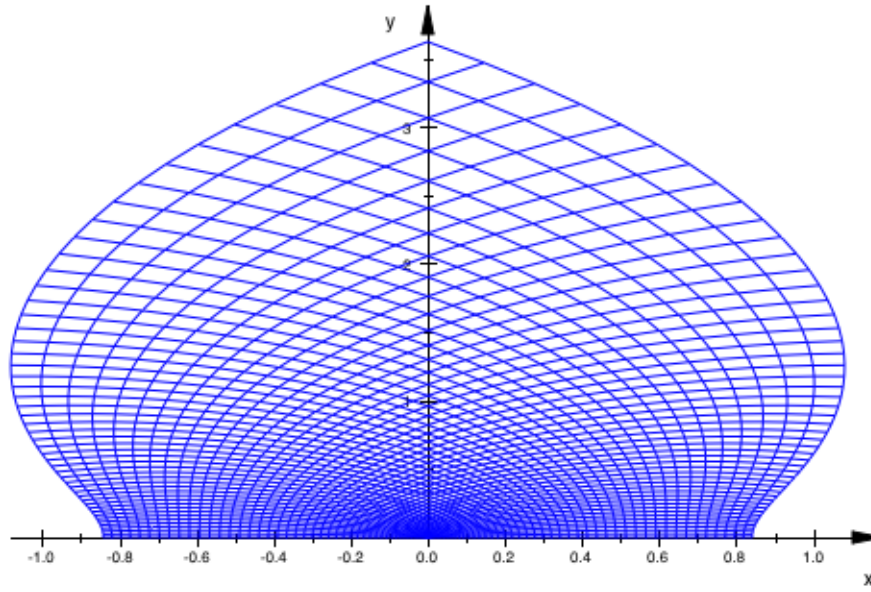


Fig. 15: Conformal plot

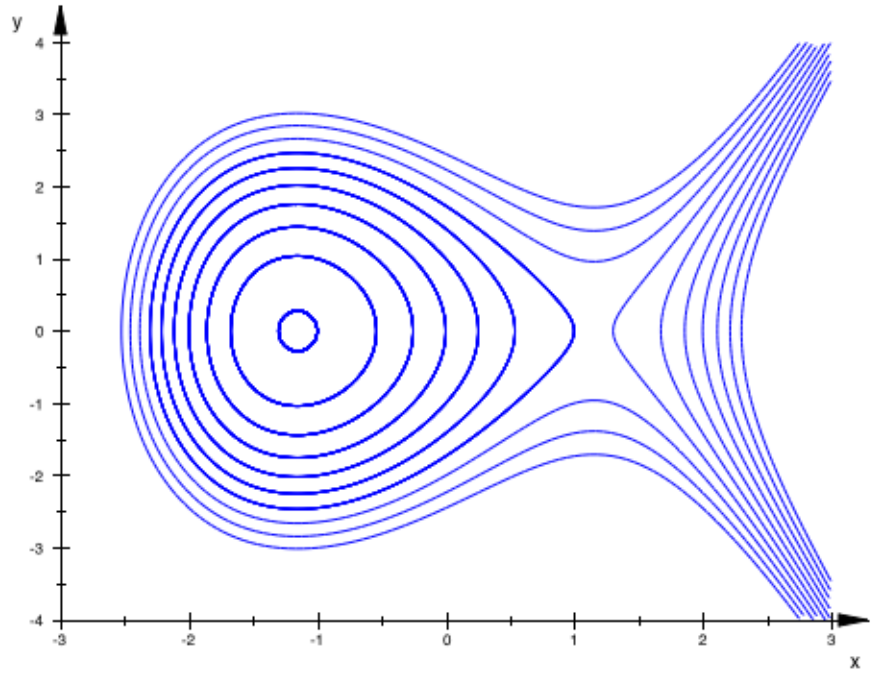


Fig. 16: Elliptic Curves

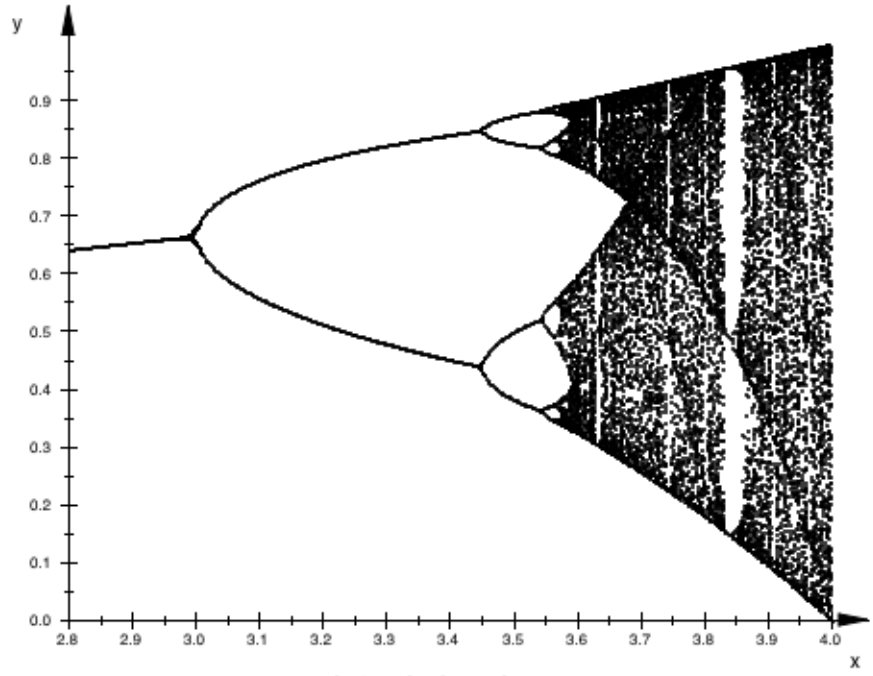


Fig. 17: Feigenbaum Diagram

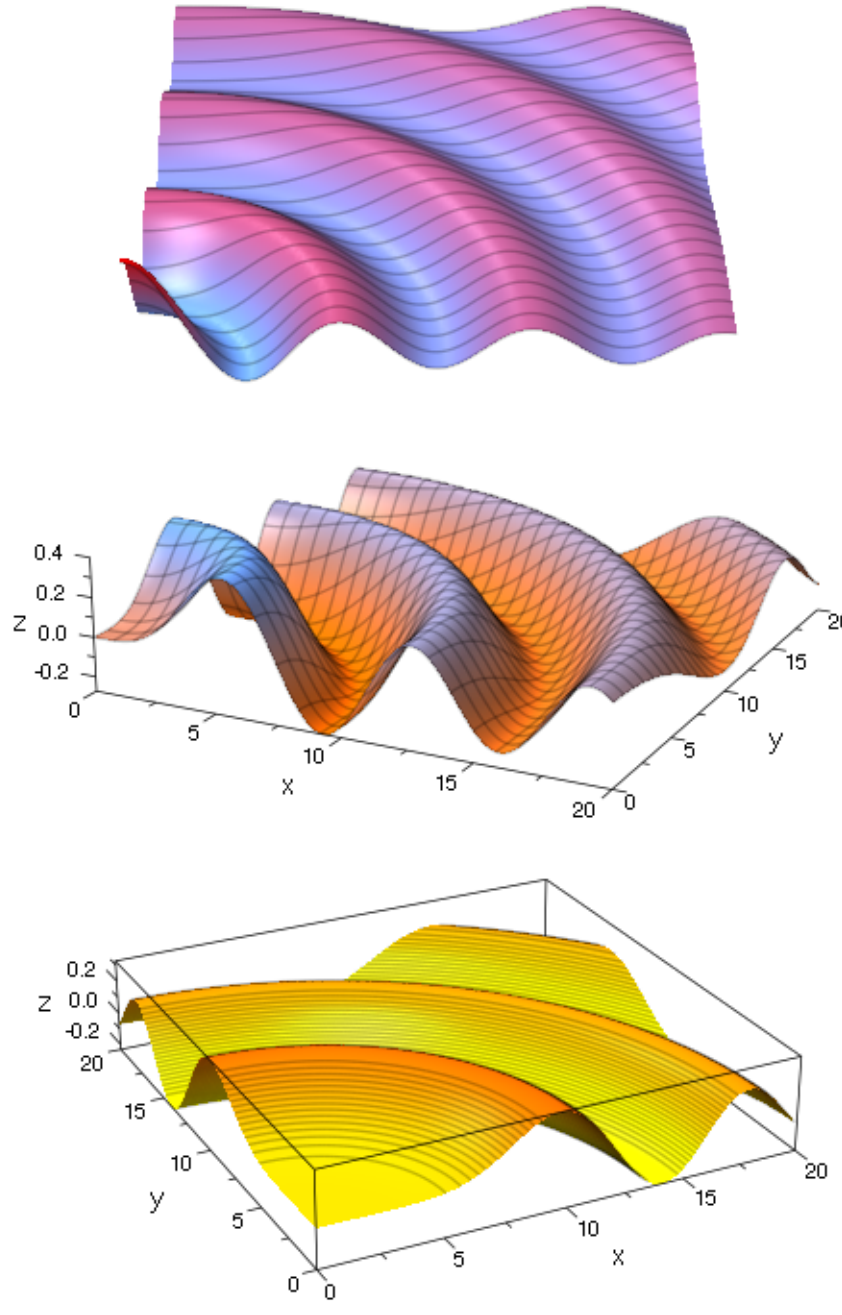


Fig. 18: Bessel Functions

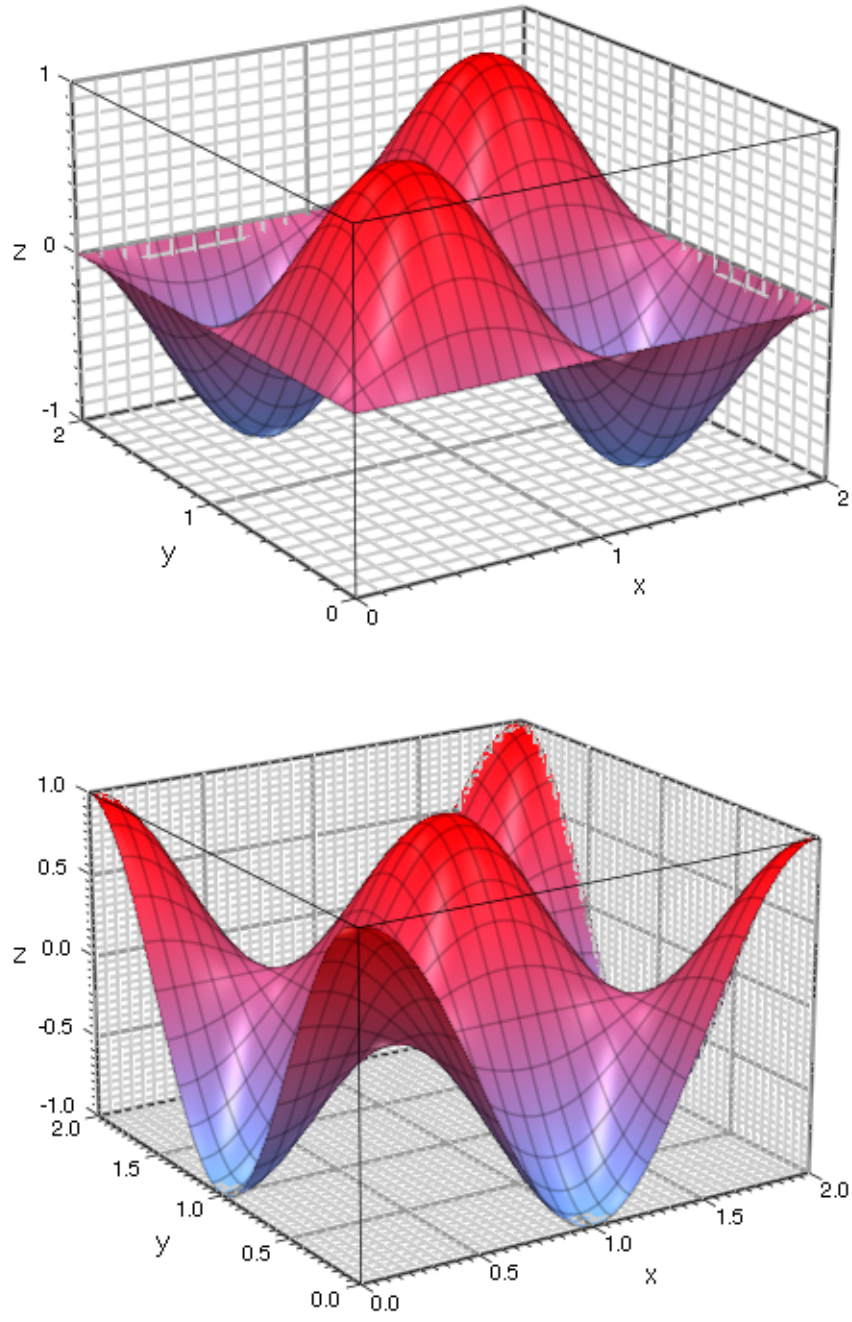


Fig. 19: Functions with Coordinate Grid

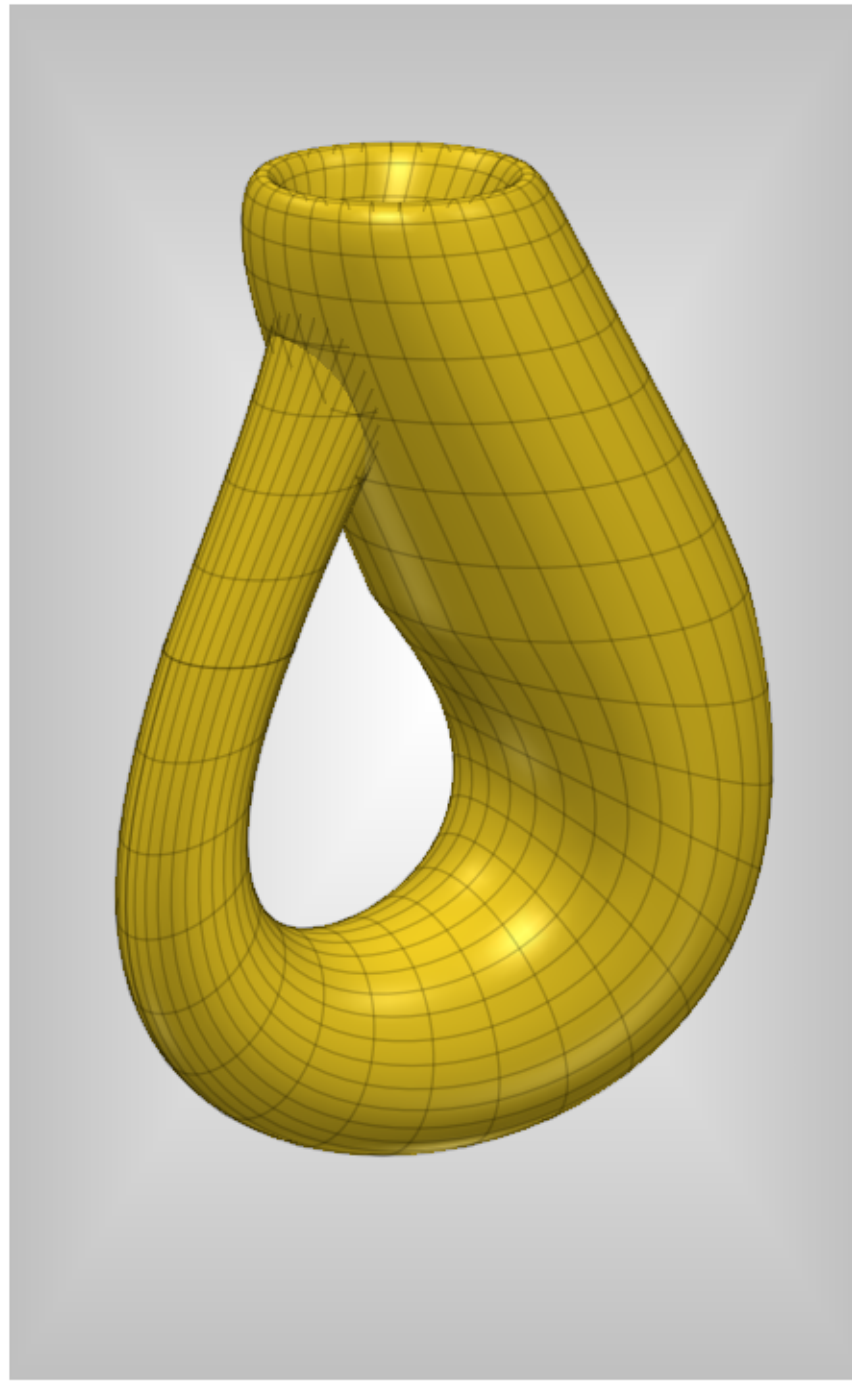


Fig. 20: Klein's Bottle

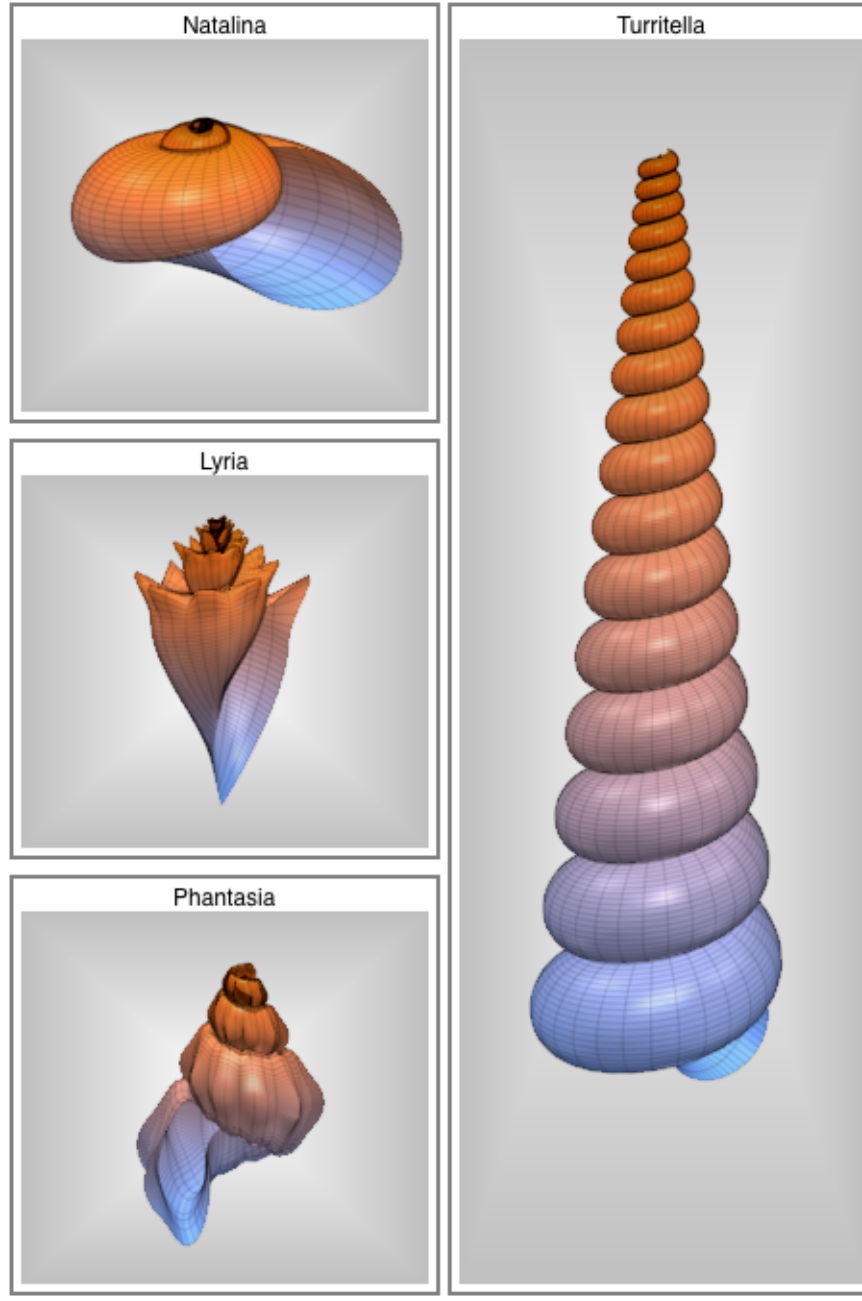


Fig. 21: Snails (Maike Kramer-Jka)

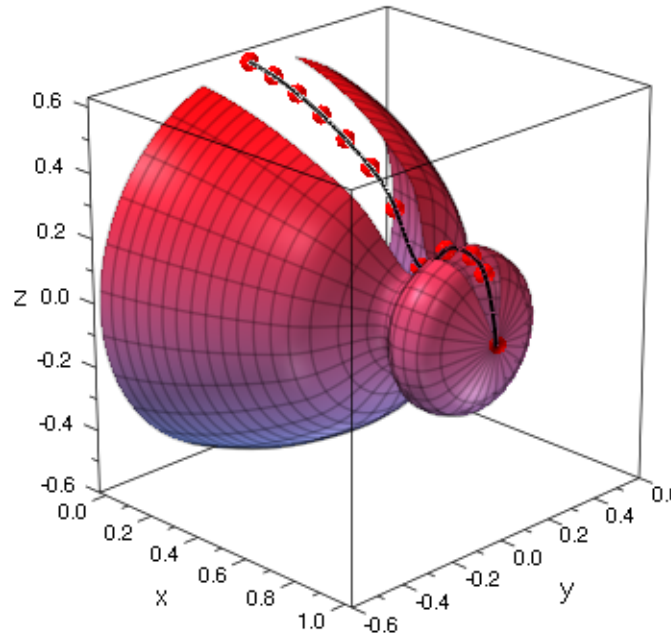


Fig. 22: Surface of Revolution

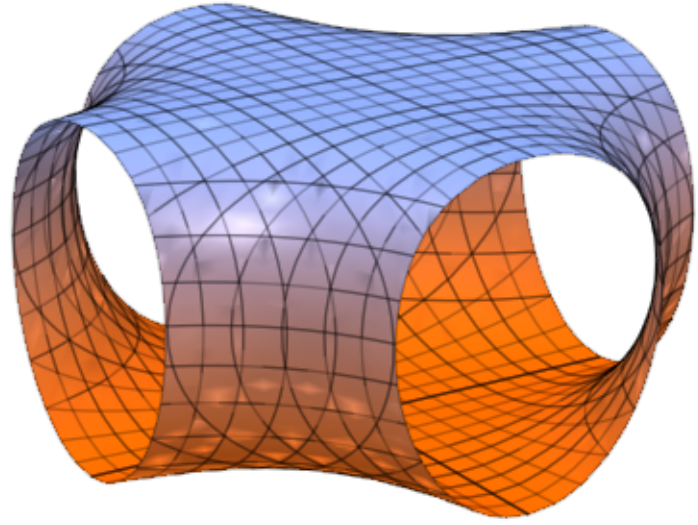


Fig. 23: Implicit Surface

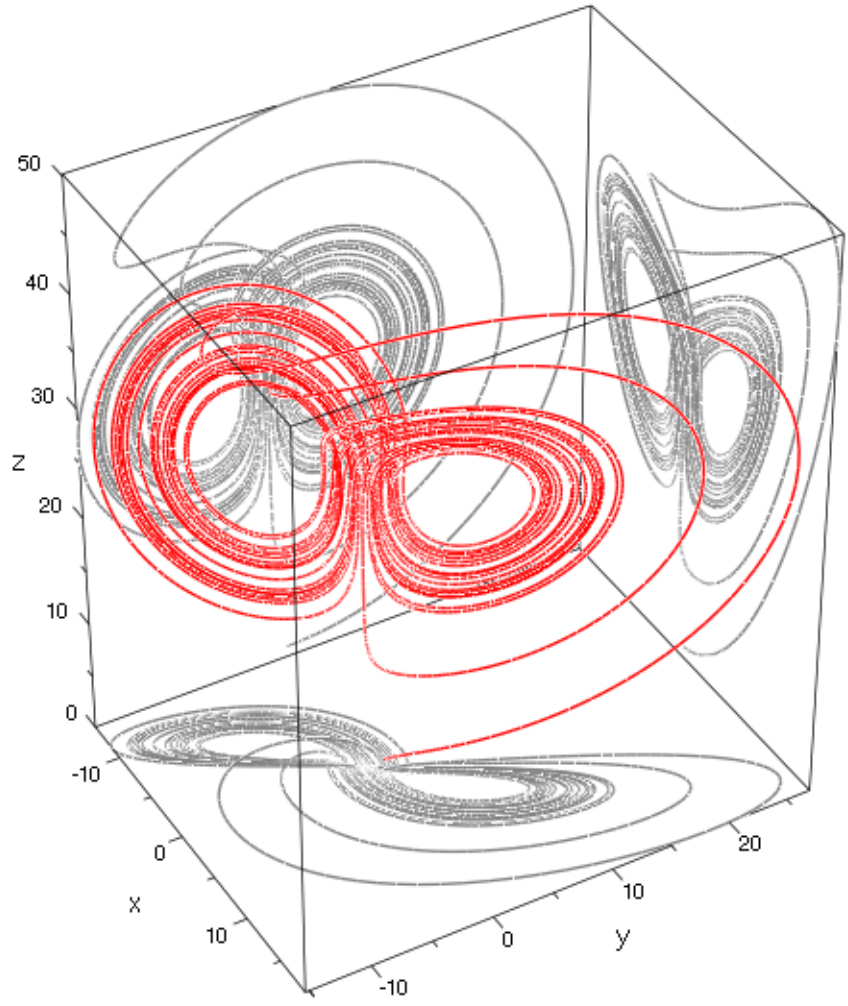


Fig. 24: Lorenz Attractor

Comments on the Graphics Gallery

On the color pages of this book you find a gallery of pictures demonstrating the power of the MuPAD® graphics. These pictures are discussed at various locations in this book and the online `plot` documentation, respectively. There, further details including the MuPAD commands for reproducing the pictures can be found.

Figure 1 shows a plot of several functions. Singularities are highlighted by "vertical asymptotes." See page 11-3. **Figure 2** shows a function plot together with a spline interpolation through a set of sample points. See page 11-39.

Figure 3 demonstrates some layout possibilities. See the examples on the help page of the graphical attribute `Layout` in the online `plot` documentation.

Figure 4 demonstrates the construction of cycloids via points fixed to a rolling wheel. See page 11-41.

Figure 5 and **Figure 6** demonstrate hatched areas between functions and within closed curves, respectively. See the examples on the help page of `plot::Hatch`.

Figure 7 shows various statistical distribution functions.

Figure 8 shows an imported bitmap inside function plots. See page 11-110.

Figure 9 shows some frames of an animation of the perturbed orbit of a small planet kicked out of a solar system by a giant planet after a near-collision. The animation is generated in Section „Animations"/„Examples" of the online `plot` documentation.

Figure 10 visualizes the vector field $\vec{v}(x, y) = (\sin(3\,\pi\,y), \sin(3\,\pi\,x))$.

Figure 11 shows three solution curves of an ODE inside the directional vector field associated with the ODE. See the examples on the help page of `plot::VectorField2d`.

Figure 12 shows several rotated copies of a function graph. See the examples on the help page of `plot::Rotate2d`.

Figure 13 shows the Mandelbrot set together with two blow ups of regions of special interest. See the examples on the help page of `plot::Density`.

Figure 14 shows a bar plot of statistical data. See the examples on the help page of `plot::Bars2d`.

Figure 15 shows the image of a rectangle in the complex plane under the map $z \to \sin(z^2)$. See the examples on the help page of `plot::Conformal`.

Figure 16 shows some elliptic curves generated as a contour plot. See the examples on the help page of `plot::Implicit2d`.

Figure 17 shows the Feigenbaum diagram of the logistic map. See the examples on the help page of `plot::PointList2d`.

Figure 18 shows Bessel functions $J_\nu(\sqrt{x^2+y^2})$ with $\nu = 0, 4, 8$. See the examples on the help page of `plot::Function3d`.

Figure 19 shows 3D function plots with coordinate grid lines. See the examples on the help page of the graphical attribute `GridVisible`.

Figure 20 shows "Klein's bottle" (a famous topological object). This surface does not have an orientation; there is no "inside" and no "outside" of this object. See the examples on the help page of `plot::Surface`.

Figure 21 models various snails using `plot::Surface` (by Maike Kramer-Jka).

Figure 22 demonstrates the re-construction of an object with rotational symmetry from measurements of its radius at various points. See page 11-44.

Figure 23 shows the solution set of the equation $z^2 = \sin(z - x^2 \cdot y^2)$ in 3D. See the examples on the help page of `plot::Implicit3d`.

Figure 24 shows the "Lorenz attractor." See page 11-116.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z