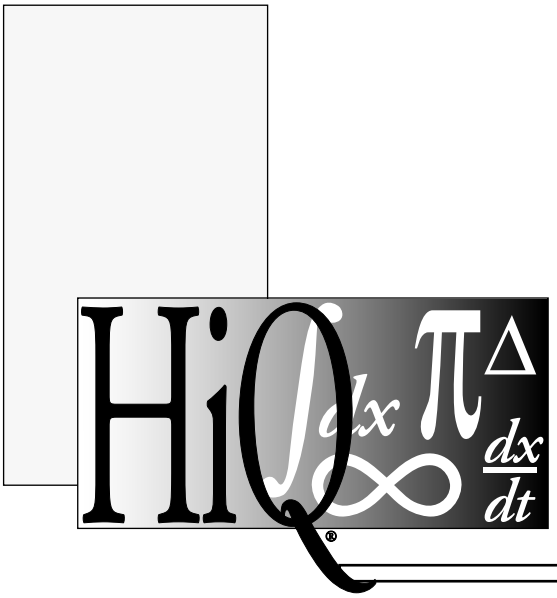


HiQ® Reference Manual

for Macintosh and Power Macintosh

Version 2.1

Part Number 320735B-01
August 1994

National Instruments Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
Technical support fax: (800) 328-2203
(512) 794-5678

Branch Offices:
Australia (03) 879 9422, Austria (0662) 435986, Belgium 02/757.00.20, Canada (Ontario) (519) 622-9310,
Canada (Québec) (514) 694-8521, Denmark 45 76 26 00, Finland (90) 527 2321, France (1) 48 14 24 24,
Germany 089/741 31 30, Italy 02/48301892, Japan (03) 3788-1921, Netherlands 03480-33466,
Norway 32-848400, Spain (91) 640 0085, Sweden 08-730 49 70, Switzerland 056/20 51 51, U.K. 0635 523545

Limited Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error-free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

Warning Regarding Medical and Clinical Use of National Instruments Products

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

CONTENTS

About The HiQ Reference Manual

Manual Organization

To properly address all of HiQ's capabilities, in a manageable book format, two volumes are used:

- **User Manual**: shows how to install HiQ on your computer and introduces HiQ concepts.
- **Reference Manual**: a full reference to all of HiQ's built-in functions including a discussion of their algorithms.

You are currently reading the HiQ Reference Manual which describes all functions available in HiQ.

Notation Conventions

Examples of HiQ-Script code look like the following excerpt.

```
function factorial(y)
   x = y;
   z = x;
   while(x > 1)
      x = x-1;
      z = z*x;
   end while;
   return z;
end function;
```

Keywords are in bold. When HiQ-Script keywords such as **while** are referred to within text, they will be set in the `Courier` typeface.

CHAPTER 1
HIQ-SCRIPT LANGUAGE CONSTANTS

HiQ provides a convenient shorthand method for supplying numerical constants in Scripts, both for numbers like π and for flag parameters required by HiQ's built-in functions.

Incorporating π into an equation, for example, is as easy as:

```
x = sin(2*a*<pi>/b);
```

Many built-in functions require as input a choice of method. For example, to export a matrix symbol to an ASCII file, you must specify the delimiter character which separates row elements. It's much easier to remember the language constant <tab_delimit> than to remember the integer 202.

The language constants reside in an ASCII text file named HiQ•Constants.PowerMac for Power Macintosh and HiQ•Constants.68kMac for 680x0 Macintosh in the same directory as the HiQ executable. You may modify this file to add your own. Any text editor that outputs straight ASCII (without word processing codes) will do, or you can import the file into a script symbol and export it to ASCII after making your changes.

Each language constant is defined by a name between angle brackets, a type (integer, real, or complex), and a value. Language constants are not case sensitive, so you do not need to use capitalization consistently.

HiQ reads the constants file once as it begins execution, so if you modify it while the HiQ process is running, you'll have to quit and restart HiQ.

Numerical Constants

Constant	Type	Value
<pi>	real	3.14159265358979323846
<e>	real	2.71828182845904523536
<i>	complex	$\sqrt{-1}$

Machine Constants

Constant	Type	Value for Power Macintosh	Value for 680X0 Macintosh
<minInt>	integer	-2147483648	-2147483648
<maxInt>	integer	2147483647	2147483647

Constant	Type	Value for Power Macintosh	Value for 680X0 Macintosh
<minNo>	real	2.2250738585072013e-308	1.6810515715560467530e-4932
<minDenormNo>	real	4.9406564584125654e-324	1.8225997659412373010e-4951
<maxNo>	real	1.7976931348623159e+308	1.1897314953572231765e+4932
<minLog>	real	-7.0839641853226410e+2	-1.1356523406294143950e+4
<maxLog>	real	7.0978271289338399e+2	1.1356523406294143950e+4
<epsilon>	real	2.2204460492503130808e-16	1.0842021724855044340e-19

Note: Use of these machine constants does not guarantee portability of compiled HiQ-Scripts between Power Macintosh and 680X0 Macintosh platforms.

numberToString Constants

Constant	Type	Value
<fixedPoint>	integer	0
<scientific>	integer	1
<complexFixedPoint>	integer	2
<complexScientific>	integer	3

Type Constants

Constant	Type	Value
<integer>	integer	4
<real>	integer	5
<complex>	integer	6

Logical Constants

Constant	Type	Value
<false>	integer	0
<true>	integer	1
<no>	integer	0
<yes>	integer	1
<cancel>	integer	0
<ok>	integer	1
<off>	integer	0
<on>	integer	1
<disable>	integer	0
<enable>	integer	1

Digital Signal Processing Window Types

Constant	Type	Value
<bartlett>	integer	1
<parzen>	integer	2
<hamming>	integer	3
<blackman>	integer	4
<rectangle>	integer	5
<chebyshev>	integer	6
<gauss>	integer	7
<kaiser>	integer	8

Digital Signal Processing constants which define redundancy indices used in power spectrum estimations

Constant	Type	Value
<nonQuat>	real	1.00
<oneQuat>	real	1.25
<twoQuat>	real	1.50
<triQuat>	real	1.75

Digital Signal Processing Filter Types

Constant	Type	Value
<lPass>	integer	1
<hPass>	integer	2
<bPass>	integer	3
<bStop>	integer	4

Initial and Boundary Value Problem Solver Flags

Constant	Type	Value
odeIvpSmooth Sequence Type		
<bulrsqn>	integer	1
<romberg>	integer	2
odeIvpStiff Nonlinear System Algorithm		
<newton>	integer	3
<brent>	integer	2
<quasiNewton>	integer	4
odeBvpGenLinear Shooting Algorithm		
<simple>	integer	0

Constant	Type	Value
<march>	integer	1
odeBvpGenLinear IVP Type		
<IV_rkf45>	integer	0
<IV_rkfs>	integer	1
<IV_bulr>	integer	2
<IV_stiff>	integer	3
odeBvpGenNonlinear Boundary Condition Type		
<nonlinear>	integer	2
<linear>	integer	3

Optimizer Flags

Constant	Type	Value
optNonLinCon		
<conjugate_grad>	integer	-1
<quasi_newton>	integer	-2
optBFGS		
<bfgs>	integer	-1
<broyden>	integer	-2

Infinite Integral Type Flags

Constant	Type	Value
<bnd2inf>	integer	1
<inf2bnd>	integer	2
<inf2inf>	integer	3

integFourier Weight Function Flags

Constant	Type	Value
<coswx>	integer	1
<sinwx>	integer	2

integOscillate Flags

Constant	Type	Value
<cosOsc>	integer	1
<sinOsc>	integer	2

CPUTimer Flags

Constant	Type	Value
<zero>	integer	0
<start>	integer	1
<stop>	integer	2
<get>	integer	3

Graph Flags and Parameters

Constant	Type	Value
Coordinate Systems		
<cartesian>	integer	0
<polar>	integer	1
<spherical>	integer	1
<cylindrical>	integer	2
Graph Flags		

Constant	Type	Value
<hidden_title>	integer	1
<hidden_axes>	integer	2
<hidden_grids>	integer	4
<hidden_annotation>	integer	8
<filled_frames>	integer	16
<shown_legend>	integer	32
Axis Flags		
<vertical_title>	integer	32
<vertical_labels>	integer	64
<reversed_placement>	integer	128
Shading		
<shade_wire>	integer	0
<shade_line>	integer	1
<shade_height>	integer	2
<shade_light>	integer	3
<wire>	integer	0
<line>	integer	1
<height>	integer	2
<light>	integer	3
Lighting		
<ambient>	integer	0
<directional>	integer	1
Projection Type		
<orthographic>	integer	0

Constant	Type	Value
<perspective>	integer	1
Axes		
<x_axis>	integer	0
<y_axis>	integer	1
<z_axis>	integer	2
Scaling		
<linear_scale>	integer	0
<log_scale>	integer	1
Plot Format		
<curve>	integer	0
<surface>	integer	0
<point>	integer	1
<connected>	integer	2
<contour>	integer	2
Colors		
<black>	integer	0
<white>	integer	1
<red>	integer	2
<green>	integer	3
<blue>	integer	4
<cyan>	integer	5
<magenta>	integer	6
<yellow>	integer	7
Marker Style		

Constant	Type	Value
<circular>	integer	0
<square>	integer	1
<diamond>	integer	2
<triangular>	integer	3
<cross>	integer	4
<x_shape>	integer	5
Contour Placement		
<overlaid>	integer	0
<projected>	integer	1
Planes		
<yz_plane>	integer	1
<xz_plane>	integer	2
<xy_plane>	integer	3
Movie Types		
<bitMap>	integer	1
<pixMap>	integer	2
<PICT>	integer	3

Symbol Locks

Constant	Type	Value
<nameLock>	integer	0
<dataLock>	integer	1
<typeLock>	integer	2
<dimLock>	integer	3

Constant	Type	Value
<deleteLock>	integer	4
<masterLock>	integer	5

Symbol Types

Constant	Type	Value	Meaning
<untyped>	integer	0	Untyped
<integerScalar>	integer	1	Integer scalar
<realScalar>	integer	2	Real scalar
<complexScalar>	integer	3	Complex scalar
<integerVector>	integer	4	Integer vector
<realVector>	integer	5	Real vector
<complexVector>	integer	6	Complex vector
<integerMatrix>	integer	7	Integer matrix
<realMatrix>	integer	8	Real matrix
<complexMatrix>	integer	9	Complex matrix
<numeric>	integer	10	Numeric symbol
<HiQFunction>	integer	11	Compiled HiQ-Script function
<HiQBIFunction>	integer	12	Built-in function
<string>	integer	13	String
<plot>	integer	14	Plot
<graph>	integer	15	Graph
<movie>	integer	16	Movie
<HiQScript>	integer	19	HiQ-Script
<script>	integer	19	HiQ-Script

Constant	Type	Value	Meaning
<bvpOdeSolver>	integer	20	Boundary Value ODE Problem Solver
<exprEvaluator>	integer	21	Expression Evaluation Problem Solver
<integrator>	integer	23	Numerical Integration Problem Solver
<ivpOdeSolver>	integer	24	Initial Value ODE Problem Solver
<nonLinSysSolver>	integer	26	Nonlinear Systems Problem Solver
<optimizer>	integer	27	Optimization Problem Solver
<polyRootSolver>	integer	28	Polynomial Roots Problem Solver
<integralEqnSolver>	integer	30	Integral Equations Problem Solver
<generalRootSolver>	integer	31	General Roots Problem Solver
<dataFitSolver>	integer	32	Data Fitting Problem Solver
<picture>	integer	33	Picture
<annotation>	integer	34	Annotation

Export Formatting Flags

Constant	Type	Value
<scientific>	integer	1
<decimal>	integer	2
<delimit_space>	integer	201
<delimit_tab>	integer	202
<delimit_comma>	integer	203
<delimit_newline>	integer	204

Import Formatting Flags

Constant	Type	Value
<importNumeric>	integer	1
<importText>	integer	2

Representing Nothing

Constant	Type	Value
<null>	integer	0
<nil>	integer	0

Input/Output Functions

Constant	Type	Value
<seekFromEnd>	integer	0
<seekFromCurrent>	integer	1
<seekFromStart>	integer	2
<flushAllFiles>	integer	-1

Specify All Symbols for symbolSave or updateDisplay

Constant	Type	Value
<allSymbols>	integer	-2147483648

Boundary Condition Types for finiteDiffMat2

Constant	Type	Value
<dirichlet_BC>	integer	1
<neumann>	integer	2

CHAPTER 2
TRIGONOMETRIC FUNCTIONS

■ arcCos

FUNCTION

y = arcCos(x)

PURPOSE

Compute the inverse cosine of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcCos(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse cosine of x. If x is a n-dimmensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse cosine of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Inverse Cosine Function arcCos(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
SymbolSetType(x,5);
SymbolSetType(y1,5);

SymbolSetVectorDim(x,m);
SymbolSetVectorDim(y1,m);
for i = 1 to  m do
x[i] = getNumber("Enter the value of x: ","1.0");
y1[i] = arcCos(x[i]);
end for;
//  Result for x = 0.5:
//      y1:    1.0471975511966

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
SymbolSetType(z,6);
```

```
SymbolSetType(y2,6);

SymbolSetVectorDim(z,n);
SymbolSetVectorDim(y2,n);
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcCos(z[j]);
end for;

//  Result for z = (1,-1):
//      y2:  -0.904556894302381 +   -1.061275061905O4i
```

ALGORITHM AND COMMENTS

Domain: $-1 \le x \le 1$
Range: $0 \le y \le \pi$
Use xArcCos(x) to improve speed, but not accuracy

■ arcCot

FUNCTION

y = arcCot(x)

PURPOSE

Compute the inverse cotangent of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcCot(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse cotangent of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse cotangent of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Inverse Cotangent Function arcCot(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
```

```
   y1[i] = arcCot(x[i]);
end for;

//  Result for x = 1:
//         y1:   0.785398163397448

n = getNumber("Enter the number of z values the function is
       to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcCot(z[j]);
end for;
//  Result for z = (1,1):
//       y2:  0.553574358897045 +  -0.402359478108525i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $0 < y < \pi$

■ arcCsc

FUNCTION

y = arcCsc(x)

PURPOSE

Compute the inverse cosecant of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcCsc(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse cosecant of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse cosecant of each of the elements of x, respectively

EXAMPLES

```
//Example for Inverse Cosecant Function arcCsc(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
       to be evaluated at:","1");
for i = 1 to  m do
```

```
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcCsc(x[i]);
end for;

//  Result for x = 2:
//         y1:    0.523598775598299

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcCsc(z[j]);
end for;

          //  Result for z = (1,1):
          //   y2:    0.452278447151191 + -0.530637530952518i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x \leq -1, 1 \leq x < \infty$
Range: $-\pi/2 \leq y < 0, 0 < y \leq \pi/2$

■ arcSec

FUNCTION

y = arcSec(x)

PURPOSE

Compute the inverse secant of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcSec(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse secant of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse secant of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Inverse Secant Function arcSec(x)

project  x, y1, y2, z;
local i, j,  m, n;
```

```
m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcSec(x[i]);
end for;

//  Result for x = -2:
//         y1:           -2.0943951023932

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcSec(z[j]);
end for;

//  Result for z = (-1,1):
//       y2:   -2.02307477394609 + -0.530637530952518i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x \leq -1$, $1 \leq x < \infty$
Range: $\pi/2 < y \leq \pi$, $0 \leq y < \pi/2$

■ arcSin

FUNCTION

y = arcSin(x)

PURPOSE

Compute the inverse sine of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcSin(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse sine of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse sine of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Inverse Sine Function arcSin(x)
project  x, y1, y2, z;
```

```
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcSin(x[i]);
end for;

//  Result for x = -0.5:
//         y1:    -0.523598775598299

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcSin(z[j]);
end for;

//  Result for z = (-1,1):
//       y2:    -0.666239432492515 +  1.06127506190504i
```

ALGORITHM AND COMMENTS

Domain: $-1 \le x \le 1$
Range: $-\pi/2 \le y \le \pi/2$
Use xArcSin(x) to improve speed, but not guarantee accuracy

■ arcTan

FUNCTION

y = arcTan(x)

PURPOSE

Compute the inverse tangent of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcTan(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse tangent of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse tangent of each of the elements of x, respectively

EXAMPLES

```
//Example for  Inverse Tangent Function arcTan(x)
project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcTan(x[i]);
end for;

//  Result for x = -1:
//         y1:   -0.785398163397448

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcTan(z[j]);
end for;

//  Result for z = (-1,1):
//       y2:    -1.01722196789785 + 0.402359478108525i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-\pi/2 < y < \pi/2$
Use xArcTan(x) to improve speed, but not guarantee accuracy

■ COS

FUNCTION

y = cos(x)

PURPOSE

Compute the cosine of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of cos(x) in radians

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the cosine of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the cosine of each of the elements of x, respectively

EXAMPLES

```
//Example for Cosine Function cos(x)
project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = cos(x[i]);
end for;

//  Result x = <pi>/3:
//          y1:  0.5

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = cos(z[j]);
end for;

//  Result for z = (1, 1):
//          y2:   0.833730025131149 +  -0.988897705762865i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-1 \le y \le 1$
Use xCos(x) to improve speed, but not accuracy

■ cot

FUNCTION

y = cot(x)

PURPOSE

Compute the cotangent of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of cot(x) in radians

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the cotangent of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the cotangent of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Cotangent Function cot(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = cot(x[i]);
end for;

//  Result x = <pi>/4:
//          y1:  1

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = cot(z[j]);
end for;

//  Result for z = (1, 1):
//          y2:  0.217621561854403 + -0.868014142895925i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$ and $x \neq \pm n\pi$ for all integer n
Range: $-\infty < y < \infty$
Singularity occurs if x = 0

CSC

FUNCTION

y = csc(x)

PURPOSE

Compute the cosecant of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of csc(x) in radians

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the cosecant of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the cosecant of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Cosecant Function csc(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = csc(x[i]);
end for;

//  Result for x = <pi>/6:
//          y1:  2

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = csc(z[j]);
end for;

//  Result for z = (1, 1):
//         y2:   0.621518017170428 + -0.303931001628426i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$ and $x \neq \pm n\pi$ for all integer n
Range: $y \leq -1, y \geq 1$
Singularity occurs if $x = n\pi$ for all integer n.

■ sec

FUNCTION

y = sec(x)

PURPOSE

Compute the secant of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of sec(x) in radians

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the secant of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the secant of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Secant Function sec(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = sec(x[i]);
end for;

//  Result for x = <pi>:
//          y1:   -1

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = sec(z[j]);
end for;

/  Result for z = (1, 1):
//         y2:     0.498337030555187 + 0.591083841721045i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$ and $x \neq \pm n\pi/2$ for n odd
Range: $y \leq -1, y \geq 1$
Singularity occurs if $x = n\pi/2$ for all odd integer n.

■ sin

FUNCTION

y = sin(x)

PURPOSE

Compute the sine of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of sin(x) in radians

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the sine of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the sine of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Sine Function sin(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = sin(x[i]);
end for;

//  Result for x = 3*<pi>/2:
//          y1:   -1

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = sin(z[j]);
end for;
/  Result for z = (1, 1):
```

```
//        y2:   1.29845758141598 + 0.6349639147847361
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-1 \le y \le 1$
Use xSin(x) to improve speed, but not accuracy

■ tan

FUNCTION

y = tan(x)

PURPOSE

Compute the tangent of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of tan(x) in radians

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the tangent of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the tangent of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Tangent Function tan(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = tan(x[i]);
end for;

//  Result for x = 7*<pi>/4:
//          y1:  -1

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = tan(z[j]);
```

```
end for;

//  Result for z = (1, 1):
//        y2:    0.271752585319512 + 1.083923327338691i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$ and $x \neq \pm n\pi/2$ for n odd
Range: $-\infty < y < \infty$
Use xTan(x) to improve speed, but not guarantee accuracy

■ xArcCos

FUNCTION

y = xArcCos(x)

PURPOSE

Compute the inverse cosine of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xArcCos(x)

OUTPUT

y (Real Scalar): the inverse cosine of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Inverse Cosine Function xArcCos(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
end for;

//  Result for x = 0.5:
//         y:           1.0471975511966
```

ALGORITHM AND COMMENTS

Domain: $-1 \le x \le 1$
Range: $0 \le y \le \pi$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xArcSin

FUNCTION

y = xArcSin(x)

PURPOSE

Compute the inverse sine of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xArcSin(x)

OUTPUT

y (Real Scalar): the inverse sine of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Inverse Sine Function xArcSin(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xArcSin(x[i]);
end for;

//  Result for x = -1:
//          y:     -1.5707963267949
```

ALGORITHM AND COMMENTS

Domain: $-1 \le x \le 1$
Range: $-\pi/2 \le y \le \pi/2$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xArcTan

FUNCTION

y = xArcTan(x)

PURPOSE

Compute the inverse tangent of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xArcTan(x)

OUTPUT

y (Real Scalar): the inverse tangent of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Inverse Tangent Function xArcTan(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xArcTan(x[i]);
end for;

//  Result for x = 1:
//          y:     0.785398163397448
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-\pi/2 < y < \pi/2$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xCos

FUNCTION

y = xCos(x)

PURPOSE

Compute the cosine of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xCos(x) in radians

OUTPUT

y (Real Scalar): the cosine of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Cosine Function xCos(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xCos(x[i]);
end for;

//  Result for x = <pi>/3:
//          y:   0.5
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-1 \leq y \leq 1$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xSin

FUNCTION

y = xSin(x)

PURPOSE

Compute the sine of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xSin(x) in radians

OUTPUT

y (Real Scalar): the sine of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Sine Function xSin(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xSin(x[i]);
end for;

//  Result for x = 3*<pi>/2:
//          y:   -1
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-1 \leq y \leq 1$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xTan

FUNCTION

y = xTan(x)

PURPOSE

Compute the tangent of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xTan(x) in radians

OUTPUT

y (Real Scalar): the tangent of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Inverse Cosine Function xTan(x)
project  x, y;
```

```
local i,  m;
m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xTan(x[i]);
end for;

//  Result for x = 7*<pi>/4:
//          y:   -1
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$ and $x \neq \pm n\pi/2$ for n odd

Range: $-\infty < y < \infty$

Algorithmic precision limited by the algorithm provided via the math coprocessor

CHAPTER 3
TRANSCENDENTAL FUNCTIONS

■ arcCosh

FUNCTION

y = arcCosh(x)

PURPOSE

Compute the inverse hyperbolic cosine of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcCosh(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse hyperbolic cosine of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse hyperbolic cosine of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the Inverse
// Hyperbolic Cosine Function arcCosh(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
       to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcCosh(x[i]);
end for;

//  Result for x = 1:
//          y1:  0.0

n = getNumber("Enter the number of z values the function is
       to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcCosh(z[j]);
end for;
```

```
//  Result for z = (1, -1):
//         y2:    1.06127506190504 + -0.9045568943023 81i
```

ALGORITHM AND COMMENTS

Domain: $1 \leq x < \infty$
Range: $0 < y < \infty$

■ arcCoth

FUNCTION

y = arcCoth(x)

PURPOSE

Compute the inverse hyperbolic cotangent of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcCoth(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse hyperbolic cotangent of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse hyperbolic cotangent of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the Inverse
// Hyperbolic Cotangent Function arcCoth(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcCoth(x[i]);
end for;

//  Result for x = -2:
//         y1:     -0.549306144334055

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
```

```
   y2[j] = arcCoth(z[j]);
end for;

//  Result for z = (-1, 1):
//         y2:    0.402359478108525 + 0.553574358897045i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < -1, 1 < x < \infty$
Range: $-\infty < y < \infty$
Singularity occurs if x =-1.0 or if x = 1.0

■ arcCsch

FUNCTION

y = arcCsch(x)

PURPOSE

Compute the inverse hyperbolic cosecant of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcCsch(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse hyperbolic cosecant of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse hyperbolic cosecant of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the Inverse
// Hyperbolic Cosecant Function arcCsch(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcCsch(x[i]);
end for;

//  Result for x = -1:
//         y1:    -0.881373587019543
```

```
n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcCsch(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:   0.530637530952518 + 0.452278447151191i
```

ALGORITHM AND COMMENTS

Domain: -1 < x < 1
Range: -∞ < y < ∞
Singularity occurs if x =0.0, arcCsch(x) set to 0.0

■ arcSech

FUNCTION

Y = arcSech(x)

PURPOSE

Compute the inverse hyperbolic secant of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcSech(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse hyperbolic secant of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse hyperbolic secant of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the Inverse
// Hyperbolic Secant Function arcSech(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcSech(x[i]);
end for;
```

```
//  Result for x = 1:
//          y1:   0

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcSech(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:    0.530637530952518 + 1.11851787964371i
```

ALGORITHM AND COMMENTS

Domain: $0 < x \leq 1$
Range: $0 < y < \infty$

■ arcSinh

FUNCTION

y = arcSinh(x)

PURPOSE

Compute the inverse hyperbolic sine of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcSinh(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse hyperbolic sine of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse hyperbolic sine of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the Inverse
// Hyperbolic Sine Function arcSinh(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
```

```
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcSinh(x[i]);
end for;

//  Result for x = -1:
//         y1:         -0.881373587019543

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcSinh(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:   1.06127506190504 + -0.666239432492515i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-\infty < y < \infty$

■ arcTanh

FUNCTION

y = arcTanh(x)

PURPOSE

Compute the inverse hyperbolic tangent of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of arcTanh(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the inverse hyperbolic tangent of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the inverse hyperbolic tangent of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the Inverse
// Hyperbolic Tangent Function arcTanh(x)

project  x, y1, y2, z;
local i, j,  m, n;
```

```
m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = arcTanh(x[i]);
end for;

//  Result for x = -0.5:
//         y1:    -0.549306144334055

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = arcTanh(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:    0.402359478108525 +  -1.01722196789785i
```

ALGORITHM AND COMMENTS

Domain: $-1 < x < 1$
Range: $-\infty < y < \infty$
Singularity occurs if x =1.0 or if x = -1.0

■ cosh

FUNCTION

y = cosh(x)

PURPOSE

Compute the hyperbolic cosine of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of cosh(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the hyperbolic cosine of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the hyperbolic cosine of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Hyperbolic Cosine Function cosh(x)
```

```
project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = cosh(x[i]);
end for;

//  Result for x = 0.0:
//         y1:  1

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = cosh(z[j]);
end for;

//  Result for z = (-1, 1):
//         y2:   0.833730025131149 + -0.988897705762865i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $1 \leq y < \infty$
If |x| > MAXLOG a warning message is returned.
Use xCosh(x) to improve speed, but not accuracy.

■ coth

FUNCTION

y = coth(x)

PURPOSE

Compute the hyperbolic cotangent of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of coth(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the hyperbolic cotangent of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the hyperbolic cotangent of each of

the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Hyperbolic Cotangent Function coth(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = coth(x[i]);
end for;

//  Result for x = -1:
//       y1:          -1.31303528549933

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = coth(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:      0.868014142895925 + 0.217621561854403i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$ and $x \neq 0$
Range: $-\infty < y \leq -1$, $1 \leq y < \infty$
Singularity occurs if x = 0.0

■ csch

FUNCTION

y = csch(x)

PURPOSE

Compute the hyperbolic cosecant of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of csch(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the hyperbolic cosecant of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the hyperbolic cosecant of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Hyperbolic Cosecant Function csch(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = csch(x[i]);
end for;

//  Result for x = -1:
//       y1:         -0.850918128239322

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = csch(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:   0.303931001628426 +  0.621518017170428i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$ and $x \neq 0$
Range: $-\infty < y < \infty$
If $|x| >$ MAXLOG a warning message is returned
Singularity occurs if $x = 0.0$

■ exp

FUNCTION

y = exp(x)

PURPOSE

Compute the exponential function of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of exp(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the exponential function of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the exponential function of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Exponential Function exp(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = exp(x[i]);
end for;

//  Result for x = 0.0:
//      y1:   1

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = exp(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:   1.46869393991589 + -2.28735528717884i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $0 < y < \infty$
If $|x| >$ MAXLOG, a warning message is returned
If $|x| <$ MINLOG, exp(x) = 1.0
Use xExp(x) to improve speed, but not accuracy

■ gd

FUNCTION

y = gd(x)

PURPOSE

Compute the gudermannian of x

INPUT

x (Real Scalar): the argument of gd(x)

OUTPUT

y (Real Scalar): the gudermannian of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Gudermannian Function gd(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = gd(x[i]);
end for;

//  Result for x = 3.14:
//          y:   1.48428472765572
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-\pi/2 < y < \pi/2$
If |x| > MAXLOG, an error message is returned

■ gdInv

FUNCTION

y = gdInv(x)

PURPOSE

Compute the inverse gudermannian of x

INPUT

x (Real Scalar): the argument of gdInv(x)

OUTPUT

y (Real Scalar) : the inverse gudermannian of x

EXAMPLES

```
// Interactive HiQ-Script Example for the Inverse
// Gudermannian Function gdInv(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = gdInv(x[i]);
end for;

//  Result for x = 1.48428472765572:
//         y:    3.13999999999996
```

ALGORITHM AND COMMENTS

Domain: $-\pi/2 < x < \pi/2$
Range: $-\infty < y < \infty$
Domain error occurs if $\tan(z) \leq 0.0$,
where $z = x/2 + \pi/4$

■ ln

FUNCTION

$y = \ln(x)$

PURPOSE

Compute the natural logarithm of x, the logarithm to the Naperian base e

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of ln(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the natural logarithm of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the natural logarithm of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Natural Logarithm Function ln(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = ln(x[i]);
end for;

//  Result for x = 1.0:
//       y1:   0

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = ln(z[j]);
end for;

//  Result for z = (1, -1):
//        y2:   0.346573590279973 + -0.785398163397448i
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$
Range: $-\infty < y < \infty$
If x = 0 a warning message is returned.
Use xLn(x) to improve speed, but not accuracy

■ log

FUNCTION

y = log(x)

PURPOSE

Compute the logarithm to the base 10 of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of log(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the base ten logarithm of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the base ten logarithm of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Base 10 Logarithm Function log(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = log(x[i]);
end for;

//  Result for x = 10:
//       y1:  1

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = log(z[j]);
end for;

//  Result for z = (1, -1):
//        y2:   0.150514997831991 +  -0.341094088846046i
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$
Range: $-\infty < y < \infty$
If x = 0 a warning message is returned.
Use xLog(x) to improve speed, but not accuracy

■ logb

FUNCTION

y = logb(b,x)

PURPOSE

Compute the base b logarithm of x.

INPUT

b (Real Scalar): the non-unity (i.e., ≠ 1) positive base for the logarithm

x (Real or Complex Scalar): the real or complex scalar argument of the logarithm

OUTPUT

y (Real or Complex Scalar): the real or complex base b logarithm of x

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Base b Logarithm Function logb(b, x)

project  x, y1, y2, z;
local i, j,  m, n, b1, b2;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   b1 = getNumber("Enter the value of b: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = logb(b1, x[i]);
end for;

//  Result for b = 5 and x = 0.04:
//       y1:   -2
//
n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   b2 = getNumber("Enter the value of b: ","1.0");
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = logb(b2, z[j]);
end for;

//  Result for b = 2 and z = (1, -1):
//         y2:   0.5 +   -1.1330900354568i
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; $b > 0$ and $b \neq 1$.
Range: $-\infty < y < \infty$
If x = 0, logb(b, x) returns an error message

Because the base b logarithm of a complex number is in general not single-valued, the function logb computes the result for the input parameter x in the principal branch of the complex plane. That is, the polar representation for any $x \neq 0$ is taken as:

$$x = |x| e^{i\theta}$$

with $-\pi < \theta \leq \pi$

■ sech

FUNCTION

y = sech(x)

PURPOSE

Compute the hyperbolic secant of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of sech(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the hyperbolic secant of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the hyperbolic secant of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Hyperbolic Secant Function sech(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = sech(x[i]);
end for;

//  Result for x = 0.0:
```

```
//        y1:  1

n = getNumber("Enter the number of z values the function is
       to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = sech(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:     0.498337030555187 + 0.591083841721045i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $0 \le y \le 1$
If |x| > MAXLOG, a warning message is returned

■ sinh

FUNCTION

y = sinh(x)

PURPOSE

Compute the hyperbolic sine of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of sinh(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the hyperbolic sine of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the hyperbolic sine of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Hyperbolic Sine Function sinh(x)

project  x, y1, y2, z;
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
       to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
```

```
   y1[i] = sinh(x[i]);
end for;

//  Result for x = 0.0:
//        y1:  0

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = sinh(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:  0.634963914784736 + -1.29845758141598i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-\infty < y < \infty$
If |x| > MAXLOG, a warning message is returned
Use xSinh(x) to improve speed, but not accuracy

■ tanh

FUNCTION

y = tanh(x)

PURPOSE

Compute the hyperbolic tangent of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the argument of tanh(x)

OUTPUT

y (Real or Complex Scalar, Vector, Matrix): the hyperbolic tangent of x. If x is a n-dimensional vector or a m by n matrix, y is the n-dimensional vector or m by n matrix containing the hyperbolic tangent of each of the elements of x, respectively

EXAMPLES

```
// Interactive HiQ-Script Examples for the
// Hyperbolic Tangent Function tanh(x)

project  x, y1, y2, z;
```

```
local i, j,  m, n;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to  m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y1[i] = tanh(x[i]);
end for;

//  Result for x = 0.0:
//       y1:  0

n = getNumber("Enter the number of z values the function is
      to be evaluated at:","1");
for  j = 1 to  n do
   z[j] = getNumber("Enter the value of z: ","1.0,1.0");
   y2[j] = tanh(z[j]);
end for;

//  Result for z = (1, -1):
//         y2:     1.08392332733869 + -0.271752585319512i
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-1 \le y < 1$
Use xTanh(x) to improve speed, but not guarantee accuracy

■ xCosh

FUNCTION

y = xCosh(x)

PURPOSE

Compute the hyperbolic cosine of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xCosh(x)

OUTPUT

y (Real Scalar): the hyperbolic cosine of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Hyperbolic Cosine Function xCosh(x)
```

```
project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xCosh(x[i]);
end for;

//  Result for x = 0.0:
//         y:  1
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $1 \leq y < \infty$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xExp

FUNCTION

y = xExp(x)

PURPOSE

Compute the exponential function of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xExp(x)

OUTPUT

y (Real Scalar): the exponential function of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Exponential Function xExp(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xExp(x[i]);
```

```
end for;

/  Result for x = 1:
//         y:  2.71828182845905
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $0 < y < \infty$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xLn

FUNCTION

y = xLn(x)

PURPOSE

Compute the natural logarithm of x, the logarithm to the Naperian base e, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xLn(x)

OUTPUT

y (Real Scalar): the natural logarithm of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Natural Logarithm Function xLn(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xLn(x[i]);
end for;

//  Result for x = 1:
//         y:  0
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$

Range: $-\infty < y < \infty$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xLog

FUNCTION

y = xLog(x)

PURPOSE

Compute the logarithm to the base 10 of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xLog(x)

OUTPUT

y (Real Scalar): the base 10 logarithm of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Base 10 Logarithm Function xLog(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xLog(x[i]);
end for;

//  Result for x = 10:
//         y:  1
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$, $b > 0$ and $b \neq 1$; Range: $-\infty < y < \infty$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xLogb

FUNCTION

y = xLogb(b,x)

PURPOSE

Compute the base b logarithm of x, calling the coprocessor directly for improved speed of evaluation

INPUT

b (Real Scalar): the non-unity (i.e., ≠ 1) positive base for the logarithm

x (Real Scalar): the argument of xLogb(b, x)

OUTPUT

y (Real Scalar): the base b logarithm of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Base b Logarithm Function xLogb(b, x)

project  b, x, y;
local i, j, m, k;
m = getNumber("Enter the number of b values the function is to
   be evaluated at:","1");
k = getNumber("Enter the number of x values the function is to
   be evaluated at:","1");
for i = 1 to  m do
   for  j = 1 to k do
      b = getNumber("Enter the value of b: ","1.0");
      x[j] = getNumber("Enter the value of x: ","1.0");
      y[j] = xLogb(b, x[j]);
   end for;
end for;

//  Result for b = 5 at x = 125:
//         y:  3
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; $b > 0$ and $b \neq 1$. Range: $-\infty < y < \infty$

Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xSinh

FUNCTION

y = xSinh(x)

PURPOSE

Compute the hyperbolic sine of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xSinh(x)

OUTPUT

y (Real Scalar): the hyperbolic sine of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Hyperbolic Sine Function xSinh(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function
   is to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xSinh(x[i]);
end for;

//  Result for x = 0.0:
//         y:  0
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$
Range: $-\infty < y < \infty$
Algorithmic precision limited by the algorithm provided via the math coprocessor

■ xTanh

FUNCTION

y = xTanh(x)

PURPOSE

Compute the hyperbolic tangent of x, calling the coprocessor directly for improved speed of evaluation

INPUT

x (Real Scalar): the argument of xTanh(x)

OUTPUT

y (Real Scalar): the hyperbolic tangent of x

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Hyperbolic Tangent Function xTanh(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function
   is to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = xTanh(x[i]);
end for;

//  Result for x = 1:
//         y: 0.761594155955765
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-1 \leq y < 1$
Algorithmic precision limited by the algorithm provided via the math coprocessor

CHAPTER 4
ORTHOGONAL POLYNOMIALS

■ aLag

FUNCTION

y = aLag(n, m, x)

PURPOSE

Compute the associated Laguerre polynomial of degree n and order m, $L_n^m(x)$, at the point x

INPUT

n (Integer Scalar): the degree of the polynomial $L_n^m(x)$

m (Integer Scalar): the order of the polynomial $L_n^m(x)$

x (Real Scalar): the argument value of the polynomial $L_n^m(x)$

OUTPUT

y (Real Scalar): the computed value of $L_n^m(x)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the Associated
//   Laguerre Polynomial of the First Kind of degree m and
//   order n at x, aLag(m, n, x)

project  m, n, x, y;
local i, p;

p = getNumber("Enter the number of values the function is to
   be evaluated at:","1");

for i = 1 to  p do
    m[i] = getNumber("Enter the value of m: ","1.0");
   n[i] = getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = aLag(m[i], n[i], x[i]);
end for;

//  Result for m = 3, n = 2, and x = 2.5:
//   y:  -1.97916666666667
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-\infty < y < \infty$
The degree, n, and the order, m, have to be non-negative integers
Results accurate to 9 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, pp. 214 - 215

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 22

■ aLeg

FUNCTION

y = aLeg(n, m, x)

PURPOSE

Compute the associated Legendre polynomial of degree n and order m, $P_n^m(x)$, of the first kind at the point x

INPUT

n (Integer Scalar): the degree of the polynomial $P_n^m(x)$

m (Integer Scalar): the order of the polynomial $P_n^m(x)$

x (Real Scalar): the argument value of the polynomial $P_n^m(x)$

OUTPUT

y (Real Scalar): the computed value of $P_n^m(x)$

EXAMPLES

```
//  Interactive HiQ-Script Example for the Associated
//  Legendre Polynomial of the First Kind of degree m and
//  order n at x, aLeg(m, n, x)

project  m, n, x, y;
local i, p;

p = getNumber("Enter the number of values the function is to be
      evaluated at:","1");

for i = 1 to  p do
   m[i] = getNumber("Enter the value of m: ","1.0");
   n[i] = getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = aLeg(m[i], n[i], x[i]);
end for;
```

```
//  Result for n = 2, m = 1, and x = 0.5:
//   y:   -1.29903810567666
```

ALGORITHM AND COMMENTS

Domain: $-1 \le x \le 1$; Range: $-\infty < y < \infty$

The degree n and the order m have to be non-negative integers with $m \le n$

Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, p. 586

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 22

■ cheb1

FUNCTION

y = cheb1(n, x)

PURPOSE

Compute the Chebyshev polynomial of degree n, $T_n(x)$, of the first kind at the point x

INPUT

n (Integer Scalar) : the degree of the polynomial $T_n(x)$

x (Real Scalar): the value at which the polynomial $T_n(x)$ is computed.

OUTPUT

y (Real Scalar): the computed value of $T_n(x)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the Chebyshev
//   Polynomial of the First Kind of degree n at x, cheb1(n, x)
project  n, x, y;
local i, p;
p = getNumber("Enter the number of values the function is to be
   evaluated at:","1");
for i = 1 to  p do
   n[i] = getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = cheb1(n[i], x[i]);
end for;
//  Result for n = 3 and x =-0.8:
//   y:  0.352
```

ALGORITHM AND COMMENTS

Domain: $-1 \le x \le 1$; Range: $-1 \le y \le 1$

The degree, n, has to be a non-negative integer

Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, pp. 194 - 199

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 22

■ cheb2

FUNCTION

y = cheb2(n, x)

PURPOSE

Compute the Chebyshev polynomial of degree n, $U_n(x)$, of the second kind at the point x

INPUT

n (Integer Scalar): the degree of the polynomial $U_n(x)$

x (Real Scalar): the value at which the polynomial $U_n(x)$ is computed

OUTPUT

y (Real Scalar): the computed value of $U_n(x)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the Chebyshev
//   Polynomial of the Second Kind of degree n at x, cheb2(n, x)
project  n, x, y;
local i, p;
p = getNumber("Enter the number of values the function is to be
   evaluated at:","1");
for i = 1 to  p do
   n[i] = getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = cheb2(n[i], x[i]);
end for;

//  Result for n = 3 and x = 0.8:
//   y:  0.896
```

ALGORITHM AND COMMENTS

Domain: $-1 \le x \le 1$; Range: $-n-1 \le y \le n+1$

The degree, n, has to be a non-negative integer

Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, pp. 194 - 199

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 22

■ harm

FUNCTION

y = harm(m, n, theta, phi)

PURPOSE

Compute the Spherical Harmonic function, $Y_{m,n}(\theta, \phi)$, at the spherical surface coordinate point (theta, phi)

INPUT

m (Integer Scalar): the first parameter of the function

n (Integer Scalar): the second parameter of the function

theta (Real Scalar): the polar angle, theta = θ, describing the (spherical surface) domain

phi (Real Scalar): the azimuthal angle, phi = ϕ, describing the (spherical surface) domain

OUTPUT

y (Complex Scalar): the computed value of $Y_{m,n}(\theta, \phi)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the Spherical Harmonic
//   Function with parameters m, n, t, and p, harm(m, n, t, p)

project  m, n, t, p, y;
local i, k;

k = getNumber("Enter the number of values the function is to be
   evaluated at:","1");

SymbolSetType(m,4);
SymbolSetType(n,4);
SymbolSetType(t,5);
SymbolSetType(x,5);
SymbolSetType(y,5);
```

```
SymbolSetVectorDim(m,k);
SymbolSetVectorDim(n,k);
SymbolSetVectorDim(t,k);
SymbolSetVectorDim(x,k);
SymbolSetVectorDim(y,k);
for i = 1 to  k do
   m[i] = getNumber("Enter the value of m: ","1");
   n[i] = getNumber("Enter the value of n: ","1");
   t[i] = getNumber("Enter the value of x: ","1.0");
   p[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = harm(m[i], n[i], t[i], p[i]);
end for;

//  Result for m = 4, n = 3, t = 0.2, and p = 0.8:
//   y:    -0.00649740741962272 +  0.00709312888769373i
```

ALGORITHM AND COMMENTS

Function is periodic in theta and phi with period 2π; Range: $\infty < y < \infty$
The first parameter, m, has to be a non-negative integer
The second parameter n assumes the values 0, ±1, ±2, ... , ± m
Results accurate to 8 decimal places

REFERENCE

Press, W.H., et. al., *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge Univ. Press, 1988, pp. 194-195

Arfken, G., *Mathematical Methods for Physicists*, 3rd. Ed., Academic Press, 1985, pp. 680 - 683

■ her

FUNCTION

y = her(n, x)

PURPOSE

Compute the Hermite polynomial of degree n, $H_n(x)$, at the point x

INPUT

n (Integer Scalar): the degree of the polynomial $H_n(x)$

x (Real Scalar): the value at which the polynomial $H_n(x)$ is computed

OUTPUT

y (Real Scalar): the computed value of $H_n(x)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the Hermite
//   Polynomial of degree n at x, her(n, x)

project  n, x, y;
local i, k;

k = getNumber("Enter the number of values the function is to be
   evaluated at:","1");

for i = 1 to  k do
   n[i] = getNumber("Enter the value of n: ","1");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = her(n[i], x[i]);
end for;

//  Result for n = 3 at x= 0.5:
//         y:  -5
```

ALGORITHM AND COMMENTS

Domain:$-\infty < x < \infty$; Range: $-\infty < y < \infty$
$H_n(x)$ is bounded globally by $1.09\sqrt{2^n\, n!}\, e^{x^2}$
The degree, n, has to be a non-negative integer
Results accurate to 11 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, p. 219 - 221

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 22

■ jac

FUNCTION

y = jac(n, a, b, x)

PURPOSE

Compute the Jacobi polynomial of degree n, $P_n^{a,b}(x)$, of the first kind at the point x

INPUT

n (Integer Scalar): the degree of the function $P_n^{a,b}(x)$

a (Real Scalar): the first parameter of the function $P_n^{a,b}(x)$

b (Real Scalar): the second parameter of the function $P_n^{a,b}(x)$

x (Real Scalar): the value at which the function $P_n^{a,b}(x)$ is computed

OUTPUT

y (Real Scalar): the computed value of $P_n^{a,b}(x)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the Jacobi Polynomial
/   of degree n and parameters a and b at x,  jac(n, a, b, x)

project  n, a, b, x, y;
local i, p;

p = getNumber("Enter the number of values the function is to be
   evaluated at:","1");
for  i = 1 to p do
   n[i] = getNumber("Enter the value of n: ","1");
   a[i] = getNumber("Enter the value of a: ","1.0");
   b[i] = getNumber("Enter the value of b: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = jac(n[i], a[i], b[i], x[i]);
end for;

//  Result for n =3, a = 0.2, b = -0.25, at x = 0.5:
//   y:           -0.4157119140625
```

ALGORITHM AND COMMENTS

Domain:$-\infty < x < \infty$; Range: $-\infty < y < \infty$
The degree, n, has to be a non-negative integer
The real parameters a, b must exceed -1: a, b > -1
Results accurate to 11 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, p. 201

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 22

■ lag

FUNCTION

y = lag(n, x)

PURPOSE

Compute the Laguerre polynomial of degree n, $L_n(x)$, at the point x

INPUT

n (Integer Scalar): the degree of the polynomial $L_n(x)$

x (Real Scalar): the value at which the polynomial $L_n(x)$ is computed

OUTPUT

y (Real Scalar): the computed value of $L_n(x)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the Laguerre
//   Polynomial of degree n at x, lag(n, x)

project n, x, y;
local i, p;

p = getNumber("Enter the number ofvalues the function is to be
   evaluated at:","1");

for i = 1 to  p do
   n[i] = getNumber("Enter the value of n: ","1");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = lag(n[i], x[i]);
end for;

   //  Result for n = 3 at x = 0.4:
//   y:         0.0293333333333333
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; $-\infty < y < \infty$
The degree, n, has to be a non-negative integer
Results accurate to 11 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, pp. 211 - 213

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 22

■ leg

FUNCTION

y = leg(n, x)

PURPOSE

Compute the Legendre polynomial of degree n, $P_n(x)$, of the first kind at x

INPUT

n (Integer Scalar): the degree of the polynomial $P_n(x)$

x (Real Scalar): the value at which the polynomial $P_n(x)$ is computed.

OUTPUT

y (Real Scalar): the computed value of $P_n(x)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the Legendre
//   Polynomial of the First Kind of degree n at x, leg(n, x)

project n, x, y;
local i, p;

p = getNumber("Enter the number ofvalues the function is to be
   evaluated at:","1");
for i = 1 to  p do
   n[i] = getNumber("Enter the value of n: ","1");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = leg(n[i], x[i]);
end for;

//  Result for n = 3 at x = 0.2:
//   y:         -0.28
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-\infty < y < \infty$
The degree, n, has to be a non-negative integer
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, pp. 187 - 188

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 22

■ qLeg

FUNCTION

y = qLeg(n, x)

PURPOSE

Compute the Legendre polynomial of degree n, $Q_n(x)$, of the second kind at x, which is the second independent solution of the differential equation:

$$(1 - x^2)\, y''(x) - 2xy'(x) + n(n+1)\, y(x) = 0$$

INPUT

n (Integer Scalar): the degree of the polynomial $Q_n(x)$

x (Real Scalar): the value at which the polynomial $Q_n(x)$ is computed

OUTPUT

y (Real Scalar): the computed value of $Q_n(x)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the Legendre
//   Polynomial of the Second Kind of degree n at x, qLeg(n, x)

project ni, x, y;
local i, p;

p = getNumber("Enter the number ofvalues the function is to be
   evaluated at:","1");
for i = 1 to  p do
   n[i] = getNumber("Enter the value of n: ","1");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = qLeg(ni[i], x[i]);
end for;

//  Result for n = 3 and x = -0.1:
//   y:          0.626867202876333
```

ALGORITHM AND COMMENTS

Domain: $-1 \le x \le 1$; Range: $-\infty < y < \infty$

The degree, n, has to be a non-negative integer

Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, pp. 191 - 192

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 8

CHAPTER 5
SPECIAL FUNCTIONS

■ ai

FUNCTION

y = ai(x)

PURPOSE

Compute the Airy function ai(x) defined by:

$$\mathrm{ai}(x) = \frac{1}{\pi}\int_0^{\infty}\cos\left(xt+\frac{t^3}{3}\right)dt = \begin{cases} \frac{\sqrt{x}}{3}\left[I_{-\frac{1}{3}}(\zeta) - I_{\frac{1}{3}}(\zeta)\right], & x>0 \\ \frac{\sqrt{-x}}{3}\left[J_{-\frac{1}{3}}(\zeta) - J_{\frac{1}{3}}(\zeta)\right], & x<0 \end{cases}$$

where $\zeta = 2|x|^{3/2}/3$ is the argument of the Bessel functions I and J

INPUT

x (Real Scalar): the value of the argument of ai(x)

OUTPUT

y (Real Scalar): the computed value of ai(x)

EXAMPLES

```
/  Interactive HiQ-Script Example for the Airy Function ai(x)
// Results given for x ranging from 1.0 to 1.9 in steps of 0.1

project  x, y;
aiPlott(x, y);

function aiPlott(x, y)

project low,  up, aigraph;
local  sttep;

low = getNumber("Enter the lower value of x in ai(x):","1.0");
up = getNumber("Enter the upper value of x in ai(x): ","2.0");
interval = getNumber("Enter the step size:  ","0.1");
```

```
x[1] = low;
y[1] = ai(low);
sttep = ( up - low)/interval;

for i = 2 to  sttep step 1 do
   x[i] = x[i-1] + interval;
   y[i] = ai(x[i]);
end for;

// Plot the results:

aigraph = new2DGraph( "Airy Function ai(x)");
aiplot = new2DDataplot("ai(x)",x,y);
addPlot(aigraph, aiplot);
end function;

// Results:
// x: 1      y:  0.135292416312881
//    1.1       0.120049427355398
//    1.2       0.106125762263313
//    1.3       0.0934746657715027
//    1.4       0.0820380498076107
//    1.5       0.0717494970081054
//    1.6       0.062536907968932
//    1.7       0.0543247927329195
//    1.8       0.0470362168668458
//    1.9       0.0405944200315295
```

ALGORITHM AND COMMENTS

Implemented Domain: $-\infty < x < 103.892$
Range: $-0.4190155 < y < 0.5356567$
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 56

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 10.4

■ bei

FUNCTION

y = bei(n, x)

PURPOSE

Compute the imaginary Bessel Kelvin function of integer order n, bei(n, x), at x

INPUT

n (Integer Scalar): the order of the function

x (Real Scalar): the value of the argument of the imaginary Bessel Kelvin function bei(n, x)

OUTPUT

y (Real Scalar): the computed value of bei(n, x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Imaginary
// Bessel Kelvin Function bei(n, x)

project  n, x, y;
local i, p;

p = getNumber("Enter the number of values the function is to
   be evaluated at:","1");

for i = 1 to  p do
   n[i] = getNumber("Enter the value of n: ","1");
    x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = bei(n[i], x[i]);
end for;

// Results:for n=0
// x:  0     y:  0
//     1         0.24956604003666
//     2         0.972291627306661
//     3         1.93758678526604
//     4         2.2926903226993
//     5         0.1160343815502
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-\infty < y < \infty$
The degree, n, has to be a non-negative integer
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 55

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 9.9

■ ber

FUNCTION

y = ber(n, x)

PURPOSE

Compute the real Bessel Kelvin function of integer order n, ber(n, x), at x

INPUT

n (Integer Scalar): the order of the real Bessel Kelvin function

x (Real Scalar): the value of the argument of the real Bessel Kelvin function

OUTPUT

y (Real Scalar): the computed value of ber(n, x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Real Bessel
// Kelvin Function ber(n, x)

project n, x, y;
local i,  m;

m = getNumber("Enter the number of  values the function is to
   be evaluated at:","1");

for i = 1 to  m do
   n[i]= getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = ber(n[i], x[i]);
end for;

// Results:for n=0
// x:  0     y:  1
//     1         0.984381781213087
//     2         0.751734182713808
//     3         -0.221380249598694
//     4         -2.56341655725858
//     5         -6.23008247866636
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-\infty < y < \infty$
The degree, n, has to be a non-negative integer
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 55

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 9.9

■ beta

FUNCTION

z = beta(x, y)

PURPOSE

Compute the Beta function B(x, y) of x and y, defined by:

$$B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1}dt$$

INPUT

x (Real Scalar): the first argument of the function B(x, y)

y (Real Scalar): the second argument of the function B(x, y)

OUTPUT

z (Real Scalar): the computed value of B(x, y)

EXAMPLES

```
// Interactive HiQ-Script Example for
// the Beta Function of x and y, beta(x, y)

project x, y, z;
local i,  m;

m = getNumber("Enter the number of values the function is to
   be evaluated at:","1");

for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
    y[i] = getNumber("Enter the value of y: ","1.0");
   z[i] = beta(x[i],y[i]);
end for;

// Results:
//  Set m = 2:
//  x:  2    y:  3
//      4          4
```

```
//  z: 0.08333333333333333
//     0.00714285714285714
```

ALGORITHM AND COMMENTS

Domain: 0< x, y < ∞; Range: 0 < z < ∞
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, p. 419

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 6

■ bi

FUNCTION

y = bi(x)

PURPOSE

Compute the Airy function bi(x) defined by:

$$\mathrm{bi}(x) = \frac{1}{\pi}\int_0^\infty \exp\left(xt - \frac{t^3}{3}\right)dt + \frac{1}{\pi}\int_0^\infty \sin\left(xt - \frac{t^3}{3}\right)dt = \begin{cases} \sqrt{\frac{x}{3}}\left[I_{-\frac{1}{3}}(\zeta) - I_{\frac{1}{3}}(\zeta)\right], & x>0 \\ \sqrt{\frac{-x}{3}}\left[J_{-\frac{1}{3}}(\zeta) - J_{\frac{1}{3}}(\zeta)\right], & x<0 \end{cases}$$

where $\zeta = 2|x|^{3/2}/3$ is the argument of the Bessel's functions I and J.

INPUT

x (Real Scalar): the value of the argument of bi(x)

OUTPUT

y (Real Scalar): the computed value of bi(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Airy Function bi(x)
// Results given for x ranging from -3.0 to -2.1 in steps of //0.1

project x,y;
biPlott(x,y);
```

```
function biPlott(x,y)

project low, up, bigraph;
local  sttep;

low = getNumber("Enter the lower value of x in bi(x):
    ","1.0");
up = getNumber("Enter the upper value of x in bi(x): ","2.0");
    interval = getNumber("Enter the step size:  ","0.1");
x[1] = low;
y[1] = bi(low);
sttep = abs(up -low)/interval;

for i = 2 to  sttep step 1 do
   x[i] =x[i-1] + interval;
   y[i] = bi(x[i]);
end for;

// Plot the results:

bigraph = new2DGraph( "Airy Function bi(x)");
biplot = new2DDataplot("bi(x)",x,y);
addPlot(bigraph, biplot);
end function;

// Results:
// x:  -3    y:  -0.198289626374927
//     -2.9      -0.262584998164697
//     -2.8      -0.319293888938312
//     -2.7      -0.367092111821008
//     -2.6      -0.405008278130034
//     -2.5      -0.432422471840705
//     -2.4      -0.449052276282108
//     -2.3      -0.454928234394365
//     -2.2      -0.450360984168207
//     -2.1      -0.435902348230727
```

ALGORITHM AND COMMENTS

Implemented Domain: -∞ <x<103.892
Range: -0.4549444 < y < ∞
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 56

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 10.4

■ comIGamma

FUNCTION

y = comIGamma(a, x)

PURPOSE

Compute the Complement of the Incomplete Gamma function $\gamma^*(a, x) = 1 - \gamma(a, x)$, of degree a at the point x

INPUT

a (Real Scalar): the degree of the function $\gamma^*(a, x)$

x (Real Scalar): the value of the argument for the function $\gamma^*(a, x)$

OUTPUT

y (Real Scalar): the computed value of $\gamma^*(a, x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the
// comIGamma Function of a and x, comIGamma(a, x)

project a, x, y;
local i, m;

m = getNumber("Enter the number of a values the function is to
   be evaluated at:","1");

for i = 1 to m do
   a[i] = getNumber("Enter the value of a: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = comIGamma(a[i],x[i]);
end for;

// Results:
//  Set m = 2:
// a:  2
//     2
// x:  2
//     5.5
//y:   0.406005849709838
//     0.0265640143500164
```

ALGORITHM AND COMMENTS

Domain: $0 \le x < \infty$; Range: $-\infty < y < \infty$
The degree a is required to be non-negative
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 45

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 6

■ erf

FUNCTION

y = erf(x)

PURPOSE

Compute the Error function erf(x) defined by:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

INPUT

x (Real Scalar): the value of the argument of the Error function erf(x)

OUTPUT

y (Real Scalar): the computed value of erf(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Error Function erf(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
   to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = erf(x[i]);
end for;

// Results:
// x:  -10    y:  -1
//     -1        -0.842700792949715
//     0         0
//     1         0.842700792949715
//     10        1
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-1 \le y \le 1$
Results accurate to 8 decimal places
Probabilists associate erf(x) with the Gauss Probability Integral

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 40

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 7

■ erfc

FUNCTION

y = erfc(x)

PURPOSE

Compute the Complement Error function erfc(x) defined by:

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt = 1 - \text{erf}(x)$$

INPUT

x (Real Scalar): the value of the argument of erfc(x)

OUTPUT

y (Real Scalar): the computed value of erfc(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Complement of the Error Function erfc(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
   to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = erfc(x[i]);
end for;
```

```
// Results:
// x:  -10   y:  2
//     -1        1.84270079294971
//     0         1
//     1         0.157299207050285
//     10        2.08848758376254e-45
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $0 \le y \le 2$
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 40

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 7

■ fHyper

FUNCTION

y = fHyper(a, b, c, x)

PURPOSE

Compute the Gauss Hypergeometric function, ${}_2F_1(a, b, c, x)$, at the point x

INPUT

a (Real Scalar): the first parameter of the Hypergeometric function ${}_2F_1(a, b, c, x)$

b (Real Scalar): the second parameter of the Hypergeometric function ${}_2F_1(a, b, c, x)$

c (Real Scalar): the third parameter of the Hypergeometric function ${}_2F_1(a, b, c, x)$

x (Real Scalar): the value of the argument of the Hypergeometric function ${}_2F_1(a, b, c, x)$

OUTPUT

y (Real Scalar): the computed value of ${}_2F_1(a, b, c, x)$

EXAMPLES

```
//   Interactive HiQ-Script Example for the
//   Gauss Hypergeometric Function fHyper(a, b, c, x)

project a, b, c, x, y;
local i, j, m, n, k, l, o, p;
```

```
p = getNumber("Enter the number of values the function is to be evaluated
at:","1");

SymbolSetType(a,5);
SymbolSetType(b,5);
SymbolSetType(c,5);
SymbolSetType(x,5);
SymbolSetType(y,5);

SymbolSetVectorDim(a,p);
SymbolSetVectorDim(b,p);
SymbolSetVectorDim(c,p);
SymbolSetVectorDim(x,p);
SymbolSetVectorDim(y,p);

for  n= 1 to p do
   a[n] = getNumber("Enter the value of a: ","1.0");
   b[n] = getNumber("Enter the value of b: ","1.0");
   c[n] = getNumber("Enter the value of c: ","1.0");
   x[n] = getNumber("Enter the value of x: ","0.5");
   y[n] = fHyper(a[n], b[n], c[n], x[n]);
end for;

//   Results: For a = 1., b = 2., c = 3.5
//   y:  1.0 y:  5.0
//   y:  0.5 y:  0.0
```

ALGORITHM AND COMMENTS

Domain: The current HiQ version is implemented only for
$-\infty < a, b, c, < \infty$ and $-1 < x < 1$.
Range: $-\infty < y < \infty$
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 60

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 15

■ fSeries

FUNCTION

y = fSeries(a, b, c, x)

PURPOSE

Compute the power series expansion of the Hypergeometric function, ${}_2F_1(a, b, c, x)$, at the point x

INPUT

a (Real Scalar): the first parameter of the series expansion for ${}_2F_1(a, b, c, x)$

b (Real Scalar): the second parameter of the series expansion for ${}_2F_1(a, b, c, x)$

c (Real Scalar): the third parameter of the series expansion for ${}_2F_1(a, b, c, x)$

x (Real Scalar): the value of the argument of the series function ${}_2F_1(a, b, c, x)$

OUTPUT

y (Real Scalar): the computed value of the series expansion for ${}_2F_1(a, b, c, x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Hypergeometric Series Function fSeries(a, b, c, x)

project a, b, c, x;
local i, j, m, n, k, l, o, p;

p = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for  n= 1 to p do
   a[n] = getNumber("Enter the value of a: ","1.0");
   b[n] = getNumber("Enter the value of b: ","1.0");
   c[n] = getNumber("Enter the value of c: ","1.0");
   x[n] = getNumber("Enter the value of x: ","1.0");
   y[n] = fSeries(a[n], b[n], c[n], x[n]);
end for;
```

ALGORITHM AND COMMENTS

Domain: The current HiQ version is implemented only for
$-\infty < a, b, c, < \infty$ and $-1 < x < 1$
Results accurate to 10 decimal places away from singular values
Also known as the Hypergeometric series

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 60

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 15

■ gamma

FUNCTION

y = gamma(x)

PURPOSE

Compute the Gamma function, Γ(x), at the point x

INPUT

x (Real Scalar): the value of the argument for Γ(x)

OUTPUT

y (Real Scalar): the computed value of Γ(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Gamma Function gamma(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
   to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = gamma(x[i]);
end for;

// Results: For a = 1., b = 2., c = 3.5
// x:  -1.5  y:  2.36327180120735
// x:  1.5   y:  0.886226925452758
// x:  4.0   y:  6.0
```

ALGORITHM AND COMMENTS

Domain: $\infty < x < \infty$, $x \neq 0,-1,-2,\ldots$; Range: $-\infty < y < \infty$ and $y \neq 0$
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 43

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 6

■ iBeta

FUNCTION

z = iBeta(a, x, y)

PURPOSE

Compute the Incomplete Beta function, $I_a(x, y)$, of x and y with parameter a, defined by:

$$I_a(x, y) = \frac{1}{B(x, y)} \int_0^a t^{x-1}(1-t)^{y-1} dt = \frac{\Gamma(x+y)}{\Gamma(x)\Gamma(y)} \int_0^a t^{x-1}(1-t)^{y-1} dt$$

INPUT

a (Real Scalar): the incomplete beta function parameter a

x (Real Scalar): the first argument of the incomplete beta function $I_a(x, y)$

y (Real Scalar): the second argument of the incomplete beta function $I_a(x, y)$

OUTPUT

z (Real Scalar): the computed value of $I_a(x, y)$

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Incomplete Beta Function iBeta(a, x, y)

project a, x, y;
local i,  k;

k = getNumber("Enter the number of values the function is to
    be evaluated at:","1");
for i = 1 to k do
   a[i] = getNumber("Enter the value of a: ","1.0");
   x[i]= getNumber("Enter the value of x: ","1.0");
   y[i] = getNumber("Enter the value of y: ","1.0");
   z = iBeta(a[i], x[i], y[i]);
end for;

// Results:
// a=0.5  x =0.7 y=0.3  z=0.272428440729948
// a=1.   x =3.  y=4.   z=1.0
```

ALGORITHM AND COMMENTS

Domain: $0 < x, y < \infty$; Range: $0 \le z \le 1$
The degree a satisfies $0.0 \le a \le 1.0$
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 58

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 6

■ iGamma

FUNCTION

y = iGamma(a, x)

PURPOSE

Compute the Incomplete Gamma function of degree a, $\gamma(a, x)$, at the point x

$$\gamma(a, x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

INPUT

a (Real Scalar): the degree of the function

x (Real Scalar): the value at which the $\gamma(a, x)$ function is computed

OUTPUT

y (Real Scalar): the computed value of $\gamma(a, x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the Incomplete
// Gamma Function of a and x, iGamma(a, x)

project a, x, y;
local i,  m;

m = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for i = 1 to m do
   a[i] = getNumber("Enter the value of a: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = iGamma(a[i], x[i]);
end for;

// Results:
// a=4.x =10.    z=0.989663949324074
// a=1.x =0.5    z=0.393469340287367
```

ALGORITHM AND COMMENTS

Domain: $a > 0$, $0 \le x < \infty$; Range: $0 \le y \le 1$
The degree a is required to be positive
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 45

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 6

■ in

FUNCTION

y = in(n, x)

PURPOSE

Compute the Modified Bessel function of degree n, $I_n(x)$, of the first kind at the point x

INPUT

n (Integer Scalar): the degree of the function $I_n(x)$

x (Real Scalar): the value of the argument of the Modified Bessel function $I_n(x)$

OUTPUT

y (Real Scalar): the computed value of $I_n(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the Modified
// Bessel Function of the First Kind of n and x, in(n, x)

project n, x, y;
local i, m;

m = getNumber("Enter the number of values the function is to
    be evaluated at:","1");
for i = 1 to m do
   n[i] = getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = in(n[i], x[i]);
end for;

// Results:
// n=1 x =1.     z=0.565159103992485
// n=2 x =1.     z=0.135747669767038
// n=0 x =0.     z=1.
```

ALGORITHM AND COMMENTS

Domain: $0 \le x < \infty$; Range: $-\infty < y < \infty$

The degree, n, has to be non-negative

Results accurate to 11 decimal places
Also known as General Hyperbolic Bessel Function

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chaps. 49, 50

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 9

■ jn

FUNCTION

y = jn(n, x)

PURPOSE

Compute the Bessel function of degree n, $J_n(x)$, of the first kind at the point x

INPUT

n (Integer Scalar): the degree of the Bessel function $J_n(x)$

x (Real Scalar): the value of the argument of the Bessel function $J_n(x)$

OUTPUT

y (Real Scalar): the computed value of $J_n(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the Bessel
// Function of the First Kind of n and x, jn(n, x)
project n, x, y;
local i, m;
m = getNumber("Enter the number of values the function is to
    be evaluated at:","1");
for i = 1 to m do
   n[i] = getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = jn(n[i], x[i]);
end for;

// Results (see AS, p. 385) :
// n=0 x =1.55   z=0.483764428364631
// n=1 x =1.55   z=0.564424467949266
// n=2 x =1.55   z=0.244525207698938
// n=4 x =1.55   z=0.0133134740269008
// n=6 x =1.55   z=0.000276064956736825
// n=8 x =1.55   z=3.01864797547905e-06
```

ALGORITHM AND COMMENTS

Domain: all real x; Range: $-\infty < y < \infty$
The degree, n, has to be non-negative
Results accurate to 11 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 53

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chaps. 9, 10

■ js

FUNCTION

y = js(n, x)

PURPOSE

Compute the Spherical Bessel function of degree n, $j_n(x)$, of the first kind defined by:

$$j_n(x) = \sqrt{\frac{\pi}{2x}}\, J_{n+1/2}(x)$$

where $J_{n+1/2}$ is the Bessel Function of the First Kind of degree n + 1/2

INPUT

n (Real Scalar): the degree of the Spherical Bessel function $j_n(x)$

x (Real Scalar): the argument of the Spherical Bessel function $j_n(x)$

OUTPUT

y (Real Scalar): the computed value of $j_n(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the Spherical
// Bessel Function of the First Kind of n and x, js(n, x)

project  ni, x, y;
local i, m;
m = getNumber("Enter the number of values the function is t
    be evaluated at:","1");

for i = 1 to m do
   ni[i] = getNumber("Enter the value of n: ","1");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = js(ni[i], x[i]);
end for;
```

```
// Results :
//  n=0 x =1.55   z=0.645021783347972
//  n=1 x =1.55   z=0.402727068093471
//  n=4 x =1.55   z=0.00547093704977624
//  n=6 x =1.55   z=9.46840627243189e-05
```

ALGORITHM AND COMMENTS

Domain: 0 < x < ∞ ; Range: -∞ < y < ∞
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 53

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chaps. 10

■ kei

FUNCTION

y = kei(n, x)

PURPOSE

Compute the imaginary Kelvin function of integer degree n,

kei(n, x), at the point x

INPUT

n (Integer Scalar): the degree of the imaginary Kelvin function

x (Real Scalar): the value of the argument of the imaginary Kelvin function

OUTPUT

y (Real Scalar): the computed value of kei(n, x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Imaginary
// Kelvin Function of n and x, kei(n, x)

project ni, x, y;
local i,  m;

m = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for i = 1 to  m do
```

```
   ni[i] = getNumber("Enter the value of n: ","1");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = kei(ni[i], x[i]);
end for;

// Results:    For  n = 0 (see Table 55.7.1 of Spanier & Oldham)
// x =3.914668  z=1.08783053886998e-08
// x =4.931811  z=0.0112160741998966
// x =8.344225  z=5.2408440434637e-11
// x =13.85827  z= 1.28042677971507e-05
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; Range: $-\infty < y < \infty$
The degree, n, has to be a non-negative integer
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 55

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 9

■ ker

FUNCTION

y = ker(n, x)

PURPOSE

Compute the real Kelvin function of integer degree n, ker (n, x), at the point x

INPUT

n (Integer Scalar): the degree of the real Kelvin function

x (Real Scalar): the value of the argument of the real Kelvin function

OUTPUT

y (Real Scalar): the computed value of ker(n, x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Real
// Kelvin Function of n and x, ker(n, x)

project ni, x, y;
local i,  m;
```

```
m = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for i = 1 to  m do
   ni[i] = getNumber("Enter the value of n: ","1");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = ker(ni[i], x[i]);
end for;

// Results:    For  n = 0 (see Table 55.7.1 of Spanier & Oldham)
// x =1.718543  z=-7.8922117431276e-09
// x =2.665845  z=-0.0710236909556133
// x =7.172120  z=0.00195668099677347
// x =10.56294  z=4.18182335981676e-10
// x =11.63219  z= -6.70596899833842e-05
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; Range: $-\infty < y < \infty$
The degree, n, has to be a non-negative integer
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 55

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 9

■ kn

FUNCTION

y = kn(n, x)

PURPOSE

Compute the Modified Bessel function of degree n, $K_n(x)$, of the second kind at the point x

INPUT

n (Integer Scalar): the degree of the function $K_n(x)$

x (Real Scalar): the value of the argument of the function $K_n(x)$

OUTPUT

y (Real Scalar): the computed value of $K_n(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the Modified
// Bessel Function of the Second Kind of degree n, kn(n, x)

project ni, x, y;
local i,  m;

m = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for i = 1 to  m do
   ni[i] = getNumber("Enter the value of n: ","1");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = kn(ni[i], x[i]);
end for;
// Results:    For  n = 0 (see p. 505  of Spanier & Oldham)
// n=16   x =5.     z=186233.58279922
// n=2    x =5.     z=0.00530894371222346
// n=50   x =100.   z=9.27452265361333e-40
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; Range: $0 < y < \infty$
The degree, n, has to be a non-negative integer
Results accurate to 11 decimal places
Also known as the Basset Function

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chaps. 51

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 9

■ lnGamma

FUNCTION

y = lnGamma(x)

PURPOSE

Compute the natural logarithm of the Gamma function, $\ln(\Gamma(x))$, at the point x

INPUT

x (Real Scalar): the value of the argument for $\ln(\Gamma(x))$

OUTPUT

y (Real Scalar): the computed value of $\ln(\Gamma(x))$

EXAMPLES

```
// Interactive HiQ-Script Example for the natural
// logarithm of the Gamma Function lnGamma(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
   to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = lnGamma(x[i]);
end for;

//  Results:   x:  3 rows
//              1e-05
//              1
//             10

//              y:  3 rows
//             11.5129196928958
//              0
//             12.8018274800815
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; Range: $-\infty < y < \infty$ and $y \neq 0$
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 43

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 6

■ mHyper

FUNCTION

y = mHyper(a, b, x)

PURPOSE

Compute the Confluent Hypergeometric function (or Kummer function),
M(a, b, x), at the point x

INPUT

a (Real Scalar): the first parameter of the function M(a, b, x)

b (Real Scalar): the second parameter of the function M(a, b, x)

x (Real Scalar): the value of the argument of the function M(a, b, x)

OUTPUT

y (Real Scalar): the computed value of the function M(a, b, x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Confluent
// Hypergeometric Function M(a, b, x)

project a, b, x;
local n, p;

p = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for  n= 1 to p do
   a[n] = getNumber("Enter the value of a: ","1.0");
   b[n] = getNumber("Enter the value of b: ","1.0");
   x[n] = getNumber("Enter the value of x: ","1.0");
   y[n] = mHyper(a[n], b[n],  x[n]);
end for;

// Results:     (see p. 466 of Spanier & Oldham)
// a=0.7` b =0.6 x=2.   z=8.94061152386701
// a=-4.  b =2.  x=-1.  z=4.175
// a=0.5. b =1.  x=-2.  z=0.46575960759364
```

ALGORITHM AND COMMENTS

Domain: $-\infty < a, b, x < \infty$; Range: $-\infty < y < \infty$
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 47

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 13

■ mSeries

FUNCTION

y = mSeries(a, b, x)

PURPOSE

Compute the power series expansion of the Confluent Hypergeometric function (or Kummer function), M(a, b, x), at the point x

INPUT

a (Real Scalar): the first parameter of the power series expansion of M(a, b, x)

b (Real Scalar): the second parameter of the power series expansion of M(a, b, x)

x (Real Scalar): the value of the argument of the power series expansion of M(a, b, x)

OUTPUT

y (Real Scalar): the computed value of a series expansion for M(a, b, x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Series Expansion of the
// Confluent Hypergeometric Function M(a, b, x)

project a, b, x;
local n, p;

p = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for  n= 1 to p do
   a[n] = getNumber("Enter the value of a: ","1.0");
   b[n] = getNumber("Enter the value of b: ","1.0");
   x[n] = getNumber("Enter the value of x: ","1.0");
   y[n] = mSeries(a[n], b[n],  x[n]);
end for;

// Results:     (see p. 466  of Spanier & Oldham)
// a=0.7  b =0.6 x=2.   z=8.94061152386701
// a=-4.  b =2.  x=-1.  z=4.175
// a=0.5. b =1.  x=-2.  z=0.46575960759364
```

ALGORITHM AND COMMENTS

Domain: $-\infty < a, b, x < \infty$; Range: $-\infty < y < \infty$

Results accurate to 10 decimal places away from singular values

Also known as the Confluent Hypergeometric series, Kummer series

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 47

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 13

■ psi

FUNCTION

y = psi(x)

PURPOSE

Compute the Psi function (also known as the Digamma function), $\psi(x)$ defined by:

$$\psi(x) = \frac{\Gamma'(x)}{\Gamma(x)} = \frac{d}{dx}\ln\Gamma(x)$$

where $\Gamma(x)$ is the Gamma function

INPUT

x (Real Scalar): the argument of the function $\psi(x)$

OUTPUT

y (Real Scalar): the computed value of $\psi(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Psi Function psi(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
   to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = psi(x[i]);
end for;

// Results:     (see Table 44.7.1 of Spanier & Oldham)
//  x=-1.5       z=0.703156640645243
//  x=1.         z=-0.577215664901533
//  x=2          z=0.422784335098467
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$, $x \neq 0, -1, -2, \ldots$; Range: $-\infty < y < \infty$;
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 44

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 6

■ struve

FUNCTION

y = struve(ν, x)

PURPOSE

Compute the Struve function $\mathbf{H}_{\nu}(x)$ defined by:

$$H_{\nu}(x) = \left(\frac{x}{2}\right)^{\nu+1} \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{x}{2}\right)^{2k}}{\Gamma\left(k+\frac{3}{2}\right)\Gamma\left(k+\nu+\frac{3}{2}\right)} = \frac{\left(\frac{x}{2}\right)^{\nu+1}}{\Gamma\left(\frac{3}{2}\right)\Gamma\left(\nu+\frac{3}{2}\right)} {}_1F_2\left(1;\frac{3}{2};\nu+\frac{3}{2};\frac{-x^2}{4}\right)$$

where $\Gamma(x)$ is the Gamma function and ${}_1F_2()$ is the Gauss Hypergeometric function.

For integer $\nu = n$, we have:

$$H_{-n-1/2}(x) = (-1)^n J_{n+1/2}(x)$$

INPUT

ν (Real Scalar): the index parameter of $\mathbf{H}_{\nu}(x)$

x (Real Scalar): the value of the argument of $\mathbf{H}_{\nu}(x)$

OUTPUT

y (Real Scalar): the computed value of $\mathbf{H}_{\nu}(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Struve Function with parameter nu, struve(nu, x)

project  nu, x, y;
local i, j, m, k;
```

```
m = getNumber("Enter the number of values the function is to
    be evaluated at:","1");


for i = 1 to  m do
   n[i] = getNumber("Enter the value of nu: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = struve(n[i], x[i]);
end for;

/ Results:    (see p. 568 of Spanier & Oldham)
// n=1      x =5.                z=0.807811945794064
// n=-2.5    x =3.14159265358979 z=0.429869376188067
```

ALGORITHM AND COMMENTS

Real Domain: $-\infty < x < \infty$
Real Range: $-\infty < y < \infty$
Negative values of x for $\mathbf{H}_{\nu}(x)$ are not computed unless ν is an integer
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 57

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 12

■ uHyper

FUNCTION

y = uHyper(a, b, x)

PURPOSE

Compute the associated Confluent Hypergeometric function (or Tricomi function), U(a, b, x), at the point x

INPUT

a (Real Scalar): the first parameter of the function U(a, b, x)

b (Real Scalar): the second parameter of the function U(a, b, x)

x (Real Scalar): the value of the argument of the function U(a, b, x)

OUTPUT

y (Real Scalar): the computed value of the function U(a, b, x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Associated
// Confluent Hypergeometric Function U(a, b, x)

project a, b, x;
local n, p;

p = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for  n= 1 to p do
   a[n] = getNumber("Enter the value of a: ","1.0");
   b[n] = getNumber("Enter the value of b: ","1.0");
   x[n] = getNumber("Enter the value of x: ","1.0");
   y[n] = uHyper(a[n], b[n],  x[n]);
end for;

// Results:     (see p. 475 of Spanier & Oldham)
//  a=0.5  b =0.5 x=9.4247779607   z=0.310662325801363
```

ALGORITHM AND COMMENTS

Domain: $-\infty < a, b < \infty$, $0 < x < \infty$ b is not an integer (function uSeries oversomes this restriction) Range: $-\infty < y < \infty$

Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 48

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 13

■ uSeries

FUNCTION

y = uSeries(a, b, x)

PURPOSE

Compute the power series expansion of the associated Confluent Hypergeometric function (or Tricomi function), U(a, b, x), at the point x

INPUT

a (Real Scalar): the first parameter of the series expansion for U(a, b, x)

b (Real Scalar): the second parameter of the series expansion for U(a, b, x)

x (Real Scalar): the value of the argument of the series function U(a, b, x)

OUTPUT

y (Real Scalar): the computed value of the series expansion for U(a, b, x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Series Expansion of the
// Associated Confluent Hypergeometric Function U(a, b, x)
project a, b, x;
local n, p;

p = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for  n= 1 to p do
   a[n] = getNumber("Enter the value of a: ","1.0");
   b[n] = getNumber("Enter the value of b: ","1.0");
   x[n] = getNumber("Enter the value of x: ","1.0");
   y[n] = uSeries(a[n], b[n],  x[n]);
end for;

// Results:     (see p. 475 of Spanier & Oldham)
//  a=0.5b =0.5x=9.4247779607z=0.310662340619204
//  a=1.5b =3.x=5.   z=0.10157380794697
//  a=1.b =1.x=20.   z=0.0477189510787948
```

ALGORITHM AND COMMENTS

Domain:$-\infty < a, b, x < \infty$; Range: $-\infty < y < \infty$
Results accurate to 10 decimal places away from singular values
Also known as the associated Confluent Hypergeometric series, Tricomi series

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 48

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, *Dover Publ.*, 1972, chap. 13

■ weber

FUNCTION

y = weber (r, x)

PURPOSE

Compute the Parabolic Cylinder function (also known as Weber function) $D_r(x)$ defined by:

$$D_r(x) = \sqrt{\frac{2}{\pi}} e^{x^2/4} \int_0^{\infty} t^r e^{-t^2/2} \cos\left(xt - \frac{r\pi}{2}\right) dt \qquad \text{if } r > -1$$

and

$$D_r(x) = \frac{1}{\Gamma(-r)} e^{-x^2/4} \int_0^{\infty} \frac{e^{(-t^2/2 - xt)}}{t^{r+1}} dt \qquad \text{if } r < 0$$

for real order r at a real value x, where Γ denotes the gamma function

INPUT

r (Real Scalar): the order of the parabolic cylinder function

x (Real Scalar): the value at which the parabolic cylinder function is computed

OUTPUT

y (Real Scalar) : the computed $D_r(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the Weber
// Function weber(n, x)

project  n, x, y;
local i, p;

p = getNumber("Enter the number of values the function is to
    be evaluated at:","1");

for i = 1 to  p do
   n[i] = getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = weber(n[i], x[i]);
end for;

// Results:
//  n:  -1.2     x: 1       y:  0.455532747080671
//       0          3           0.105399224437244
//       2.5        2           1.0920159710597
```

ALGORITHM AND COMMENTS

Domain:$-\infty < r, x < \infty$; Range: $-\infty < y < \infty$

REFERENCE

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, pp. 445-450

■ y n

FUNCTION

y = yn(n, x)

PURPOSE

Compute the Bessel function of degree n, $Y_n(x)$, of the second kind at the point x

INPUT

n (Real Scalar): the degree of the Bessel function $Y_n(x)$

x (Real Scalar): the value of the argument of the Bessel function $Y_n(x)$

OUTPUT

y (Real Scalar): the computed value of $Y_n(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the Bessel
// Function of the Second Kind of n and x, yn(n, x)

project n, x, y;
local i, m;

m = getNumber("Enter the number of values the function is to
    be evaluated at:","1");
for i = 1 to m do
   n[i] = getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
y[i] = yn(n[i], x[i]);
end for;

// Results:    (see p. 541 of Spanier & Oldham)
// n=1/3  x=4.9383758  z=-0.16366143543485
// n=-1   x=5.         z=-0.147863143391227
// n=7    x=50.        z=0.0959120278245426
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; Range: $-\infty < y < \infty$
The degree, n, has to be a real number > -200, unless n is an integer
Results accurate to 11 decimal places
Also known as Neumann Function

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 54

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chaps. 9, 10

■ ys

FUNCTION

y = ys(n, x)

PURPOSE

Compute the Spherical Bessel function of degree n, $y_n(x)$, of the second kind defined by:

$$y_n(x) = \sqrt{\frac{\pi}{2x}} Y_{n+1/2}(x)$$

where $Y_{n+1/2}$ is the Bessel Function of the Second Kind of degree n+1/2

INPUT

n (Real Scalar): the degree of the Spherical Bessel function $y_n(x)$

x (Real Scalar): the value of the argument of the Spherical Bessel function $y_n(x)$

OUTPUT

y (Real Scalar): the computed value of $y_n(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the Spherical
// Bessel Function of the Second Kind of n and x, ys(n, x)

project n, x, y;
local i, m;

m = getNumber("Enter the number of values the function is to
    be evaluated at:","1");
for i = 1 to m do
   n[i] = getNumber("Enter the value of n: ","1.0");
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = ys(n[i], x[i]);
end for;

// Results:

//  n=2    x= 3.14159265358979  z=-0.221555282884192
//  n=4    x= 3.14159265358979  z=-0.789893980789581
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; Range: $-\infty < y < \infty$
Results accurate to 8 decimal places, except near x = 0

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 53

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 10

■ zeta

FUNCTION

y = zeta(x)

PURPOSE

Compute the zeta function (also known as the Riemann zeta function) $\zeta(x)$ defined by:

$$\zeta(x) = \sum_{k=1}^{\infty} k^{-x}$$

INPUT

x (Real Scalar): the value argument of the function $\zeta(x)$

OUTPUT

y (Real Scalar): the computed value of $\zeta(x)$

EXAMPLES

```
// Interactive HiQ-Script Example for the Riemann
// Zeta Function zeta(x)

project  x, y;
local i,  m;
m = getNumber("Enter the number of x values the function is
   to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = zeta(x[i]);
end for;
// Results:     (see p. 29 of Spanier & Oldham)
// n=-3   z=0.00833333333333333
// n=-1   z=0.08333333333333333
// n=5    z=ß734306198445
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$ and $x \neq 1$; Range: $-\infty < y < \infty$
Infinite singularity at $x = 0$;
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 3

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, Chap. 23

CHAPTER 6
INTEGRAL FUNCTIONS

■ cn

FUNCTION

y = cn(u, k)

PURPOSE

Compute the Jacobi elliptic function cn(u,k) defined by:

$$cn(u, k) = \cos(\phi)$$

where

$$u = \int_0^{\phi} \frac{d\theta}{\sqrt{1 - k\sin^2\theta}}$$

i.e., cn(u, k) is a periodic function (with period 4K(k), where K(k) is the complete elliptic integral of the first kind) defined in terms of the inverse of the incomplete elliptic integral of the first kind, F(ϕ, k)

INPUT

u (Real Scalar): the value of the argument of the function cn(u, k)

k (Real Scalar): the integrand parameter of the function cn(u, k)

OUTPUT

y (Real Scalar): the computed value of cn(u, k)

EXAMPLES

```
// Interactive HiQ-Script Example for the Jacobi
// Elliptic Function of u with parameter k, cn(u, k)

project  k, u, y;
local i, j, m, o;

m = getNumber("Enter the number of k values the function is
   to be evaluated at:","1");
o = getNumber("Enter the number of u values the function is to
   be evaluated at:","1");
```

```
for i = 1 to m do
   for  j = 1 to o do
      k = getNumber("Enter the value of k: ","1.0");
      u[j] = getNumber("Enter the value of u: ","1.0");
      y[j] = cn(u[j], k);
   end for;
end for;

Result:
//     For the number of k values to be evaluated  m = 1, the number
//      of u values to be evaluated o = 1, with the argument value
//       u = 1.0, and the parameter value k = 0.5
//
//     y:  1 rows
//       0.595976567672141
```

ALGORITHM AND COMMENTS

Domain: $-\infty < u < \infty$, $0 \le k \le 1$; Range: $-1 \le y \le \infty$
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 63

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 16

■ comel1

FUNCTION

y = comel1(k)

PURPOSE

Compute the Complete Elliptic Integral of the First Kind:

$$K(k) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - k\sin^2\theta}}$$

where $k = \sin^2\alpha$, α is the modular angle. Note that the parameter k is not squared in the integrand, as it is in the definition of the Jacobi Elliptic functions; i.e., we use the Abramowitz and Stegun form of K (see references)

INPUT

k (Real Scalar): the parameter of the function K(k)

OUTPUT

y (Real Scalar): the computed value of K(k)

EXAMPLES

```
//  Interactive HiQ-Script Example for the Complete
//   Elliptic Integral Function of the First Kind, comel1(k)

project  k, y;
local i,  m;

m = getNumber("Enter the number of values the function is to be evaluated
at:","1");

SymbolSetType(k,5);
SymbolSetType(y,5);

SymbolSetVectorDim(k,m);
SymbolSetVectorDim(y,m);

for i = 1 to m do
   k[i] = getNumber("Enter the value of k: ","1.0");
   y[i] = comel1(k[i]);
end for;

//  Results :
//     For the number of values to be evaluated m = 2, and the
//      parameter values k = 0.5 and 0
//
//     y:  2 rows
//         1.85407467730137
//         1.5707963267949
```

ALGORITHM AND COMMENTS

Domain: $0 \leq k < 1$
Range: $\pi/2 \leq y < \infty$
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 61

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 17

■ comel2

FUNCTION

y = comel2(k)

PURPOSE

Compute the Complete Elliptic Integral of the Second Kind:

$$E(k) = \int_0^{\pi/2} \sqrt{(1 - k\sin^2\theta)}\, d\theta$$

where $k = \sin^2\alpha$, α is the modular angle. Note that the parameter k is not squared in the integrand, as it is in the definition of the Jacobi Elliptic functions; i.e., we use the Abramowitz and Stegun form of E (see references)

INPUT

k (Real Scalar): the parameter of the function E(k)

OUTPUT

y (Real Scalar): the computed value of E(k)

EXAMPLES

```
//  Interactive HiQ-Script Example for the Complete
//   Elliptic Integral Function of the Second Kind, comel2(k)

project  k, y;
local i,  m;

m = getNumber("Enter the number of values the function is to be evaluated
at:","1");

SymbolSetType(k,5);
SymbolSetType(y,5);

SymbolSetVectorDim(k,m);
SymbolSetVectorDim(y,m);

for i = 1 to m do
   k[i] = getNumber("Enter the value of k: ","1.0");
   y[i] = comel2(k[i]);
end for;

//  Results:
//     For the number of k values to be evaluated  m = 3, and the
```

```
//      parameter values k = 0.7, 0.4 and 0
//
//      y:  3 rows
//         1.24167056794582
//         1.39939213889743
//         1.5707963267949
```

ALGORITHM AND COMMENTS

Domain: $0 \le k \le 1$
Range: $1 \le y \le \pi/2$
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 61

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 17

■ cosI

FUNCTION

y = cosI(x)

PURPOSE

Compute the Cosine Integral function ci(x) defined by:

$$ci(x) = -\int_x^\infty \frac{\cos(t)}{t} dt = \gamma + \ln(|x|) + \sum_{k=1}^{\infty} \frac{(-x^2)^k}{2k(2k)!}$$

where γ is Euler's constant

INPUT

x (Real Scalar): the value of the argument for cosI(x)

OUTPUT

y (Real Scalar): the computed value of cosI(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Cosine
// Integral Function, cosI(x)

project  x, y;
local i,  m;
```

```
m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = cosI(x[i]);
end for;

//  Results:
//     For the number of x values to be evaluated  m = 2, and the
// argument values  x = -10 and 10
//
//     y:  2 rows
//      -0.0454564330044554
//      -0.0454564330044554
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; Range: $-\infty < y < 0.4727$
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 38

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 5.2

■ daw

FUNCTION

y = daw(x)

PURPOSE

Compute Dawson's Integral function daw(x) defined by:

$$\mathrm{daw}(x) = e^{-x^2}\int_0^x e^{t^2}dt$$

INPUT

x (Real Scalar): the value of the argument for daw(x)

OUTPUT

y (Real Scalar): the computed value of daw(x)

EXAMPLES

```
// Interactive HiQ-Script Example for Dawson's
// Integral Function, daw(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = daw(x[i]);
end for;

//  Results:
//     For the number of x values to be evaluated  m = 3 and the
//     the argument values x = -3, -1, 4
//
//     y:  3 rows
//       -0.178271030610558
//       -0.538079506912768
//        0.129348001236005
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-0.5441044225 < y < 0.5441044225$
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 42

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 7

■ dilog

FUNCTION

y = dilog(x)

PURPOSE

Compute the Dilogarithm function dilog(x) (also known as Spence's Integral) defined by:

$$\text{dilog}(x) = -\int_1^x \frac{\ln(t)}{t-1}\,dt$$

INPUT

x (Real Scalar): the value of the argument for dilog(x)

OUTPUT

y (Real Scalar): the computed value of dilog(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Dilogarithm Function, dilog(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
      to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = dilog(x[i]);
end for;

//  Results:
//     For the number of x values to be evaluated  m = 2 and the
//     the argument values x = 1,100
//
//    y:  2 rows
//          0
//        -12.1924216690332
```

ALGORITHM AND COMMENTS

Domain: $x \geq 0$; Range: $-\infty < y < 1.6449340668$
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 25

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, pp. 1004-1005

■ dn

FUNCTION

y = dn(u, k)

PURPOSE

Compute the Jacobi elliptic function dn(u,k) defined by:

$$dn(u, k) = \sqrt{1 - k^2 \sin^2 \Phi}$$

where

$$u = \int_0^{\phi} \frac{d\theta}{(1 - k^2 \sin^2 \theta)^{1/2}}$$

i.e., dn(u, k) is a periodic function (with period 2K(k), where K(k) is the complete elliptic integral of the first kind) defined in terms of the inverse of the incomplete elliptic integral of the first kind, F(ϕ, k)

INPUT

u (Real Scalar): the argument of the function dn(u, k)

k (Real Scalar): the parameter of the function dn(u, k)

OUTPUT

y (Real Scalar): the computed value of dn(u, k)

EXAMPLES

```
// Interactive HiQ-Script Example for the Jacobi
// Elliptic Function of u with parameter k, dn(u, k)

project  k, u, y;
local i, j, m, o;

m = getNumber("Enter the number of k values the function is to
   be evaluated at:","1");
o = getNumber("Enter the number of u values the function is to
   be evaluated at:","1");
for i = 1 to m do
   for  j = 1 to o do
      k = getNumber("Enter the value of k: ","1.0");
      u[j] = getNumber("Enter the value of u: ","1.0");
      y[j] = dn(u[j], k);
   end for;
end for;

//  Result:
//     For the number of k values to be evaluated  m = 1, the number
//      of u values to be evaluated o = 1, with the argument value
//       u = 1 , and the parameter value k = 0.5
//
//     y:  1 row
//       0.823161001631596
```

ALGORITHM AND COMMENTS

Domain: $-\infty < u < \infty$, $0 \le k \le 1$

Range: $(1 - k^2)^{1/2} \le y \le 1$ Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 63

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 16

■ el1

FUNCTION

y = el1(p, k)

PURPOSE

Compute the (Incomplete) Elliptic Integral of the First Kind:

$$F(p|k) = \int_0^p \frac{d\theta}{\sqrt{1 - k\sin^2\theta}}$$

where $k = \sin^2\alpha$, α is the modular angle. Note that the parameter k is not squared in the integrand, as it is in the definition of the Jacobi Elliptic functions; i.e., we use the Abramowitz and Stegun form of F (see references)

INPUT

p (Real Scalar): the argument of the function F(p|k)

k (Real Scalar): the parameter of the function F(p|k)

OUTPUT

y (Real Scalar): the computed value of F(p|k)

EXAMPLES

```
// Interactive HiQ-Script Example for the Elliptic
// Integral Function of the First Kind, el1(p, k)

project  k, p, y;
local i, j, m, o;

m = getNumber("Enter the number of k values the function is to
   be evaluated at:","1");
o = getNumber("Enter the number of p values the function is to
   be evaluated at:","1");
```

```
for i = 1 to m do
   for  j = 1 to o do
      k = getNumber("Enter the value of k: ","1.0");
      p[j] = getNumber("Enter the value of p: ","1.0");
      y[j] = el1(p[j], k);
   end for;
end for;

//  Result:
//     For the number of k values to be evaluated  m = 1, the number
//      of p values to be evaluated o = 1, with the argument value
//       p = -1 , and the parameter value k = 0.5
//

//     y:  1 row
//        -1.08321677284517
```

ALGORITHM AND COMMENTS

Domain: $-\pi/2 \le p \le \pi/2$, $0 \le k \le 1$ (if $p = \pi/2$, $0 \le k < 1$)
Range: $-\infty < y < \infty$
Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 62

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 17

■ el2

FUNCTION

y = el2(p, k)

PURPOSE

Compute the (Incomplete) Elliptic Integral of the Second Kind:

$$E(p|k) = \int_0^p \sqrt{1 - k\sin^2\theta}$$

where $k = \sin^2\alpha$, α is the modular angle. Note that the parameter k is not squared in the integrand, as it is in the definition of the Jacobi Elliptic functions; i.e., we use the Abramowitz and Stegun form of E (see references)

INPUT

P (Real Scalar): the argument of the function E(p|k)

k (Real Scalar): the parameter of the function E(p|k)

OUTPUT

y (Real Scalar): the computed value of E(p|k)

EXAMPLES

```
// Interactive HiQ-Script Example for the Elliptic
// Integral Function of the Second Kind, el2(p, k)

project  k, p, y;
local i, j, m, o;

m = getNumber("Enter the number of k values the function is to
   be evaluated at:","1");
o = getNumber("Enter the number of p values the function is to
   be evaluated at:","1");
for i = 1 to m do
   for  j = 1 to o do
      k = getNumber("Enter the value of k: ","1.0");
      p[j] = getNumber("Enter the value of p: ","1.0");
      y[j] = el2(p[j], k);
   end for;
end for;

//  Result:
//     For the number of k values to be evaluated  m = 1, the number
//      of p values to be evaluated o = 1, with the argument value
//       p = 12 , and the parameter value k = 0.5
//
//     y:  1 row
//        10.2533081849568
```

ALGORITHM AND COMMENTS

Domain: $-\pi/2 \le p \le \pi/2$, $0 \le k \le 1$

Range: $-\infty < y < \infty$

Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 62

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 17

■ expI

FUNCTION

y = expI(n, x)

PURPOSE

Compute the Exponential Integral function defined by:

$$E(n, x) = \int_{1}^{\infty} t^{-n} e^{-xt} dt$$

where x > 0, n ≥ 0. This function is a generalization of the classical Exponential Integral function:

$$Ei(x) = \int_{-\infty}^{x} \frac{\exp(t)}{t} dt$$

where E(1, x) = -Ei(-x)

INPUT

n (Integer Scalar): the integer parameter in E(n, x)

x (Real Scalar): the value of the argument for E(n, x)

OUTPUT

y (Real Scalar): the computed value of E(n, x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Exponential
// Integral Function of n and x, expI(n, x)

project  n, x, y;
local i, j, m, k;

m = getNumber("Enter the number of n values the function is to
   be evaluated at:","1");
k = getNumber("Enter the number of x values the function is to
   be evaluated at:","1");

for i = 1 to m do
   for  j = 1 to k do
      n = getNumber("Enter the value of n: ","1");
      x[j] = getNumber("Enter the value of x: ","1.0");
      y[j] = expI(n, x[j]);
```

```
   end for;
end for;

// Result:
// For the number of n values to be evaluated  m = 1, the number
// of x values to be evaluated k = 1, with the integer
//      parameter value n = 10 , and the argument value x = 0.5
//
//     y:  1 row
//        0.0634583004271272
```

ALGORITHM AND COMMENTS

Domain: $x > 0$ if $n = 0, 1$; $x \geq 0$ if $n \geq 2$; Range: $0 < y < \infty$
n is restricted to non-negative integer values
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 37

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 5

■ fCosI

FUNCTION

y = fCosI(x)

PURPOSE

Compute the Fresnel Cosine Integral function C(x) defined by:

$$C(x) = \int_0^x \cos\left(\frac{\pi}{2}t^2\right)dt$$

This definition is not universally standardized; another common form is:

$$C(x) = \sqrt{\frac{2}{\pi}} \int_0^x \cos(t^2)\,dt$$

We follow the Abramowitz and Stegun definition (reference below)

INPUT

x (Real Scalar): the value of the argument for C(x)

OUTPUT

y (Real Scalar): the computed value of C(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Fresnel
// Integral Cosine Function, fCosI(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
                  to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = fCosI(x[i]);
end for;

//  Results:
//     For the number of x values to be evaluated  m = 3 and the
//     the argument values x = -33, 0, 100
//
//    y:  3 rows
//       -0.509645751654485
//        0
//        0.499999898678818
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-0.7798934 \le y \le 0.7798934$
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 39

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 7.3

■ fSinI

FUNCTION

y = fSinI(x)

PURPOSE

Compute the Fresnel Sine Integral function S(x) defined by:

$$S(x) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right)dt$$

This definition is not universally standardized; another common form is:

$$S(x) = \sqrt{\frac{2}{\pi}} \int_0^x \sin(t^2)\,dt$$

We follow the Abramowitz and Stegun definition (reference below)

INPUT

x (Real Scalar): the value of the argument for S(x)

OUTPUT

y (Real Scalar): the computed value of S(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the Fresnel
// Integral Sine Function, fSinI(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
               to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = fSinI(x[i]);
end for;

//  Results:
//     For the number of x values to be evaluated  m = 3 and the
//     the argument values x = -100, 0, 100
//
//    y:  3 rows
//       -0.496816901147838
//        0
//        0.496816901147838
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-0.7139722 \le y \le 0.7139722$
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 39

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions, Dover Publ.*, 1972, chap. 7.3

■ hCosI

FUNCTION

y = hCosI(x)

PURPOSE

Compute the Hyperbolic Cosine Integral function Chi(x) defined by:

$$\mathrm{Chi}(x) = r + \ln|x| + \int_0^x \frac{\cosh t - 1}{t} dt$$

where r denotes the Euler number

INPUT

x (Real Scalar): the value of the argument for Chi(x)

OUTPUT

y (Real Scalar): the computed value of Chi(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Sine Integral Function, hCosI(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
                  to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = hCosI(x[i]);
end for;
```

```
//  Results:
//     For the number of x values to be evaluated  m = 3 and the
//     the argument values x = -3, 2, 10
//
//    y:  3 rows
//       4.96039209476561
//       2.45266692264691
//     246.11448604245
```

ALGORITHM AND COMMENTS

The domain for the approximation of Chi(x) is restricted to $0 < x \leq 88$

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p. 231

■ hSinI

FUNCTION

y = hSinI(x)

PURPOSE

Compute the Hyperbolic Sine Integral function Shi(x) defined by:

$$\mathrm{Shi}(x) = \int_0^x \frac{\sinh(t)}{t}\,dt$$

INPUT

x (Real Scalar): the value of the argument for Shi(x)

OUTPUT

y (Real Scalar): the computed value of Shi(x)

EXAMPLE

```
// Interactive HiQ-Script Example for the
// Sine Integral Function, hSinI(x)

project  x, y;
local i,  m;

m = getNumber("Enter the number of x values the function is
                to be evaluated at:","1");
```

```
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = hSinI(x[i]);
end for;

//  Results:
//     For the number of x values to be evaluated  m = 3 and the
//     the argument values x = -12 ,-0.5, 11
//
//    y:  3 rows
//      -7479.7657983534
//         -0.506996749819667
//       3035.70340919907
```

ALGORITHM AND COMMENTS

The domain for the approximation of Shi(x) is restricted to $-88 \le x \le 88$

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p. 231

■ sinI

FUNCTION

y = sinI(x)

PURPOSE

Compute the Sine Integral function Si(x) defined :

$$Si(x) = \int_0^x \frac{\sin(t)}{t} dt = \sum_{k=0}^{\infty} \frac{x(-x^2)^k}{(2k+1)(2k+1)!}$$

INPUT

x (Real Scalar): the value of the argument for Si(x)

OUTPUT

y (Real Scalar): the computed value of Si(x)

EXAMPLES

```
// Interactive HiQ-Script Example for the
// Sine Integral Function, sinI(x)
```

```
project  x, y;
local i,  m;
m = getNumber("Enter the number of x values the function is
                to be evaluated at:","1");
for i = 1 to m do
   x[i] = getNumber("Enter the value of x: ","1.0");
   y[i] = sinI(x[i]);
end for;
//  Results:
//     For the number of x values to be evaluated  m = 3 and the
//     the argument values x = -3, 0, 100
//    y:  3 rows
//       -1.84865252799947
//        0
//        1.56222546688906
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$; Range: $-1.8519 < y < 1.8519$
Results accurate to 10 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 38

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 5.2

■ sn

FUNCTION

y = sn(u, k)

PURPOSE

Compute the Jacobi elliptic function sn(u,k)defined by:

$$sn(u, k) = \sin(\phi)$$

where

$$u = \int_0^{\phi} \frac{d\theta}{(1 - k^2 \sin^2\theta)^{1/2}}$$

i.e., sn(u, k) is a periodic function (with period 4K(k), where K(k) is the complete elliptic integral of the first kind) defined in terms of the inverse of the incomplete elliptic integral of the first kind, $F(\phi, k)$

INPUT

u (Real Scalar): the value of the argument of sn(u, k)

k (Real Scalar): the parameter of sn(u, k)

OUTPUT

y (Real Scalar): the computed value of sn(u, k)

EXAMPLES

```
// Interactive HiQ-Script Example for the Jacobi
// Elliptic Function of u with parameter k, sn(u, k)

project  k, u, y;
local i, j, m, o;

m = getNumber("Enter the number of k values the function is to
   be evaluated at:","1");
o = getNumber("Enter the number of u values the function is to
   be evaluated at:","1");

for i = 1 to m do
   for  j = 1 to o do
      k = getNumber("Enter the value of k: ","1.0");
      u[j] = getNumber("Enter the value of u: ","1.0");
      y[j] = sn(u[j], k);
   end for;
end for;

//  Result:
//     For the number of k values to be evaluated  m = 1, the number
//      of u values to be evaluated o = 1, with the argument value
//      value u =  1.0 , and the argument value k = 0.5
//
//    y:  1 row
//        0.803001824895644
```

ALGORITHM AND COMMENTS

Domain: $-\infty < u < \infty$, $0 \le k \le 1$; Range: $-1 \le y \le 1$ Results accurate to 8 decimal places

REFERENCES

Spanier, J. and Oldham, K.B., *An Atlas of Functions*, Hemisphere Publ. Corp., 1987, chap. 63

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, chap. 16

CHAPTER 7
INTEGRAL FORMULA FUNCTIONS

■ adSimp

FUNCTION

y = adSimp(integFct, a, b, tolerance, n)

PURPOSE

Compute the definite integral of a real-valued smooth function f(x) over the interval [a,b] using an adaptive Simpson's method

INPUT

integFct (Function): the integrand function f(x)

a (Real Scalar): the lower limit of the definite integral

b (Real Scalar): the upper limit of the definite integral

tolerance (Real Scalar): the accuracy required for the computed integral

n (Integer Scalar): the maximum number of levels for interval splitting allowed in the adaptive Simpson's integration method

OUTPUT

Y (Real Scalar): the computed definite integral of f(x) over the interval [a,b]

EXAMPLES

```
// Example for: adSimp(integFct, a, b, tolerance, n)

// integFct is the user specified function,
// the lower limit a = 0,
//  upper limit b = 1, tolerance = 1.0e-11, and the
//  number of points
//  n = 100:

a = 0;
b = 1;
tolerance = 1e-11;
n = 100;

function integFct(x)
  return sin(x);
end function;
```

```
y = adSimp(integFct, a, b, tolerance, n);

//  Result :
//      y:          0.45969769413483
```

ALGORITHM AND COMMENTS

The adaptive Simpson's method is a recursive procedure which "optimizes" the number of function evaluations required in computing the definite integral of a function f(x) over [a, b] by applying the standard Simpson's rule. The method uses a local (truncation) error estimation to determine if an accurate integral has been achieved over the current interval. If not, the method will split the interval into two equally sized subintervals and recompute the integral for each subinterval. For more details, see the reference list below.

Comments:

1) For a practical implementation of the algorithm, we convert the recursive procedure into a nonrecursive procedure.

2) A minimum number of allowable levels in the adaptive Simpson's method is set to be 100. That is to say any input integer n < 100 for this function will be automatically set to 100.

REFERENCE

Burden, R.L. and Faires, J.D., *Numerical Analysis*, 3rd. ed., Prindle, Weber and Schmidt, Boston, 1985, pp. 174-175

■ chebSing1

FUNCTION

y = chebSing1(integFct, a, b, n)

PURPOSE

Compute the definite integral of a real-valued smooth function f(x) over the interval [a,b] with respect to the singular weight function

$$\frac{1}{\sqrt{(x-a)\ (b-x)}}$$

using the nth degree Chebyshev polynomials of the first kind

INPUT

integFct (Function): the given smooth function f(x)

a (Real Scalar): the lower limit of the definite integral

b (Real Scalar): the upper limit of the definite integral

n (Integer Scalar): the degree of the Chebyshev polynomial of the first kind used to evaluate the integral,

where n is a positive integer

OUTPUT

y (Real Scalar): the computed definite integral of

$$\frac{f(x)}{\sqrt{(x-a)\ (b-x)}}$$

over [a,b]

EXAMPLES

```
//  Example for: chebSing1(integFct, a, b, n)

//  integFct is the user specified function, the lower limi
//  a = 0,the upper limit b = 1, and the degree n = 10:

a = 0;
b = 1;
n = 10;

function integFct(x)
  return sin(x)+1;
end function;

y = chebSing1(integFct, a, b, n);

//  Result :
//          y:           4.55507810386709
```

ALGORITHM AND COMMENTS

The formula for computing the integral of a smooth function f(x) with respect to the weight function:

$$\frac{1}{\sqrt{(x-a)\ (b-x)}}$$

is given by:

$$\int_a^b \frac{f(x)}{\sqrt{(x-a)\ (b-x)}}dx \approx \sum_{i=1}^{n} w_i f(y_i)$$

where

$$y_i = \left(\frac{b-a}{2}\right)x_i + \left(\frac{b+a}{2}\right)$$

x_i is the ith root of the Chebyshev polynomial of the first kind and

$$w_i = \frac{\pi}{n}$$

for i = 1, ... , n.

REFERENCE

Abramowitz, M. and Stegun, I. (eds.), *Handbook of Mathematical Functions*, Dover, New York, 1972, p. 889

■ chebSing2

FUNCTION

y = chebSing2(integFct, a, b, n)

PURPOSE

Compute the definite integral of a real-valued smooth function f(x) over the interval [a,b] with respect to the weight function

$$\sqrt{(x-a)\ (b-x)}$$

using the nth degree Chebyshev polynomials of the second kind

INPUT

integFct (Function): the given smooth function f(x)

a (Real Scalar): the lower limit of the definite integral

b (Real Scalar): the upper limit of the definite integral

n (Integer Scalar): the degree of the Chebyshev polynomial of the second kind used to evaluate the integral, where n is a positive integer

OUTPUT

y (Real Scalar): the computed definite integral of

$$\frac{f(x)}{\sqrt{(x-a)\ (b-x)}}$$

over [a,b]

EXAMPLES

```
//  Example for: chebSing2(integFct, a, b, n)

//  integFct is the user specified function, the lower limit
```

```
//  a = 0, the upper limit b = 1, and the degree n = 10:
a = 0;
b = 1;
n = 10;

function integFct(x)
  return sin(x)+1;
end function;

y = chebSing2(integFct, a, b, n);

//  Result :
//          y:          0.575146581523662
```

ALGORITHM AND COMMENTS

The formula for computing the integral of a smooth function f(x) with respect to the weight function

$$\sqrt{(x-a)\ (b-x)}$$

is given by:

$$\int_a^b f(x)\sqrt{(x-a)\ (b-x)}\,dx \approx \sum_{i=1}^{n} w_i f(y_i)$$

where

$$y_i = \left(\frac{b-a}{2}\right)x_i + \left(\frac{b+a}{2}\right)$$

x_i is the ith root of the Chebyshev polynomial of the second kind and

$$w_i = \frac{\pi}{n+1}\sin^2\left(\frac{i}{n+1}\right)\pi$$

for i=1, ... , n.

REFERENCE

Abramowitz, M. and Stegun, I. (eds.), *Handbook of Mathematical Functions*, Dover, New York, 1972, p. 889

■ gauss

FUNCTION

y = gauss(integFct, a, b, n)

PURPOSE

Compute the definite integral of a real-valued smooth function f(x) over the interval [a, b] using Gauss formulas of orders 2-10, 12 or 16

INPUT

integFct (Function): the integrand function f(x)

a (Real Scalar): the lower limit of the definite integral

b (Real Scalar): the upper limit of the definite integral

n (Real Scalar): the order of the Gaussian formula used to compute the definite integral

OUTPUT

y (Real Scalar): the computed definite integral of f(x) over the interval [a,b]

EXAMPLES

```
//  Example for: gauss(integFct, a, b, n)

//  integFct is the user specified function, the lower limit
//   a = 0, the upper limit b = 1, and the Gaussian order n = 10:

a = 0;
b = 1;
n = 10;

function integFct(x)
  return sin(x)+cos(x);
end function;

y = gauss(integFct, a, b, n);

//  Result :
//          y:           1.30116867893976
```

ALGORITHM AND COMMENTS

The Gauss formula for computing the definite integral of a function f(x) over [a,b] is given by:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i F(y_i)$$

where

$$y_i = \left(\frac{b-a}{2}\right)x_i + \left(\frac{b+a}{2}\right)$$

for i=1, …, n and x_i is the ith root of the Legendre polynomial of degree n-1,

$$w_i = \frac{2}{(1-x_i)}\left[P'_n(x_i)\right]^2$$

Theoretically, if f(x) is an 2n-times continuously differentiable function over [a, b], the upper bound for the truncation error of the nth order Gauss formula is:

$$\frac{(b-a)^{2n+1}(n!)^4}{(2n+1)\left[(2n)!\right]^3}\left\|f^{(2n)}\right\|$$

where

$$\left\|f^{(2n)}\right\| = \max_{x \in (a,b)}\left|f^{(2n)}(x)\right|$$

REFERENCE

Abramowitz, M. and Stegun, I. (Eds.), *Handbook of Mathematical Functions*, Dover, 1972, pp. 887-888

■ herIntegral

FUNCTION

y = herIntegral(integFct, n)

PURPOSE

Compute the integral of a real-valued smooth function f(x) over (-∞, ∞) with respect to the weight function exp($-x^2$) using the n-point Gaussian formula for Hermite integration

INPUT

integFct (Function): the given smooth function f(x)

n (Integer Scalar): the number of points used in applying the Gaussian formula for Hermite integration, where n is an integer satisfying $2 \le n \le 10$ or n = 12, 16, 20

OUTPUT

y (Real Scalar): the computed integral of exp($-x^2$) over (-∞, ∞)

EXAMPLES

```
//  Example for: herIntegral(integFct, n)

//  integFct is the user specified function, and the number
//  of points n = 10:

n = 10;

function integFct(x)
  return sin(x)+cos(x);
end function;

y = herIntegral(integFct,n);

//  Result :
//         y:           1.38038844704308
```

ALGORITHM AND COMMENTS

The Gaussian formula for computing the Hermite integral of a smooth function f(x) is given by:

$$\int_{\infty}^{-\infty} e^{-x^2} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

where the abscissas, x_i and the weight factors, w_i can be found, for example in Table 25.10 of the reference given below.

REFERENCE

Abramowitz, M. and Stegun, I. (eds.), *Handbook of Mathematical Functions*, Dover, New York, 1972, pp. 890, 924

■ integParab

FUNCTION

s = integParab(xVector, yVector, a, b)

PURPOSE

Compute the integral of a real tabulated function y(x) over the interval [a, b] using the rule of overlapping parabolas

INPUT

xVector (Real Vector): abscissas x_i, $1 \le i \le n$, where function y(x) is tabulated (inequally spaced, n has to be ≥

3). It is required that $x_{i+1} > x_i$.

yVector (Real Vector): ordinates $y_i = y(x_i)$, $1 \le i \le n$, values of real-valued function y(x)

a (Real Scalar): the lower limit of integration

b (Real Scalar): the upper limit of integration

OUTPUT

s (Real Matrix): the computed definite integral

EXAMPLE

```
project   a, b, error, s;

local      x, xVector, yVector;

n = 17;
a = 0.0;
b = 1.0;

for i = 1 to n do
    x = (i - 1)/(n - 1);
    xVector[i]  = x^1.5;
    yVector[i] = 1/(1. + xvector[i]^2);
end for;

s = integParab(xVector, yVector, a, b);
error = arcTan(b) - arcTan(a)  - s;
```

ALGORITHM AND COMMENTS

Rule ([1], §2.3, p.60-61) of overlapping parabolas is used to compute the approximate value of the integral

$$s_i = \int_{x_i}^{x_{i+1}} y(x)\,dx = \frac{1}{2}\int_{x_i}^{x_{i+1}} (p_i(x) + p_{i+1}(x))\,dx$$

Here $p_i(x)$, $1 \le i < n$, is a quadratic polynomial that interpolates y(x) at three consecutive points:

$$p_i(x) = a_i x^2 + b_i x + c_i,\ p_i(x_j) = y_j,\ j = i-1, i, i+1$$

Evident modification is needed when i = 1, i = n-1 and only one parabola is available, and to incorporate the case $a \ne x_i$, $b \ne x_i$. If a = b the result s = 0 is returned. If a > b we interchange these two parameters and the return value will be -s. Let ia and ib be indices such that

$$x_{imin} \le a < x_{imin+1},\ x_{imax} < a \le x_{imax+1}$$

If $a < x_1$ then imin = 1, if $b > x_n$ then imax = n-1. Now the approximate value of integral is evaluated by

$$s = \sum_{imin}^{imax} s_i$$

REFERENCES

[1]. Philip J. Davis and Philip Rabinovitz, Methods of numerical integration, Second Edition, Academic Press, Inc., 1984

■ integSpline

FUNCTION

s = integSpline(xVector, yVector, a, b)

PURPOSE

Compute the integral of a real tabulated function y(x) over the interval [a, b] using the natural cubic spline interpolation of the input data.

INPUT

xVector (Real Vector): abscissas x_i, $1 \le i \le n$, where function y(x) is tabulated (inequally spaced, n has to be ≥ 3). It is required that $x_{i+1} > x_i$.

yVector (Real Vector): ordinates $y_i = y(x_i)$, $1 \le i \le n$, values of real-valued function y(x)

a (Real Scalar): the lower limit of integration

b (Real Scalar): the upper limit of integration

OUTPUT

s (Real Matrix): the computed definite integral

EXAMPLE

Let y(x) = sin(πx), $0 \le x \le 1$, n = 17, x_i = [(i-1)/(n-1)]^1.5:

```
project   a, b, error, s;

local      x, xVector, yVector;

n = 17;
a = 0.0;
b = 1.0;

for i = 1 to n do
```

```
    x = (i - 1)/(n - 1);
    xVector[i]  = x^1.5;
    yVector[i] = 1/(1. + xvector[i]^2);
end for;

s = integSpline(xVector, yVector, a, b);
error = arcTan(b) - arcTan(a)  - s;
```

ALGORITHM AND COMMENTS

Rule based on natural cubic spline interpolation (that minimizes the interpolant's curvature in the $L_2(x_1,x_2)$-norm) is used to compute the approximate value of the integral

$$s_i = \int_{x_i}^{x_{i+1}} y(x)\,dx = \int_{x_i}^{x_{i+1}} p_i(x)\,dx$$

Here $p_i(x)$ is a representation of the natural cubic spline p(x) over the ith interval, $1 \le i < n$. This means that p(x) is piecewise polynomial,

$$p(x) = p_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i,\ x_i < x < x_{i+1}$$

and that 1) $p(x_i) = y_i$, $1 \le i \le n$; 2) p is C^2-continuous; and 3) $p''(x_1) = p''(x_n) = 0$.

Evident modification is needed when $a \ne x_i$, $b \ne x_i$. If a = b the result s = 0 is returned. If a > b we interchange these two parameters and the return value will be -s. Let ia and ib be indices such that

$$x_{imin} \le a < x_{imin+1},\ x_{imax} < a \le x_{imax+1}$$

If $a < x_1$ then imin = 1, if $b > x_n$ then imax = n-1. Now the approximate value of integral is evaluated by

$$s = \sum_{imin}^{imax} s_i$$

REFERENCES

[1]. Philip J. Davis and Philip Rabinovitz, Methods of numerical integration, Second Edition, Academic Press, Inc., 1984

■ lagIntegral

FUNCTION

y = lagIntegral(integFct, n)

PURPOSE

Compute the integral of a real-valued smooth function f(x) over [0, ∞) with respect to the weight function e^{-x} using the n-point Gaussian formula for Laguerre integration

INPUT

integFct (Function): the given smooth function f(x)

n (Integer Scalar): the number of points used in applying the Gaussian formula for Laguerre integration, where n is an integer satisfying $2 \le n \le 10$ or n = 12, 15

OUTPUT

y (Real Scalar): the computed integral of $f(x)e^{-x}$ over (0, ∞)

EXAMPLES

```
//  Example for: lagIntegral(integFct, n)

//  integFct is the user specified function, and the number
//  of points n = 10:
n = 10;

function integFct(x)
  return cos(x);
end function;

y = lagIntegral(integFct,n);

//  Result :
//          y:          0.500000509799794
```

ALGORITHM AND COMMENTS

The Gaussian formula for computing the Laguerre integral of a given smooth function F(x) is given by:

$$\int_0^{\infty} e^{-x} f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

where the abscissas, x_i and the weight factors, w_i can be found, for example in Table 25.9 of the reference.

REFERENCE

Abramowitz, M. and Stegun, I. (eds.), *Handbook of Mathematical Functions*, Dover, New York, 1972, pp. 890, 923

■ logSing

FUNCTION

y = logSing(integFct, n)

PURPOSE

Compute the definite integral of a real-valued smooth function f(x) over the interval [0, 1] with respect to the singular weight function ln(x) using the n-point Gaussian integration formula for logarithmic singularities

INPUT

integFct (Function): the given smooth function f(x)

n (Integer Scalar): the number of points used in applying the Gaussian integration formula for logarithmic singularities, where n is an integer satisfying $2 \le n \le 4$

OUTPUT

y (Real Scalar): the computed definite integral of f(x)ln(x) over the interval (0, 1)

EXAMPLES

```
//  Example for: logSing(integFct, n)

//  integFct is the user specified function, and the number
//  of points n = 4:

n = 4;

function integFct(x)
  return cos(x);
end function;

y = logSing(integFct, n);

//  Result :
//           y:         -0.946082320887991
```

ALGORITHM AND COMMENTS

The Gaussian formula for computing the logarithmic singular integral of a given smooth function f(x) is given by:

$$\int_0^1 \ln(x)\, f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

where the abscissas, x_i and the weight factors, w_i can be found, for example in Table 25.7 of the reference.

REFERENCE

Abramowitz, M. and Stegun, I. (eds.), *Handbook of Mathematical Functions*, Dover, New York, 1972, pp. 890, 920

■ moment

FUNCTION

y = moment(integFct, k, n)

PURPOSE

Compute the kth moment of a real-valued smooth function f(x): $\int_0^1 x^k f(x)\, dx$, using the n-point Gaussian integration moment formula

INPUT

integFct (Function): the given smooth function f(x)

k (Integer Scalar): the order of the moment to be computed, where k is an integer satisfying $0 \le k \le 5$

n (Integer Scalar): the number of points to be used in applying the Gaussian integration formula for the moment, where n is an integer satisfying $1 \le n \le 8$

OUTPUT

y (Real Scalar): the computed kth moment of the smooth function f(x)

EXAMPLES

```
//  Example for:  moment(integFct, k, n)

//  integFct is the user specified function, the order of the
//  moment k = 5, and the number of points n = 8:

k = 5;
n = 8;

function integFct(x)
  return sin(x)*cos(2.1*x);
end function;

y = moment(integFct,k,n);
```

```
//  Result :
//       y = -0.0315572190885035
```

ALGORITHM AND COMMENTS

The Gaussian integration formula for computing the kth moment of a given function f(x) is given by:

$$\int_0^1 x^k f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

where the abscissas, x_i and the weight factors, w_i can be found in Table 25.8 of the reference given below.

REFERENCE

Abramowitz, M. and Stegun, I. (Eds.), *Handbook of Mathematical Functions*, Dover, New York, 1972, pp. 888, 921-922

■ simp

FUNCTION

y = simp(integFct, a, b, n)

PURPOSE

Compute the definite integral of a real-valued smooth function f(x) over the interval [a,b] using the extended (or composite) Simpson's rule

INPUT

integFct (Function): the integrand function f(x)

a (Real Scalar): the lower limit of the definite integral

b (Real Scalar): the upper limit of the definite integral

n (Integer Scalar): the (odd integer) number of equally spaced points over [a,b] that the extended Simpson's integration rule uses

OUTPUT

y (Real Scalar): the computed definite integral of f(x) over the interval [a,b]

EXAMPLES

```
//  Example for: simp(integFct, a, b, n)

//  integFct is the user specified function, the lower limit
//  a = 0, the upper limit b = 1, and the number of points
```

```
//  n = 11:

a = 0;
b = 1;
n = 11;

function integFct(x)
  return sin(x);
end function;

y = simp(integFct, a, b, n);

//  Result :
//      y:          0.459697949823821
```

ALGORITHM AND COMMENTS

The extended Simpson's rule for computing the definite integral of a given function f(x) over [a,b] using an odd number of points n is given by:

$$\int_a^b f(x)\,dx \approx h \sum_{i=1}^{\frac{n-1}{2}} \frac{[f(x_{2i-1} + 4f(x_{2i}) f(x_{2i+1}))]}{3}$$

where

$$h = \frac{b-a}{n-1}, x_k = a + (k-1)h, \text{for} k = 1, \ldots, n$$

Theoretically, if f(x) is a four times continuously differentiable function over [a, b], the upper bound for truncation error of the extemded Simpson's rule is:

$$\frac{n}{90} h^5 \left\| f^{(4)} \right\|$$

where

$$\left\| f^{(4)} \right\| = \max_{x \in (a,b)} \left| f^{(4)}(x) \right|$$

REFERENCE

Abramowitz, M. and Stegun, I. (Eds.), *Handbook of Mathematical Functions*, Dover, 1972, p. 886

■ trap

FUNCTION

y = trap(integFct, a, b, n)

PURPOSE

Compute the definite integral of a real-valued smooth function f(x) over the interval [a,b] using the modified trapezoidal rule

INPUT

integFct (Function): the integrand function f(x).

a (Real Scalar): the lower limit of the definite integral

b (Real Scalar): the upper limit of the definite integral

n (Integer Scalar): the number of equally spaced points over [a, b] that the modified trapezoidal integration rule uses

OUTPUT

y (Real Scalar): the computed definite integral of f(x) over the interval [a,b]

EXAMPLES

```
//  Example for: trap(integFct, a, b, n)

//  integFct is the user specified function, the lower limit
//  a = 0, the upper limit b = 1, and the number of points
//  n = 10:

a = 0;
b = 1;
n = 10;

function integFct(x)
  return sin(x);
end function;

y = trap(integFct, a, b, n);

//  Result :
//      y:          0.459696624263404
```

ALGORITHM AND COMMENTS

The modified trapezoidal rule for computing the definite integral of a function f(x) over [a,b] using n points is different from the standard trapezodial rule by adding the correction term. That is,

$$\int_a^b f(x)\,dx \approx h\sum_{i=0}^{n-1}\frac{[f(x_i) + f(x_{i+1})]}{2} + \frac{h}{24}[-f(x_{-1}) + f(x_1) + f(x_{n-1}) + f(x_{n+1})]$$

where

$$h = \frac{b-a}{n-1},\ x_i = a + (i-1)\,h,\ \text{for} i = -1, 0, 1, \ldots, n+1$$

Theoretically, if f(x) is a four times continuously differentiable function over [a-h, b+h], the upper bound for truncation error of the modified trapezodial rule is

$$\frac{11n}{720}h^5\left\|f^{(4)}\right\|$$

where

$$\left\|f^{(4)}\right\| = \max_{x\in(a-h,\,b+h)}\left|f^{(4)}(x)\right|$$

REFERENCE

Abramowitz, M. and Stegun, I. (Eds.), *Handbook of Mathematical Functions*, Dover, 1972, p. 885

CHAPTER 8
DERIVATIVE FORMULA FUNCTIONS

■ biharmonic

FUNCTION

z = biharmonic(Fct, x, y, order, h)

PURPOSE

Approximate the biharmonic transformation, $(\partial^4/\partial x^4 + 2\partial^4/\partial x^2\,\partial y^2 + \partial^4/\partial y^4)$, of a real-valued function of two variables f(x,y) using low order (2 or 4) finite difference formulas with step size h.

INPUT

Fct (Function): the function f(x,y) upon which the biharmonic operator operates

x (Real Scalar): the x coordinate of the point (x, y) where the biharmonic transformation is approximated

y (Real Scalar): the y coordinate of the point (x, y) where the biharmonic transformation is approximated

order (Integer Scalar): the integer order (2 or 4) of the finite difference formula used

h (Real Scalar): the step size used in the finite difference formula

OUTPUT

z (Real Scalar): the computed approximation of the biharmonic transformation at the point (x, y)

EXAMPLES

```
// Example for: biharmonic(Fct, x, y, order, h),
// the biharmonic transformation of f(x, y)

// Fct is the function f(x, y) to be operated on, coordinates
// evaluated at are x = y = 1, the order = 4, and
// step size h = 0.1:

x = 1;
y = 1;
order = 4;
h = 0.1;

function Fct(x,y)
  return -9.0*x + 7.2*x*y - 9.0*y^3 + 10.0*y - 5.0;
```

```
end function;

y = biharmonic(Fct, x, y, order, h);

//  Result :
//     y:       1.18539437525082e-13
```

ALGORITHM AND COMMENTS

Algorithm Description:

The orders of finite difference formulas used in the algorithm are:

1) Second order

$$[20f_{0,0} - 8(f_{1,0} + f_{0,1} + f_{-1,0} + f_{0,-1}) + 2(f_{1,1} + f_{1,-1} + f_{-1,1} + f_{-1,-1}) + (f_{0,2} + f_{2,0} + f_{-2,0} + f_{0,-2})] / h^4$$

2) Fourth order

$$[-(f_{0,3} + f_{0,-3} + f_{3,0} + f_{-3,0}) + 14(f_{2,0} + f_{0,2} + f_{-2,0} + f_{0,-2}) - 77(f_{0,1} + f_{0,-1} + f_{1,0} + f_{-1,0}) + 184f_{0,0} + 20(f_{1,1} + f_{1,-1} + f_{-1,1} + f_{-1,-1}) - (f_{1,2} + f_{2,1} + f_{-2,1} + f_{2,-1}) - (f_{-1,2} + f_{-2,1} + f_{-1,-2} + f_{-2,-1})] / 6h^4$$

where $f_{i,j} = f(x+ih, y+jh)$.

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathermatical Functions*, Dover, 1972, p. 885

■ derivative

FUNCTION

y = derivative(Fct, n, h, x)

PURPOSE

Approximate the first order derivative of a real-valued differentiable function f(x) using low order (≤ 4) finite difference formulas with step size h

INPUT

Fct (Function): the single variable function f(x) to be differentiated

n (Integer Scalar): the integer order (≥1 and ≤4) of the finite difference formula used

h (Real Scalar): the step size used in the finite difference formula

x (Real Scalar): the value where the derivative is approximated

OUTPUT

y (Real Scalar): the computed approximation of the derivative of f(x) at the point x

EXAMPLES

```
// Example for: derivative(Fct, n, h, x), the derivative
// of the function f(x)

// Fct is the function f(x) to be differentiated, ,
// the order n = 4, the step size is h = 0.1, and
// the evaluation point is x = 1:

n = 4;
h = 0.1;
x = 1;

function Fct(x)
  return -4.1*x^4 + 5.0*x^2  + sin(2.9*x) -240.0;
end function;

y = derivative(Fct, n, h, x);

//  Result :
//      y:          -9.21512144390063
```

ALGORITHM AND COMMENTS

The following different orders of finite difference formulas are used :

1) First order : [f(a+h) - f(a)] / h

2) Second order : [f(a+h) - f(a-h)] / 2h

3) Third order: -[2f(a-h) +3f(a) -6f(a+h) +f(a+2h)] / 6h

4) Fourth order: [f(a-2h) - 8f(a-h) +8f(a+h) -f(a+2h)] / 12h.

■ finiteDiffMat1

FUNCTION

a = finiteDiffMat1(n, ord, h);

PURPOSE

Compute a linear operator (matrix a) that is a finite difference approximation of d/dx of orders 1, 2 and 4. Operator a acts at the mesh functions y_i, $1 \le i \le n$. If such a function is a discretization of a smooth function $y_i = y(x_i)$, $x_{i+1}-x_i = h$ then a particular order approximation means that

$$\frac{dy(x_i)}{dx} = (ay)_i + O(h^2)$$

For order 1: forward and backward differences are provided by setting parameter ord equal to 1 and -1 respectively.

INPUT

n (Integer Scalar): number of mesh points (dimension of y)

ord (Integer Scalar): order of approximation (values 1, -1, 2, 4 are valid)

h (Real Scalar): the step of mesh net

OUTPUT

a (Real Matrix): the matrix of finite difference approximation to d/dx of order ord.

EXAMPLE

Let $y(x) = \sin(\pi x)$, $0 \le x \le 1$, $n = 17$, $h = 1/16$, $x_i = (i-1)h$. Vector dy will contain approximation to dy/dx of order ord, and in vector err we will store the difference between dy and exact derivative:

```
project   a, h, n, ord, y, dy, err;

local x;

n = 17;
ord = 2;
h = 1./(n - 1);

for i = 1 to n do
    x = (i - 1)/(n - 1);
    y[i] = sin(<pi>*x);
end for;
a = finiteDiffMatr(n, ord, h);
dy = a*y;

for i = 1 to n do
    x = (i - 1)/(n - 1);
    err[i] = | <pi>*cos(<pi>*x) - dy[i] |;
end for;
```

ALGORITHM AND COMMENTS

The following formulas are used for the matrix a:

If ord = 1 then a = (1/h) of

$$\begin{matrix} -1 & 1 & 0 & 0 & * \\ 0 & -1 & 1 & 0 & * \\ * & * & * & * & * \\ * & 0 & 1 & -3 & 2 \end{matrix}$$

If ord = -1 then a = (1/h) of

$$\begin{matrix} -2 & 3 & -1 & 0 & * \\ 0 & -1 & 1 & 0 & * \\ * & * & * & * & * \\ * & * & 0 & -1 & 1 \end{matrix}$$

If ord = -1 then a = (1/2h) of

$$\begin{matrix} -3 & 4 & -1 & 0 & * \\ -1 & 0 & -1 & 0 & * \\ * & * & * & * & * \\ * & * & -1 & 0 & -1 \\ * & * & 1 & -4 & 3 \end{matrix}$$

If ord = -1 then a = (1/24h) of

$$\begin{matrix} -50 & 96 & -72 & 32 & -6 & \quad i=1 \\ -6 & -20 & 36 & -12 & 2 & \quad i=2 \\ 2 & -16 & 0 & 16 & -2 & \quad 2 < i < n-1 \\ 2 & 12 & -36 & 20 & 6 & \quad i = n-1 \\ 6 & -32 & 72 & -96 & 50 & \quad i = n \end{matrix}$$

If ord = 2 we need $n \geq 4$, if ord = 4 we need $n \geq 6$.

REFERENCES

W.E. Schiesser, The Numerical Method of Lines, Academic Press, Inc., 1991.

■ finiteDiffMat2

FUNCTION

a = finiteDiffMat2(n, ord, h, bctype);

PURPOSE

Compute a linear operator (matrix a) that is a finite difference approximation of dx/dx_2 of orders 2 and 4. Operator a acts at the mesh functions y_i, $1 \le i \le n$. If such a function is a discretization of a usual function $y_i = y(x_i)$, $x_{i+1}-x_i = h$ then a particular order approximation means that

$$\frac{d^2}{dx^2} y(x_i) = (ay)_i + O(h^2)$$

When i =1, n some boundary conditions (BC) must be included that are specified through the parameter bctype. In case of Dirichlet BC the values y_1, y_n are utilized by the matrix a itself. In case of Newman BC the formula has to be modified for i =1, n

$$\frac{d^2}{dx^2} y(x_{1/n}) = (ay)_{1/n} \mp \frac{\gamma}{h} \frac{dy}{dx}(x_{1/n}) + O(h^2)$$

INPUT

n (Integer Scalar): number of mesh points (dimension of y)

ord (Integer Scalar): order of approximation (values 2, 4 are valid)

h (Real Scalar): the step of mesh net

bctype (Integer Scalar): type of boundary conditions; two HiQ language constants DIRICHLET_BC, NEWMAN_BC are available.

OUTPUT

a (Real Matrix): the matrix of finite difference approximation to d^2/dx^2 of order ord.

EXAMPLE

Let $y(x) = \sin(\pi x)$, $0 \le x \le 1$, n = 17, h = 1/16, $x_i = (i-1)h$. Vector dy will contain approximation to dy/dx of order ord, and the difference between dy and the exact derivative is stored in vector err:

```
project   a, h, n, ord, y, dy, err;

local x;

n = 17;
ord = 4;
h = 1./(n - 1);

for i = 1 to n do
    x = (i - 1)/(n - 1);
    y[i] = sin(<pi>*x);
end for;

a = finiteDiffMat2(n, ord, h);
dy = a*y;
```

```
for i = 1 to n do
    x = (i - 1)/(n - 1);
    err[i] = | -<pi>*<pi>*sin(<pi>*x) - dy[i] |;
end for;
```

ALGORITHM AND COMMENTS

The following formulas are used for the matrix a (and the constant γ):

If ord = 2 then for Dirichlet BC, a = $(1/h^2)$ of

$$\begin{array}{ccccccc} 2 & -5 & 4 & -1 & * & & i = 1 \\ * & 1 & -2 & 1 & * & & 1 < i < n \\ * & -1 & 4 & -5 & 2 & & i = n \end{array}$$

If ord = 2 then for Newman BC, $\gamma = 3$, and a = $(1/h^2)$ of

$$\begin{array}{ccccccc} -3.5 & 4 & -0.5 & 0 & * & & i = 1 \\ * & 1 & -2 & 1 & * & & 1 < i < n \\ * & 0 & -0.5 & 4 & -3.5 & & i = n \end{array}$$

If ord = 4 then for Dirichlet BC, a = $(1/24h^2)$ of

$$\begin{array}{cccccccc} 90 & -308 & 428 & -312 & 122 & -20 & * & i = 1 \\ 20 & -30 & -8 & 28 & -12 & 2 & * & i = 2 \\ * & -2 & 32 & -60 & 32 & -2 & * & 2 < i < n-1 \\ * & 2 & -12 & 28 & -8 & -30 & 20 & i = n-1 \\ * & -20 & 122 & -312 & 428 & -308 & 20 & i = n-1 \end{array}$$

If ord = 4 then for Newman BC, $\gamma = 100$, and a = $(1/24h^2)$ of

$$\begin{array}{cccccccc} -415/3 & 192 & -72 & 64/3 & -3 & * & * & i = 1 \\ 20 & -30 & -8 & 28 & -12 & 2 & * & i = 2 \\ * & -2 & 32 & -60 & 32 & -2 & * & 2 < i < n-1 \\ * & 2 & -12 & 28 & -8 & -30 & 20 & i = n-1 \\ * & * & -3 & 64/3 & -72 & 192 & -415/3 & i = n-1 \end{array}$$

If ord = 2 we need $n \geq 4$, if ord = 4 we need $n \geq 6$.

REFERENCES

W.E. Schiesser, The Numerical Method of Lines, Academic Press, Inc., 1991.

■ laplacian

FUNCTION

z = laplacian(Fct, x, y, order, h)

PURPOSE

Approximate the Laplacian in rectangular coordinates, ($\partial^2/\partial x^2 + \partial^2/\partial y^2$), of a real-valued function of two variables f(x,y) at a point (x, y) using low order (2 or 4) finite difference formulas with step size h

INPUT

Fct (Function): the function f(x,y) operated upon by the Laplacian

x (Real Scalar): the x coordinate of the point (x, y) where the Laplacian is approximated

y (Real Scalar): the y coordinate of the point (x, y) where the Laplacian is approximated

order (Integer Scalar): the integer order (2 or 4) of the finite difference formula used

h (Real Scalar): the step size used in the finite difference formula

OUTPUT

z (Real Scalar): the computed approximation of the Laplacian at (x, y)

EXAMPLES

```
// Example for: laplacian(Fct, x, y, order, h)

// Fct is the function f(x,y) operated upon, the evaluation
// point is x = 2, y = 1, the order = 4,
// and the step size h = -0.1:

x = 2;
y = 1;
order = 4;
h = -0.1;

function Fct(x,y)
  return -4.1*x^4 + 2.9*x + 5.0*x^2*y^2 +
          2.0*y^3 + 21.0*y^2 -26.0*y^4 -240.0;
end function;

z = laplacian (Fct, x, y, order, h);

//  Result :
//      y:            -404.8
```

ALGORITHM AND COMMENTS

The finite difference formulas used in the algorithm are:

1) Second order $(f_{1,0} + f_{0,1} + f_{-1,0} + f_{0,-1} - 4f_{0,0}) / h^2$;

2) Fourth order $[16(f_{1,0} + f_{0,1} + f_{-1,0} + f_{0,-1}) - (f_{2,0} + f_{0,2} + f_{-2,0} + f_{0,-2}) - 60f_{0,0}] / 12h^2$;

where

$f_{i,j} = f(x+ih, y+jh)$.

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathermatical Functions*, Dover, 1972, p. 885

■ numDerivative

FUNCTION

z = numDerivative(xVector, yVector, k, a)

PURPOSE

Compute the approximate kth-order derivative at the point a by numerically differentiating the Lagrange interpolation polynomial using a given set of n data points

INPUT

xVector (Real Vector): the first n-dimensional vector that contains the abscissas of the n data points used to construct the Lagrange interpolation polynomial

yVector (Real Vector): the second n-dimensional vector that contains the ordinate values of the n data points used to construct the Lagrange interpolation polynomial

k (Integer Scalar): the order of the derivative

a (Real Scalar): the abscissa at which the derivative of the data (x, y) is approximated

OUTPUT

z (Real Scalar): the approximated kth-order derivative at x = a

EXAMPLES

```
//  Example for: numDerivative(xVector, yVector, k, a)

//  The abscissas are:
xVector = { -2, -1, 0, 1, 2 };
//  The ordinates are:
yVector = { 81, 16, 1, 0, 1 };
//  The order of the derivative is k = 1, and the
//  evaluation abscissa is a = 0.6:
k = 1;
a = 0.6;

z = numDerivative(xVector, yVector, k, a);
```

```
//  Result :
//     z:        -0.256
```

ALGORITHM AND COMMENTS

The ith data point is simply (xVector [i], yVector [i]).

The kth-order derivative is computed by using the functions lagrinterp and polyDerivative together with applying Horner's rule to evaluate the polynomial.

■ partDerivative

FUNCTION

z = partDerivative(Fct, x, y, k, p, h)

PURPOSE

Approximate the first and second order partial derivatives $\partial^{k+p} / \partial x^k \partial y^p$ of a real-valued function f(x,y) at any point using second order finite difference formulas with step size h

INPUT

Fct (Function): the function f(x,y) whose partial derivatives are approximated

x (Real Scalar): the x coordinate of the point (x, y) where the partial derivatives are computed

y (Real Scalar): the y coordinate of the point (x, y) where the partial derivatives are computed

k (Integer Scalar): the order of the partial derivative with respect to the x direction

p (Integer Scalar): the order of the partial derivative with respect to the y direction

h (Real Scalar): the step size used in the finite difference formulas

OUTPUT

z (Real Scalar): the computed partial derivative at (x,y).

EXAMPLES

```
// Example for: partDerivative(Fct, x, y, p, q, h)

// The evaluation coordinates are x = 1, y = 2;
// the derivative orders are p = 1 (x direction),
// q = 1 (y direction); and the step //size h = 0.3:

x = 1;
y = 2;
p = 1;
q = 1;
h = 0.3;
```

```
function Fct(x, y)
  return -100.0 + 5.0*x + 9.0*y + x^2 + y^2 + -22.1*x*y;
end function;

z = partDerivative(Fct, x, y, p, q, h);

//  Result :
//     z:                  -22.1
```

ALGORITHM AND COMMENTS

The following finite difference formulas are used in the algorithm:

1) $\partial f_{0,0}/\partial x = (f_{1,1} - f_{-1,1} + f_{1,-1} - f_{-1,-1}) / 4h$;
   $\partial f_{0,0}/\partial y = (f_{1,1} - f_{1,-1} + f_{-1,1} - f_{-1,-1}) / 4h$;

2) $\partial^2 f_{0,0}/\partial x^2 = (f_{1,0} - 2 f_{0,0} + f_{-1,0}) / h^2$;
   $\partial^2 f_{0,0}/\partial y^2 = (f_{0,1} - 2 f_{0,0} + f_{0,-1}) / h^2$;

3) $\partial^2 f_{0,0}/\partial x \partial y = (f_{1,1} - f_{-1,1} - f_{1,-1} + f_{-1,-1}) / 4h^2$,

where

$f_{i,j} = f(x+ih, y+jh)$.

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover, 1972, p. 884

■ polyDerivative

FUNCTION

v = polyDerivative(u, k)

PURPOSE

Generate the (coefficients of the) polynomial resulting from the kth-order derivative of a given polynomial

INPUT

u (Real or Complex Vector): the n-dimensional vector containing the coefficients of the polynomial of degree n-1 :

$$P_n(x) = A_n x^{n-1} + A_{n-1} x^{n-2} + ... + A_1$$

k (Integer Scalar): the order of the derivative

OUTPUT

v (Real or Complex Vector): the (n-k)-dimensional vector containing the coefficients of the resultant polynomial $P^{(k)}_n(x)$, i.e., the kth order derivative of $P_n(x)$:

$$P^{(k)}_n(x) = B_{n-k}\, x^{n-k-1} + ... + B_1$$

EXAMPLES

```
//  Example for: polyDerivative(u, k)

//  The coefficient vector of the polynomial to be differentiated is:
u = { -5, 4, -3, 2, -1 };
//  The order of the derivative is k = 2:
k = 2;

v = polyDerivative(u,k);

//  Result : v:  3 rows
//                       -6
//                       12
//                      -12
```

ALGORITHM AND COMMENTS

In the case where the derivative order, k, is larger than or equal to the order of the polynomial, n, the function will return a one-dimensional vector containing the constant value of the desired derivative.

CHAPTER 9
SERIES FUNCTIONS

■ sComp

FUNCTION

w = sComp(u, v, c)

PURPOSE

Compute the coefficients of the power series that equals the composition of two given power series, i.e., given the vectors u and v containing the first n coefficients of the power series:

$$f(x) = \sum_{k=1}^{\infty} a_k x^k \qquad g(x) = \sum_{k=1}^{\infty} b_k x^k \qquad \text{where} a_1 = 1$$

compute the output vector of coefficients c_k, k = 1, ... , n, for the power series that equals the composition of g with f:

$$h(x) = g(f(x)) = \sum_{k=1}^{\infty} c_k x^k$$

INPUT

u (Real Vector): the vector of coefficients a_k, k = 1, ... , n, a_1 = 1

v (Real Vector): the vector of coefficients b_k, k = 1, ... , n

c (Integer Scalar): the flag indicating the composition type

OUTPUT

w (Real Vector): the vector of coefficients c_k, k = 1, ... , n

EXAMPLE

The composition of the series f(x) = exp(x) - 1 with the series g(y) = ln(1 + y) yields the series h(x) = g(f(x)) = x. Thus the first five coefficients of f(x), truncated to ten decimal places, are: a[1] = 1, a[2] = 0.5, a[3] = 0.1666666666, a[4] = 0.0416666666, and a[5] = 0.0083333333. The first five coefficients of g(y) are: b[1] = 1, b[2] = -0.5, b[3] = 0.3333333333, b[4] = -0.25, and b[5] = 0.2. The resulting coefficients for h(x) should be: c[1] = 1, c[2] = c[3] = c[4] = c[5] = 0.

```
//  Example for: sComp(u,v,c), the coefficients of the
```

```
//  series composition of the two series with coefficients
//  contained in u and v

//  The series exp(x) - 1 has coefficients u:
u = {1, 0.5, 0.1666666666, 0.0416666666, 0.0083333333};

//  The series ln(1 + y) has coefficients v:
v = {1, -0.5, 0.3333333333, -0.25, 0.2};

//  The composition type is (set default = 1):
comptype = 1;

w = sComp(u,v,comptype);

//  Result: w:  5 rows
//     1
//     0
//    -1.00000000006996e-10
//    -4.99999999357351e-11
//    -4.16666666582877e-11
```

ALGORITHM AND COMMENTS

The assumption that $a_1 = 1$ loses no generality.
The coefficients will be exactly computed (to precision) if the input coefficients are exact.

REFERENCE

Nijenhuis, A. and Wilf, H.S., *Combinatorial Algorithms*, 2nd. ed., Academic Press, 1978

■ sInv

FUNCTION

v = sInv(u)

PURPOSE

Compute the coefficients of the power series that equals the inverse of a given power series, i.e., given the vector u containing the first n coefficients of the power series:

$$f(x) = \sum_{k=1}^{\infty} a_k x^k \qquad \text{where} a_1 = 1$$

compute the output vector of coefficients b_k, k = 1, ... , n, $b_1 = 1$, for the power series that equals the inverse of f:

$$x = f^{-1}\left(\sum_{k=1}^{\infty} a_k x^k\right) = \sum_{k=1}^{\infty} b_k x^k \qquad \text{where} a_1 = 1, b_1 = 1$$

INPUT

u (Real Vector): the vector of coefficients a_k, k = 1, ... , n, $a_1 = 1$

OUTPUT

v (Real Vector): the vector of coefficients b_k, k = 1, ... , n, $b_1 = 1$

EXAMPLE

The inverse of the series for the function exp(x) - 1 is the series for the function ln(1 + x). The first four coefficients of exp(x) - 1, truncated to ten decimal places, are: a[1] = 1, a[2] = 0.5, a[3] = 0.1666666666, and a[4] = 0.0416666666. The computed coefficients of ln(1 + x) should be: b[1] = 1, b[2] = -0.5, b[3] = 0.3333333333, and b[4] = -0.25.

```
//  Example for: sInv(u), the coefficients of the
//  series inverse of the series with coefficients
//  contained in u

//  The series exp(x) - 1 has coefficients u:
u = {1, 0.5, 0.1666666666, 0.0416666666, 0.0083333333};

v = sInv(u);

//  Result: v:  5 rows
//             1
//            -0.5
//             0.3333333334
//            -0.2500000001
//             0.200000000116667
```

ALGORITHM AND COMMENTS

The assumption that $a_1 = 1$ loses no generality.
The coefficients will be exactly computed (to precision) if the input coefficients are exact.

REFERENCE

Nijenhuis, A. and Wilf, H.S., *Combinatorial Algorithms,* 2nd. ed., Academic Press, 1978

■ sPower

FUNCTION

v = sPower(u, p)

PURPOSE

Compute the pth power of a given power series, i.e., given a vector u containing the first n+1 coefficients a_k, k = 0, 1, ... , n, $a_0 = 1$, of the power series for u:

$$f(x) = \sum_{k=0}^{\infty} a_k x^k$$

compute the output vector of coefficients b_k, k = 0, 1, ... , n, $b_0 = 1$, for the power series:

$$g(x) = f(x)^p = 1 + \sum_{k=1}^{\infty} b_k x^k$$

INPUT

u (Real Vector): the vector of coefficients a_k, k = 0, 1, ... , n, where $a_0 = 1$

p (Integer Scalar): the integer power of the power series f(x)

OUTPUT

v (Real Vector): the vector of coefficients b_k, k = 0, 1, ... , n

EXAMPLE

The function exp(x) raised to the power ln(2) is the function 2^x, i.e., exp(x)^(ln(2)) = 2^x. The first five coefficients of the series for exp(x), truncated to ten decimal places, are: a[1] = 1, a[2] = 1, a[3] = 0.5, a[4] =0.1666666666, and a[5] = 0.0416666666. The value of ln(2) is p = 0.6931471806. The result should be: b[1] = 1, b[2] = 0.6931471806, b[3] = 0.240226507, b[4] = 0.055504108, and b[5] = 0.0096181291.

```
//  Example for: sPower(u,p), the coefficients of the
//  series resulting from raising the series with
//  coefficients contained in u to the pth power

//  The series exp(x)  has coefficients u:
u = {1, 1, 0.5, 0.1666666666, 0.0416666666};

//  The power p = ln(2):
p = 0.6931471806;
```

```
v2 = sPower(u,p);

//  Results:  v2:  5 rows
//        1
//        0.6931471806
//        0.240226506986865
//        0.055504108628234
//        0.00961812907782148
```

ALGORITHM AND COMMENTS

The assumption that $a_0 = 1$ loses no generality. The coefficients will be exactly computed (to precision) if the input coefficients are exact.

REFERENCE

Nijenhuis, A. and Wilf, H.S., *Combinatorial Algorithms,* 2nd. ed., Academic Press, 1978

■ sRatio

FUNCTION

w = sRatio(u, v)

PURPOSE

Compute the coefficients of the power series that equals the ratio of two given power series, i.e., given the vectors u and v containing the first n+1 coefficients of the power series:

$$f(x) = \sum_{k=0}^{\infty} a_k x^k \qquad g(x) = \sum_{k=0}^{\infty} b_k x^k \qquad \text{where} b_0 = 1$$

compute the output vector of coefficients c_k, k = 0, 1, ... , n, for the power series:

$$h(x) = \frac{f(x)}{g(x)} = \sum_{k=0}^{\infty} c_k x^k$$

INPUT

u (Real Vector): the vector of coefficients a_k, k = 0, 1, ... , n

v (Real Vector): the vector of coefficients b_k, k = 0, 1, ... , n, with $b_0 = 1$

OUTPUT

w (Real Vector): the vector of coefficients c_k, k = 0, 1, ... , n

EXAMPLE

The ratio of the series for the function exp(x) to the series for the function exp(-x) results in the series for the function exp(2x), since: exp(x)/exp(-x) = exp(2x). The first five coefficients of the series for exp(x), truncated to ten decimal places, are: a[1] = 1, a[2] = 1, a[3] = 0.5, a[4] =0.1666666666, and a[5] = 0.0416666666. The series for exp(-x) has coefficients b[1] = 1, b[2] = -1, b[3] = 0.5,
b[4] = -0.1666666666, and b[5] = 0.0416666666. The resulting coefficients should be: c[1] = 1, c[2] = 2, c[3] = 2, c[4] = 1.3333333333, and c[5] = 0.6666666667.

```
//  Example for: sRatio(u,v), the coefficients of the
//  series that equals the ratio of the series with
//  coefficients contained in u and v, respectively.

//  The series exp(x) has coefficients u:
u = {1; 1; 0.5; 0.1666666666; 0.0416666666};

//  The series exp(-x) has coefficients v:
v = {1; -1; 0.5; -0.1666666666; 0.0416666666};
w = sRatio(u,v);

//  Results:  w:  5 rows
//           1
//           2
//           2
//           1.3333333332
//           0.6666666664
```

ALGORITHM AND COMMENTS

The assumption that $b_0 = 1$ loses no generality.
The coefficients will be exactly computed (to precision) if the input coefficients are exact.

REFERENCE

Nijenhuis, A. and Wilf, H.S., *Combinatorial Algorithms*, 2nd. ed., Academic Press, 1978

■ sRev

FUNCTION

w = sRev(u, v)

PURPOSE

Compute the coefficients of the power series for y in terms of x given z as a power series in x and z as a power

series in y, i.e., given the vectors u and v containing the first n coefficients of the power series:

$$z = f(x) = \sum_{k=1}^{\infty} a_k x^k \qquad z = g(y) = \sum_{k=1}^{\infty} c_k y^k \qquad \text{where} c_1 = 1$$

compute the output vector of coefficients b_k, k = 1, ... , n, for the power series that expresses y as a power series in x:

$$y = h(x) = \sum_{k=1}^{\infty} b_k x^k$$

This is known as the "reversion of series".

INPUT

u (Real Vector): the vector of coefficients a_k, k = 1, ... , n

v (Real Vector): the vector of coefficients c_k, k = 1, ... , n, c_1 = 1

OUTPUT

w (Real Vector): the vector of coefficients b_k, k = 1, ... , n

EXAMPLE

The series for y from the equation exp(-x) - 1 = exp(y) - 1 is the series for y = -x. The first five coefficients of exp(-x) - 1, truncated to ten decimal places, are:
a[1] = -1, a[2] = 0.5, a[3] = -0.1666666666, a[4] = 0.0416666666, and
a[5] = -0.0083333333. The coefficients of exp(y) - 1 are: c[1] = 1, c[2] = 0.5,
c[3] = 0.1666666666, c[4] = 0.0416666666, and c[5] = 0.0083333333. The result should be: b[1] =-1,
b[2] = b[3] = b[4] = b[5] = 0.

```
//  Example for: sRev(u,v), the coefficients of the
//  series reversion resulting from expressing the
//  series with coefficients contained in u in terms
//  of the series with coefficients contained in v

//  The series exp(-x) - 1 has coefficients u:
u = {-1; 0.5; -0.1666666666; 0.0416666666; -0.0083333333};

//  The series exp(y) - 1 has coefficients u:
v = {1; 0.5; 0.1666666666; 0.0416666666; 0.0083333333};
w = sRev(u,v);

//  Results:  w:  5 rows
//                        -1
//                         0
//                         0
```

```
//                          0
//                          0
```

ALGORITHM AND COMMENTS

The assumption that $c_1 = 1$ loses no generality.
The coefficients will be exactly computed (to precision) if the input coefficients are exact.

REFERENCE

Nijenhuis, A. and Wilf, H.S., *Combinatorial Algorithms*, 2nd. ed., Academic Press, 1978

CHAPTER 10
NUMERICAL FUNCTIONS

■ abs

FUNCTION

y = abs(x)

PURPOSE

Returns the absolute value of the complex scalar, vector, or matrix x, i.e., returns a scalar, vector, or matrix y, each element of which consists of the absolute value of the corresponding element of x

INPUT

x (Integer, Real, or Complex Scalar, Vector, or Matrix): the complex scalar, n-dimensional vector, or m by n matrix argument of abs(x)

OUTPUT

y (Integer or Real Scalar, Vector, or Matrix): the value of abs(x)

EXAMPLES

```
// Examples for: abs(x), the absolute value of x

// The real argument is x = -1.1:
x = -1.1;
y1 = abs(x);

// The complex argument is z = (1,-1):
z = (1,-1);
y2 = abs(z);

// Result :
// y1:  1.1
// y2:  1.4142135623731
```

ALGORITHM AND COMMENTS

Note: if x is of integer type, the answer will be of integer type, i.e., there is no loss of precision in this case.

■ arg

FUNCTION

y = arg(x)

PURPOSE

Compute the principal value of the argument of x

INPUT

x (Real or Complex Scalar, Vector, Matrix): the complex scalar, n-dimensional vector, or m by n matrix argument of arg(x)

OUTPUT

y (Real Scalar, Vector or Matrix): the principal value of the argument of x. If x is a vector or a matrix, the output is the n-dimensional vector or the m by n matrix containing the principal value of the argument of each component for x

EXAMPLES

```
// Examples for: arg(x), the principal value of x

// The real argument is x = -1.1:
x = -1.1;
y1 = arg(x);
// The complex argument is z = (1, -1):
z = (1,-1);
y2 = arg(z);

// Result :
// y1:  3.14159265358979
// y2: -0.785398163397448
```

ALGORITHM AND COMMENTS

Complex Domain: All non-zero complex numbers;
Real Range: $-\pi < y \leq \pi$

■ ceil

FUNCTION

y = ceil(x)

PURPOSE

Returns the nearest integer floating-point value not smaller than the real floating-point value of the complex scalar, vector, or matrix x, i.e., returns a scalar, vector, or matrix y, each element of which consists of the nearest integer floating-point value not smaller than the real floating-point value

INPUT

x (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix argument of ceil(x)

OUTPUT

y (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix value of ceil(x)

EXAMPLE

```
// An example for: ceil(x), the nearest integer floating point
// value not smaller than x

// The real argument is x = 1.95:
x =1.95;
y = ceil(x);

//  Result:
//         y:  2
```

ALGORITHM AND COMMENTS

This function was designed to behave as the corresponding C function does.

■ conj

FUNCTION

y = conj(x)

PURPOSE

Returns the complex conjugate of the complex scalar, vector, or matrix x, i.e., returns a scalar, vector, or matrix y, each element of which consists of the complex conjugate of the corresponding element of x

INPUT

x (Complex Scalar, Vector or Matrix): the argument of conj(x)

OUTPUT

y (Complex Scalar, Vector or Matrix): the scalar, vector or matrix given by conj(x)

EXAMPLES

```
// Examples for: conj(x), the complex conjugate of x
```

```
// The complex argument is x = (0, 1):
x = (0, 1);
y = conj(x);

//  Results:
// y:  0 + -1i

//  Compute conj(u) for a 5-dimensional complex vector u:

u = {(1,1); (-2,-2); (3,3); (-4,-4); (5,5)};
v =conj(u);

// Results:
//      v:  5 rows
//        1 - 1i
//       -2 + 2i
//        3 - 3i
//       -4 + 4i
//        5 - 5i

//  Compute conj(A) for a 3 by 2 complex matrix A:

A = {(1,1), (-2,-2);
       (3,3), (-4,-4);
       (5,5), (-6,-6)};

C =conj(A);

// Results:
//     C:  3 rows, 2 columns
//     1 - 1i          -2 + 2i
//     3 - 3i          -4 + 4i
//     5 - 5i          -6 + 6i
```

■ floor

FUNCTION

y = floor(x)

PURPOSE

Returns the nearest integer floating-point value not greater than the real floating-point value of the complex scalar, vector, or matrix x, i.e., returns a scalar, vector, or matrix y, each element of which consists of the nearest integer floating-point value not greater than the real floating-point value of the corresponding element of x

INPUT

x (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix argument of floor(x)

OUTPUT

y (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix value of floor(x)

EXAMPLE

```
// An example for: floor(x), the nearest integer floating
// point value not  greater than x

// The real argument is x = 1.95:
x =1.95;
y = floor(x);

// Result :
// y:  1
```

ALGORITHM AND COMMENTS

This function is designed to behave as the corresponding C function does.

■ fractPart

FUNCTION

y = fractPart(x)

PURPOSE

Returns the fractional part of the real floating-point value of the complex scalar, vector, or matrix x, i.e., returns a scalar, vector, or matrix y, each element of which consists of the fractional part of the real floating-point value of the corresponding element of x

INPUT

x (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix argument of fractPart(x)

OUTPUT

y (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix value of fractPart(x)

EXAMPLE

```
// An example for: fractPart(x), the fractional part of x

// The real argument is x = 1.95:
x =1.95;
y = fractPart(x);
```

```
// Result :
// y:  0.95
```

ALGORITHM AND COMMENTS

This function is designed to return the fractional part typically returned from the corresponding C function modf(x, *y).

■ gcd

FUNCTION

y = gcd(u)

PURPOSE

Returns the greatest common divisor of the array of integer values contained in the vector u

INPUT

u (Integer Vector): the argument of the function gcd(u)

OUTPUT

y (Integer Scalar): the gcd of the values u_1, u_2, ... , u_n, where n is the dimension of the vector u

EXAMPLE

```
// An example for: gcd(u), the greatest common divisor of the
// elements of u

// The argument vector is:
u = { 2; 1; 4; 3};
y = gcd(u);

// Result :
// y:  1
```

ALGORITHM AND COMMENTS

The algorithm used is based on a modified version of Euclid's algorithm.

■ gcd2

FUNCTION

y = gcd2(a, b)

PURPOSE

Returns the greatest common divisor of the pair of integer values a and b

INPUT

a (Integer Scalar): the first argument of the function gcd2(a,b)

b (Integer Scalar): the second argument of the function gcd2(a,b)

OUTPUT

y (Integer Scalar): the gcd of the values a and b

EXAMPLE

```
// An example for: gcd2(a,b), the greatest common divisor of
// a and b

// The first integer argument is a = 18; the second integer
// argument is b = 12:
a = 18;
b = 12;
y = gcd2(a,b);

// Result :
// y:  6
```

ALGORITHM AND COMMENTS

The algorithm used is based on a modified version of Euclid's algorithm, same as in gcd(u).

■ intPart

FUNCTION

y = intPart(x)

PURPOSE

Returns the integer portion of the real floating-point value of the complex scalar, vector, or matrix x, i.e., returns a scalar, vector, or matrix y, each element of which consists of the integer portion of the real floating-point value of the corresponding element of x

INPUT

x (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix argument of intPart(x)

OUTPUT

y (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix value of intPart(x)

EXAMPLE

```
// An example for: intPart(x), the integer part of x
// The real argument is x = 1.95:
x =1.95;
y = intPart(x);

// Result :
// y:  1
```

ALGORITHM AND COMMENTS

This function is designed to return the integer part typically returned from the corresponding C function modf(x, *y).

■ lcm

FUNCTION

y = lcm(u)

PURPOSE

Returns the least common multiple of the array of integer values contained in the integer vector u

INPUT

u (Integer Vector): the argument vector of the function lcm(u)

OUTPUT

y (Integer Scalar): The lcm of the values u_1, u_2, ... , u_n, contained in the n dimensional vector u

EXAMPLE

```
// An example for: lcm(u), the least common multiple of the
// elements in u

// The argument vector is:
u = { 2; 1; 4; 3};
y = lcm(u);
// Result : y:  12
```

■ lcm2

FUNCTION

y = lcm2(a, b)

PURPOSE

Returns the least common multiple of the integer pair a, b

INPUT

a (Integer Scalar): the first argument of the function lcm2(a, b)

b (Integer Scalar): the second argument of the function lcm2(a, b)

OUTPUT

y (Integer Scalar): The lcm of the integer values a and b

EXAMPLE

```
// An example for: lcm2(a,b), the least common multiple of
// a and b

// The first integer argument is a = 2; the second integer
// argument
// argument is b = 3:
a = 2;
b = 3;
y = lcm2(a,b);

// Result :
// y:  6
```

■ max

FUNCTION

y = max(A)

PURPOSE

Returns the maximum value among the elements contained in the vector or matrix argument A

INPUT

A (Real Vector or Matrix): argument of max(A)

OUTPUT

y (Real Scalar): the largest element contained in A

EXAMPLE

```
// An example for: max(u), the maximum element value of u

// The argument vector is:
u = { -2; -1; 4; 3};
```

```
y = max(u);

// Result :
// y:  6
```

■ max2

FUNCTION

z = max2(x, y)

PURPOSE

Returns the maximum value of the two values x and y

INPUT

x (Integer or Real Scalar): the first argument of max2(x, y)

y (Integer or Real Scalar): the second argument of max2(x, y)

OUTPUT

z (Integer or Real Scalar): the larger of the two values x and y

EXAMPLE

```
// An example for: max2(x,y), the maximum value of x and y

// The first real argument is x = 1.95; the second real
// argument is
// y = 3.2:
x =1.95;
y = 3.2;
z = max2(x,y);

// Result :
// y:  4
```

ALGORITHM AND COMMENTS

If both arguments are integers, then an integer is returned.

■ min

FUNCTION

y = min(A)

PURPOSE

Returns the minimum value among the elements contained in the vector or matrix argument A

INPUT

A (Real Vector or Matrix): argument of min(A)

OUTPUT

y (Real Scalar): the minimum element contained in A

EXAMPLE

```
// An example for:  min(u), the minimum element value of u

// The argument vector is:
u = { -2; -1; 4; 3};
y = min(u);

// Result :
// y:  -2
```

■ min2

FUNCTION

z = min2(x, y)

PURPOSE

Returns the minimum value of the two integer or real values x and y

INPUT

x (Integer or Real Scalar): the first argument of min2(x, y)

y (Integer or Real Scalar): the second argument of min2(x, y)

OUTPUT

z (Integer or Real Scalar): the smaller of the two values x and y

EXAMPLE

```
// An example for:  min2(x,y), the minimum value of x and y

// The first real argument is x=1.95; the second real argument
// is y  = 3.2:
x =1.95;
y = 3.2;
z = min2(x,y);
```

```
// Result :
// z:  1.95
```

ALGORITHM AND COMMENTS

If both arguments are integers, then an integer is returned.

■ mod

FUNCTION

z = mod(x, y)

PURPOSE

Returns the real value which is the remainder of the ratio x/y, i.e., the number f with the same sign as x such that x = iy + f for some integer i, |f| < |y|

INPUT

x (Real Scalar): the first argument of the function mod(x, y)

y (Real Scalar): the second argument of the function mod(x, y)

OUTPUT

z (Real Scalar): the value of mod(x, y)

EXAMPLE

```
// An example for: mod(x,y), the modulus remainder of x/y

// The first real argument is x = -3; the second real argument
// is y = 2.1:
x = -3;
y = 2.1;
z = mod(x,y);

// Result :
// z:  -0.9
```

ALGORITHM AND COMMENTS

If y = 0, an error is returned.

This implementation is designed after the mod(x, y) function in C.

pow

FUNCTION

z = pow(x, y)

PURPOSE

Compute x to the power y, i.e., x^y

INPUT

x (Real or Complex Scalar, Vector or Matrix): the real or complex scalar, n-dimensional vector, or m by n matrix

y (Real or Complex Scalar): the power

OUTPUT

z (Real or Complex Scalar, Vector or Matrix): the computed x^y. If x is a vector or a matrix, z is the n-dimensional vector or the m by n matrix containing the value of each component of x^y

EXAMPLES

```
// Examples for: pow(x,y), x to the y power
// The first real argument is x = 0.0625; the second real
// argument is y = 0.25:
x = 0.0625;
y = 0.25;
z = pow(x,y);

// Result :
// z:  0.5

// The first complex argument is a = (7, -1); the second real
// argument is y = 0:
a = (7,-1);
y = 0;

// Result :
// z1:  1 + 0i
```

ALGORITHM AND COMMENTS

The complex power of a complex number is in general not single-valued; the function pow computes the result for input parameter x in the principal branch of the complex plane. That is, the polar representation for any $x \neq 0$ is taken as

$$x = |x|e^{i\theta}$$

with $-\pi < \theta \leq \pi$.

For x = 0, the only valid y value for pow is y > 0, in all other cases, an error message will be returned.

■ round

FUNCTION

y = round(x)

PURPOSE

Returns the real floating-point value rounded to the nearest integer floating-point value of the complex scalar, vector, or matrix x, i.e., returns a scalar, vector, or matrix y, each element of which consists of the real floating-point value rounded to the nearest integer floating-point value of the corresponding element of x

INPUT

x (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix argument round(x)

OUTPUT

y (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix value of round(x)

EXAMPLE

```
// An example for: round(x), the nearest integer floating
// point value to x

// The real argument is x = 1.95:
x = 1.95;
y = round(x);

// Result :
// y:  2
```

ALGORITHM AND COMMENTS

This function operates as the corresponding standard C function does.

■ sign

FUNCTION

y = sign(x)

PURPOSE

For a scalar argument, returns the value of 1 if the sign of the real floating-point argument is positive; -1, if the sign of the real floating-point argument is negative; for a vector or matrix, it returns these values for each of the elements

INPUT

x (Real Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix argument sign(x)

OUTPUT

y (Integer Scalar, Vector, or Matrix): the scalar, n-dimensional vector, or m by n matrix value sign(x)

EXAMPLE

```
// An example for: sign(x), the signum of x

// The real argument is x = 1.95:
x = 1.95;
y = sign(x);

// Result :
// y:  1
```

■ sqrt

FUNCTION

y = sqrt(x)

PURPOSE

Returns the square root of a real floating-point value

INPUT

x (Real or Complex Scalar): a real or complex floating-point value

OUTPUT

y (Real or Complex Scalar): the square root of x (a real or complex floating-point value)

EXAMPLES

```
// Examples for: sqrt(x), the square root of x

// The real argument is x = 2:
x = 2;
y = sqrt(x);

// Result :
// y:  1.4142135623731
```

```
// The complex argument is z = (1,-1):
z = (1,-1);
y1 = sqrt(z);

// Result :
// y1:  1.09868411346781 -0.455089860562227i
```

ALGORITHM AND COMMENTS

The real domain of this function is $x \geq 0$. To return a complex root, use the complex form of x, i.e., for the square root of -1, take the square root of the complex number (-1, 0).

CHAPTER 11
POLYNOMIAL FUNCTIONS

■ degreePoly

FUNCTION

n = degreePoly(u)

PURPOSE

Determine the degree of a polynomial

INPUT

u (Real or Complex Vector): the n-dimensional vector which stores the coefficients of the polynomial P(x), of degree at most n-1, in ascending order, i.e., $P(x) = u_1 + u_2x + ... + u_nx^{n-1}$

OUTPUT

n (Integer Scalar): the non-negative integer specifying the degree of P(x)

EXAMPLE

```
// An example for: degreePoly(u), the degree of polynomial u

// The coefficients vector of the polynomial for which the
// degree is determined:
u = {5; -4; 3; -2; 1};
n = degreePoly(u);

// Result :
// n:  4
```

ALGORITHM AND COMMENTS

The degree of P(x) is the non-zero monomial of the highest degree in P(x). Thus, if $u_n = ... = u_{k+1} = 0$, and $u_k \neq 0$ where $1 \leq k \leq n$, then the degree of P(x) is k-1.

If $u_n = u_{n-1} = ... = u_1 = 0$, an error message will be returned to indicate that the degree of P(x) is not defined.

REFERENCE

Burden, L. R. and Faires, J.D., *Numerical Analysis*, 3rd ed., Prindle, Weber and Schmidt, Boston, 1985, P. 66

■ derivativePoly

FUNCTION

f = derivativePoly(u, k, b)

PURPOSE

Evaluate the kth order derivative of a given polynomial at b

INPUT

u (Real or Complex Scalar): the n-dimensional vector containing the coefficients of the polynomial of degree at most n-1 :

$$P_n(x) = u_n x^{n-1} + u_{n-1} x^{n-2} +....+ u_1$$

k (Integer Scalar): the order of the derivative

b (Real or Complex Scalar): the point where the kth order derivative is evaluated

OUTPUT

f (Real or Complex Scalar): the computed kth order derivative of the input polynomial at b

EXAMPLE

```
// An example for: derivativePoly(u,k,b), the kth order
//  derivative of polynomial u at b

// The coefficients vector of the polynomial for  which the
// derivative is evaluated:
u = {5; -4; 3; -2; 1};
// The order of the derivative is k = 2:
k = 2;
// The point where the derivative is computed is b = 1.0:
b = 1.0;
f = derivativePoly(u,k,b);

// Result :
// f:  6
```

■ multPoly

FUNCTION

f = multPoly(u, v, b)

PURPOSE

Evaluate the product of two polynomials at b

INPUT

u (Real or Complex Vector): the m-dimensional vector which stores the coefficients of the first polynomial, P(x) of degree at most m-1, in the ascending order, i.e., $P(x) = u_1 + u_2x +... + u_mx^{m-1}$

v (Real or Complex Vector): the n-dimensional vector which stores the coefficients of the second polynomial, Q(x) of degree at most n-1, in the ascending order, i.e., $Q(x) = v_1 + v_2x +... + v_nx^{n-1}$

b (Real or Complex Scalar): the point where P(x)Q(x) is evaluated

OUTPUT

f (Real or Complex Scalar): the evaluated P(b)Q(b)

EXAMPLE

```
// An example for: multPoly(u,v,b), the product of polynomials
// u and v

// The coefficients vectors of the polynomials for which the
// product is evaluated:
u = {5; -4; 3; -2; 1};
v = {-1; 2; -1; 3; -1};
// The point where the product is evaluated is b = 1.0:
b = 1.0;
f = multPoly(u,v,b);

// Result :
// f:  6
```

SEE ALSO

poly(u, b)

ALGORITHM AND COMMENTS

The P(b) and Q(b) are evaluated by Horner's rule before the product is calculated

■ poly

FUNCTION

f = poly(u, b)

PURPOSE

Evaluate a polynomial at b

INPUT

u (Real or Complex Vector): the n-dimensional vector which stores the coefficients of the polynomial, P(x) of degree at most n-1, in the ascending order,

i.e., $P(x) = u_1 + u_2x + ... + u_nx^{n-1}$

b (Real or Complex Scalar): the point where P(x) is evaluated

OUTPUT

f (Real or Complex Scalar): the computed result of P(b)

EXAMPLE

```
// An example for: poly(u,b), the polynomial u evaluated at b

// The coefficients vector of the polynomial evaluated is:
u = {5; -4; 3; -2; 1};
// The point where the polynomial is evaluated is b = 1.0:
b = 1.0;
f = poly(u,b);

// Result :
// f:   3
```

ALGORITHM AND COMMENTS

The result, P(b), is computed using the well-known Horner's rule in the following manner

$P(b) = (...(u_nb+u_{n-1})b+u_{n-2})b+...)b+u_1$

REFERENCE

Burden, L. R. and Faires, J.D., *Numerical Analysis*, third edition, Prindle, Weber and Schmidt, Boston, 1985, P. 67

■ ratPoly

FUNCTION

f = ratPoly(u, v, b)

PURPOSE

Evaluate the rational polynomial at b

INPUT

u (Real or Complex Vector): the m-dimensional vector which stores the coefficients of the numerator of the rational polynomial, in the ascending order, i.e., $P(x) = u_1 + u_2x + ... + u_mx^{m-1}$

v (Real or Complex Vector): the n-dimensional vector which stores the coefficients of the denomiator of the rational polynomial, in the ascending order,
i.e., $Q(x) = v_1 + v_2x + ... + v_nx^{n-1}$

b (Real or Complex Scalar): the point where the rational polynomial P(x)/Q(x) is evaluated

OUTPUT

f (Real or Complex Scalar): the evaluated P(b)/Q(b)

EXAMPLE

```
// An example for: ratPoly(u,v,b), the rational polynomial u/v
// evaluated at b

// The coefficients vectors of the rational polynomial
// evaluated:
u = {5; -4; 3; -2; 1};
v = {-1; 2; -1; 3; -1};
// The point where the rational polynomial are evaluated is
// b = 1.0:
b= 1.0;
f = ratPoly(u,v,b);

// Result :
// f:  1.5
```

SEE ALSO

poly(u, b)

ALGORITHM AND COMMENTS

The P(b) and Q(b) are evaluated by Horner's rule before the quotient, P(b)/Q(b), is calculated.

If Q(b) = 0, an error message will be returned

CHAPTER 12
GEOMETRIC FUNCTIONS

■ angleLine

FUNCTION

alpha = angleLine(a1, b1, c1, a2, b2, c2)

PURPOSE

Compute the angle between the lines given by the equations $a_1x+b_1y+c_1 = 0$ and $a_2x+b_2y+c_2 = 0$

INPUT

a1, b1, c1 (Real Scalars): the coefficients of the first line equation, $a_1x+b_1y+c_1=0$

a2, b2, c2 (Real Scalars): the coefficients of the second line equation, $a_2x+b_2y+c_2=0$

OUTPUT

alpha (Real Scalar): the computed angle between the two lines

EXAMPLE

```
// An example for: angleLine(a1,b1,c1,a2,b2,c2), the angle
// between two lines a1x+b1y+c1 = 0 and a2x+b2y+c2 = 0

// The coefficients of the first line equation
// are a1 = 1, b1 = 0 and c1 = 1:
a1 = 1;
b1 = 0;
c1 = 1;
// The coefficients of the second line equation
// are a2 = -1, b2 = 1 and c2 = 0:
a2 = -1;
b2 = 1;
c2 = 0;

alpha = angleLine(a1,b1,c1,a2,b2,c2);

// Result :
// alpha:  0.785398163397448
```

ALGORITHM AND COMMENTS

The computation of angle between two the lines $a_1x+b_1y+c_1 = 0$ and $a_2x+b_2y+c_2 = 0$ has been considered in terms of their slopes as follows.

Case 1: if $b_1 \neq 0$ and $b_2 \neq 0$, set

$$m_1 = \frac{-a_1}{b_1} \qquad m_2 = \frac{-a_2}{b_2}$$

then apply the formula given in function "angleSlope" to compute the desired angle.

Case 2: if $b_1 = b_2 = 0$, and $a_1, a_2 \neq 0$, then it is obvious that the angle, alpha, is zero.

Case 3: if $b_1 = 0$, $a_1 \neq 0$ and $a_2, b_2 \neq 0$, set

$$m_1 = 0 \qquad m_2 = \frac{b_2}{a_2}$$

then apply the formula given in function "angleSlope" to compute the desired angle.

Case 4: if $b_2 = 0$, $a_2 \neq 0$ and $a_1, b_1 \neq 0$, set

$$m_1 = \frac{b_1}{a_1} \qquad m_2 = 0$$

then apply the formula given in function "angleSlope" to compute the desired angle.

■ angleSlope

FUNCTION

alpha = angleSlope(m1, m2)

PURPOSE

Compute the angle between two lines with slopes m_1 and m_2

INPUT

m1 (Real Scalar): the slope of the first line

m2 (Real Scalar): the slope of the second line

OUTPUT

alpha (Real Scalar): the computed angle between the given two lines

EXAMPLE

```
// An example for: angleSlope(m1,m2), the angle between two
// lines with slopes m1 and m2

// The slopes of the first and second lines are
// m1 = 5 and m2 = -0.2:
m1 = 5;
```

```
m2 = -0.2;

alpha = angleSlope(m1,m2);

// Result :
//            alpha:  1.5707963267949
```

ALGORITHM AND COMMENTS

The angle between the two lines with slopes m_1, m_2 is given by

$$\text{alpha} = \frac{\pi}{2} \qquad \text{if} m_1 = \frac{-1}{m_2}$$

$$\text{alpha} = \left| \tan^{-1} \frac{m_2 - m_1}{1 + m_2 m_1} \right| \qquad \text{otherwise}$$

■ area

FUNCTION

a = area(x1, y1, x2, y2, x3, y3)

PURPOSE

Compute the area of triangle with vertices at (x_1,y_1), (x_2,y_2), and (x_3,y_3)

INPUT

x1,y1 (Real Scalars): the coordinates of the first vertex

x2,y2 (Real Scalars): the coordinates of the second vertex

x3,y3 (Real Scalars): the coordinates of the third vertex

OUTPUT

a (Real Scalar): the computed area of the triangle

EXAMPLE

```
// An example for: area(x1,y1,x2,y2,x3,y3) , the area of the
// triangle with vertices (x1,y1), (x2,y2) and (x3,y3)

// The first vertex coordinates are x1 = 0, y1 = 3:
x1 = 0;
y1 = 3;
// The second vertex coordinates are
// x2 = -2, y2 = 0:
x2 = -2;
```

```
y2 = 0;
// The third vertex coordinates are x3 = 4, y3 = 0:
x3 = 4;
y3 = 0;
a = area(x1,y1,x2,y2,x3,y3);

// Result :
// a:  9
```

ALGORITHM AND COMMENTS

The area of the triangle with vertices (x_1,y_1), (x_2,y_2) and (x_3,y_3) is given by

$$\text{Area} = \frac{1}{2}\left|(y_1 - y_2)\,(x_2 - x_3) - (x_1 - x_2)\,(y_2 - y_3)\right|$$

■ conic

FUNCTION

t = conic(a, b, c, d, e, f)

PURPOSE

Determine the type of conic section for the general quadratic equation $ax^2+bxy+cy^2+dx+ey+f = 0$

INPUT

a, b, c, d, e, f (Real Scalars): the coefficients of the quadratic equation, $ax^2+bxy+cy^2+dx+ey+f = 0$

OUTPUT

t (Integer Scalar): a flag indicating the type of conic section, i.e.,type = 0 - not a conic section, type = 1 - an ellipse,type = 2 - a hyperbola,type = 3 - a parabola, type = 4 - a circle

EXAMPLE

```
// An example for: conic(a,b,c,d,e,f), the conic section
// given by the general quadratic equation
// ax^2+bxy+cy^2+dx+ey+f = 0

// The coefficients of the general quadratic equation are
// a = 1,b = 2, c = 3, d = 4, e = 5, f = -56:
a = 1;
b = 2;
c = 3;
d = 4;
e = 5;
f = -56;
```

```
t = conic(a,b,c,d,e,f);

// Result :
//  t:  1
```

ALGORITHM AND COMMENTS

The given quadratic equation $ax^2+bxy+cy^2+dx+ey+f = 0$ is first converted into the special quadratic equation

$$a'u^2+c'v^2+d'u+e'v+f = 0$$

where

$$u = x\cos\theta + y\sin\theta \qquad v = -x\sin\theta + y\cos\theta$$

$$a' = a\cos^2\theta + b\cos\theta\sin\theta + c\sin^2\theta$$

$$c' = a\sin^2\theta - b\cos\theta\sin\theta + c\cos^2\theta$$

$$d' = d\cos\theta + e\sin\theta \qquad e' = -d\sin\theta + e\cos\theta$$

with

$$\theta = \frac{1}{2}\tan^{-1}\left(\frac{b}{a-c}\right)$$

The determination of type of conic section for the original equation $ax^2+bxy+cy^2+dx+ey+f = 0$ has been considered in terms of the equivalent equation $a'u^2+c'v^2+d'u+e'v+f = 0$ as follows.

Case 1: if $a' \neq 0$, $c' \neq 0$, $a' \neq c'$, $r = (c'd'^2+a'e'^2)/(4a'c')-f \neq 0$
and $a'c' > 0$, $c'r' > 0$; then it is an ellipse (i.e., type =1).

Case 2: if $a' \neq 0$, $c' \neq 0$, $a' \neq c'$, $r = (c'd'^2+a'e'^2)/(4a'c')-f \neq 0$
and $a'c' < 0$, then it is a hyperbola (i.e., type = 2).

Case 3: if $a' = 0$, $c' \neq 0$, and $d' \neq 0$ (or, $c' \neq 0$, $a' = 0$, and $e' \neq 0$), then it is a parabola (i.e., type = 3).

Case 4: if $a' = c' \neq 0$, $a' \neq c'$, $r = (c'd'^2+a'e'^2)/(4a'c')-f \neq 0$
and $a'c' > 0$, $c'r' > 0$; then it is a circle (i.e., type = 4).

Case 5: if a', c', d', e' and f satisfy none of the above four cases ,then it is not a conic section (i.e., type = 0).

■ dist

FUNCTION

d = dist(x1, y1, x2, y2)

PURPOSE

Compute the distance between the points (x_1, y_1) and (x_2, y_2)

INPUT

x1, y1 (Real Scalars): the coordinates of the first point

x2, y2 (Real Scalars): the coordinates of the second point

OUTPUT

d (Real Scalar): the computed distance between (x_1,y_1) and (x_2,y_2).

EXAMPLE

```
// An example for: dist(x1,y1,x2,y2), the distance between
// points (x1,y1) and (x2,y2)

// The first point coordinates are x1 = -6, y1 = 3:
x1 = -6;
y1 = 3;
// The second point coordinates are x2 = -2, y2 = 0:
x2 = -2;
y2 = 0;
d = dist(x1,y1,x2,y2);

// Result :
// d:  5
```

ALGORITHM AND COMMENTS

The distance between (x1,y1) and (x2,y2) is given by

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

■ distPToLine

FUNCTION

d = distPToLine(x1, y1, a, b, c)

PURPOSE

Compute the distance from the point (x1,y1) to the line given by the equation ax+by+c = 0

INPUT

x1,y1 (Real Scalars): the coordinates of the point

a,b,c (Real Scalars): the coefficients of the line equation, ax+by+c=0

OUTPUT

d (Real Scalar): the computed distance from the point to the line

EXAMPLE

```
// An example for: distPToLine(x1,y1,a,b,c), the distance from
// point (x1,y1) to line ax+by+c = 0

// The coordinates of the point are x1 = 0, y1 = 3:
x1 = 0;
y1 = 3;
// The coefficients of the line are a = 1, b = -2, c = 1:
a = 1;
b = -2;
c = 1;
d = distPToLine(x1,y1,a,b,c);

// Result :
// d:  0.894427190999916
```

ALGORITHM AND COMMENTS

The distance from a point (x_1,y_1) to the line ax+by+c = 0, is given by

$$d = \frac{|ax_1 + by_1 + c|}{\sqrt{a^2 + b^2}}$$

■ radius

FUNCTION

r = radius(a, d, e, f)

PURPOSE

Compute the radius of the circle $ax^2+ay^2+dx+ey+f = 0$

INPUT

a, d, e, f (Real Scalars): the coefficients of the quadratic equation, $ax^2+ay^2+dx+ey+f = 0$

OUTPUT

r (Real Scalar): the radius of the circle.

EXAMPLE

```
// An example for: radius(a,d,e,f), the radius of the circle
// given by ax^2+ay^2+dx+ey+f = 0

// The coefficients of the circle are a = 1, d= 4, e = -2,
// f = -31:
a = 1;
d = 4;
e = -2;
f = -31;
r = radius(a,d,e,f);

// Result :
// r:  6
```

ALGORITHM AND COMMENTS

The radius of the circle is given by

$$r = \frac{\sqrt{d^2 + e^2 - 4af}}{2|a|}$$

provided that $a \neq 0$ and $d^2+e^2-4af > 0$.

■ slope

FUNCTION

m = slope(x1, y1, x2, y2)

PURPOSE

Compute the slope of the line passing through the points (x_1, y_1) and (x_2, y_2)

INPUT

x1, y1 (Real Scalars): the coordinates of the first point

x2, y2 (Real Scalars): the coordinates of the second point

OUTPUT

m (Real Scalar): the computed slope of the line passing through (x_1,y_1) and (x_2,y_2)

EXAMPLE

```
// An example for: slope(x1,y1,x2,y2), the slope of the line
// passing through points (x1,y1) and (x2,y2)

// The coordinates of the first point are x1 = -6, y1 = 3:
x1 = -6;
y1 = 3;
// The coordinates of the second point are x2 = -2, y2 = 0:
x2 = -2;
y2 = 0;
m = slope(x1,y1,x2,y2);

// Result :
// m:   -0.75
```

ALGORITHM AND COMMENTS

The slope of the line passing through (x1,y1) and (x2,y2) is given by

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

for $x_1 \neq x_2$.

CHAPTER 13
BASIC MATRIX FUNCTIONS

■ bkSv

FUNCTION

u = bkSv(A, iVector, bVector)

PURPOSE

Perform forward substitution to find y such that Ly = B and backward substitution to find x such that Ux = y to solve a linear system of equations in the form Ax = LUx = L(Ux) = B

INPUT

A (Real or Complex Matrix): a square matrix A in Crout's decomposed LU form

iVector (Integer Vector): a vector iVector of the largest pivot element row indices returned from the LU decomposition function, LUD(A)

bVector (Real or Complex Vector): a vector B of real right-hand side values

OUTPUT

u (Real or Complex Vector): a vector of real solution values.

EXAMPLES

```
//  Examples for: bkSv(A, iVector, bVector)

//  Compute bkSv(LUDA, pivotA, bA) for  a 3-dimensional problem
//  with real matrix LUDA:

LUDA = { 1,    -1.1,         -2.1;
        -1,     1.1,         -2.1;
        -1,      -1,         -0.9 };
pivotA = {3; 3; 3};
bA = {-5.94;  9.79;  -3.41};
u = bkSv(LUDA, pivotA, bA);

//  Results:
//  u:  3 rows
//      1.1
//     -2.2
//      3.3

//  Compute bkSv(LUDB, pivotB, bB) for a 2-dimensional problem
// with complex matrix LUDB:
```

```
LUDB ={ (1, 1),    (-1,  1);
       (0.5, 1.5),  (4, 3)};
pivotB = {1; 2};
bB = {(2, -4);  (-7, 3)};
u = bkSv(LUDB, pivotB, bB);

//  Results:
//  u:  2 rows
//            1 +  -1i
//          -2 +   2i
```

SEE ALSO

LUD(A); solve(A, B).

ALGORITHM AND COMMENTS

See the reference for algorithmic details

REFERENCES

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., 1988. *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, NY, pp. 44 - 45

■ colDim

FUNCTION

n = colDim(A)

PURPOSE

Return the column dimension of a given matrix

INPUT

A (Real or Complex Matrix): a given m by n matrix

OUTPUT

n (Integer Scalar): the column dimension of the matrix A

EXAMPLES

```
//  An example for: colDim(A)
//  Compute colDim(A) for a 5 by 6 real  matrix A:
A = {   1, -1, -1, -1, -1, 0;
       -1,  2,  0,  0,  0, -1;
       -1,  0,  3,  1,  1, -2;
       -1,  0,  1,  4,  2, -3;
       -1,  0,  1,  2,  5, -4};
```

```
n = colDim(A);

//  Result :
//              n:  6
```

■ cond1

FUNCTION

c = cond1(A)

PURPOSE

Compute the condition number of a m by m nonsingular matrix A using the L_1 norm

INPUT

A (Real or Complex Matrix): a m by m real or complex matrix

OUTPUT

c (Real Scalar): cond1(A), the L_1 condition number of A

EXAMPLES

```
//  Examples for: cond1(A)
//  Compute  cond1(A) for a 5 by 5 real  matrix A:

A = {  1, -1, -1, -1, -1;
       -1,  2,  0,  0,  0;
       -1,  0,  3,  1,  1;
       -1,  0,  1,  4,  2;
       -1,  0,  1,  2,  5 };

c = cond1(A);

//  Result :     c:        1539

C = {  (1.0,1),   (-1,1.0),   (-1,1),  (-1,1), (-1,1);
       (-1,2.0),     (2, 2),   (0, 2),  (0, 2), (0., 2);
       (-1, 3),    (0, 3.0),   (3, 3),  (1, 3),   (1,3);
       (-1, 4),      (0, 4),    (1, 4),  (4, 4),   (2,4);
       (-1, 5),       (0, 5),   (1,  5),  (2, 5),  (5, 5) };

c1 = cond1(C);

//  Result :      c1:           14.4101574770224
```

ALGORITHM AND COMMENTS

If the input matrix A is singular, the condition number does not exist and an error message is returned.

■ cond2

FUNCTION

c = cond2(A)

PURPOSE

Compute the condition number of a m by m nonsingular matrix A using the L_2 norm

INPUT

A (Real or Complex Matrix): a m by m real or complex matrix

OUTPUT

c (Real Scalar): cond2(A), the L_2 condition number of A

EXAMPLES

```
//  Examples for: cond2(A)
//  Compute cond2(A) for a 5 by 5 real matrix A:

A = {   1, -1, -1, -1, -1;
       -1,  2,  0,  0,  0;
       -1,  0,  3,  1,  1;
       -1,  0,  1,  4,  2;
       -1,  0,  1,  2,  5 };
c1 = cond2(A);

//  Result :
//             c1:           865.980122392855

//  Compute cond2(C) for a 5 by 5 complex matrix C:

C = {  (1, 1),   (-1,1),   (-1,1),  (-1,1), (-1,1);
      (-1,2),    (2, 2),   (0, 2),  (0, 2),  (0, 2);
       (-1, 3),   (0, 3),   (3, 3),  (1, 3),  (1, 3);
       (-1, 4),   (0, 4),   (1, 4),  (4, 4),  (2, 4);
       (-1, 5),   (0, 5),   (1, 5),  (2, 5),  (5, 5) };
c2 = cond2(C);
//  Result :
//             c1:           865.980122392855
```

ALGORITHM AND COMMENTS

If the input matrix A is singular, the condition number does not exist and an error message is returned.

■ condF

FUNCTION

c = condF(A)

PURPOSE

Compute the condition number of a m by m nonsingular matrix A using the Frobenius norm (also called the Schur or Euclidean norm)

INPUT

A (Real or Complex Matrix): a m by m real or complex matrix

OUTPUT

c (Real Scalar): condF(A), the L_F condition number of A

EXAMPLES

```
//  Examples for: condF(A)
//  Compute condF(A) for a 5 by 5 real  matrix A:

A = {   1, -1, -1, -1, -1;
       -1,  2,  0,  0,  0;
       -1,  0,  3,  1,  1;
       -1,  0,  1,  4,  2;
       -1,  0,  1,  2,  5 };
c = condF(A);

//  Result :     c:           1001.63616148779

//  Compute  condF(C) for a 5 by 5 complex matrix C:

C = { (1.0,1),   (-1,1.0),  (-1,1),  (-1,1), (-1,1);
       (-1,2.0),   (2, 2),  (0, 2),  (0, 2), (0, 2);
       (-1, 3),    (0, 3),  (3, 3),  (1, 3),  (1,3);
       (-1, 4),    (0, 4),  (1, 4),  (4, 4),  (2,4);
       (-1, 5),    (0, 5),  (1, 5),  (2, 5), (5, 5) };
c1 = condF(C);
//  Result :    c1:           15.4605537080441
```

ALGORITHM AND COMMENTS

If the input matrix A is singular, the condition number does not exist and an error message is returned.

■ condI

FUNCTION

c = condI(A)

PURPOSE

Compute the condition number of a m by m nonsingular matrix A using the infinity norm

INPUT

A (Real or Complex Matrix): a m by m real or complex matrix

OUTPUT

c (Real Scalar): condI(A), the L_∞ condition number of A

EXAMPLES

```
//  Examples for: condI(A)
//  Compute condI(A) for a 5 by 5 real (integer)  matrix A:

A = {   1, -1, -1, -1, -1;
       -1,  2,  0,  0,  0;
       -1,  0,  3,  1,  1;
       -1,  0,  1,  4,  2;
       -1,  0,  1,  2,  5 };
c = condI(A);

//  Result :
//              c:      1539

//  Compute condI(C) for a 5 by 5 complex matrix C:

C = {(1.0,1),    (-1,1.0),   (-1,1),  (-1,1), (-1,1);
       (-1, 2),     (2, 2),  (0, 2),  (0, 2),  (0,2);
       (-1, 3),     (0, 3),  (3, 3),  (1, 3),  (1,3);
       (-1, 4),     (0, 4),  (1, 4),  (4, 4),  (2,4);
       (-1, 5),     (0, 5),  (1, 5),  (2, 5),  (5,5) };
c1 = condI(C);
//  Result :     c1:           19.7748284911659
```

ALGORITHM AND COMMENTS

If the input matrix A is singular, the condition number does not exist and an error message is returned.

■ copy

FUNCTION

B = copy(A)

PURPOSE

Create a new matrix which contains the elements in a given matrix A

INPUT

A (Real or Complex Matrix) : a given m by n matrix

OUTPUT

B (Real or Complex Matrix): the m by n matrix generated by copy(A) that contains the same elements as A

EXAMPLES

```
//  Examples for: copy(A)
//  Compute  copy(u) for a 3-dimensional real vector u:

u = {1;   2;   3};
v = copy(u);

//  Results :
//  v:  3 rows
//                  1
//                  2
//                  3

//  Compute copy(w) for a 2-dimensional complex vector w:

w = {(1,1);  (2,2); (3,3); (4,4)};
z = copy(w);

//  Results :
//  z:  4 rows
//              1 + 1i
//              2 + 2i
//              3 + 3i
//              4 + 4i

//  Compute copy(B) for a 3 by 2 real matrix B:

B = {  1.1,   2.1;
       3.1,   4.2;
       5.2,   6.2};
C = copy(B);
```

```
//  Results :
//  C:  3 rows, 2 columns
//                      1.1     2.1
//                      3.1     4.2
//                      5.2     6.2

//  Compute  copy(C) for a 2  by 2 complex matrix C:

C = {(1,1), (2, 2);
     (3,3),  (4,4)};
D = copy(C);

//  Results :
//  D:  2 rows, 2 columns
//                    1 + 1i          2 + 2i
//                    3 + 3i          4 + 4i
```

ALGORITHM AND COMMENTS

The equivalent operation in HiQ-Script:

```
B = A;
```

also creates a new matrix B with the contents of A.

■ det

FUNCTION

y = det(A)

PURPOSE

Compute the determinant of a matrix A

INPUT

A (Real or Complex Matrix): a m by m matrix

OUTPUT

y (Real Scalar): the value of the determinant (which is zero, if the matrix is singular)

EXAMPLES

```
//  Examples for: det(A)
//  Compute det(A) for a 5 by 5 real  matrix A:

A = {   1, -1, -1, -1, -1;
       -1,  2,  0,  0,  0;
       -1,  0,  3,  1,  1;
       -1,  0,  1,  4,  2;
```

```
       -1,  0,  1,  2,  5 };
y = det(A);

//  Result :
//            y:       1

//  Compute det(C) for a 5 by 5 complex matrix C:

C = { (1.0,1),   (-1,1.0),   (-1,1),  (-1,1), (-1,1);
      (-1, 2),   (2, 2),   (0, 2),  (0, 2),  (0, 2);
      (-1, 3),   (0, 3),   (3, 3),  (1, 3),  (1, 3);
      (-1, 4),   (0, 4),   (1, 4),  (4, 4),  (2, 4);
      (-1, 5),   (0, 5),   (1, 5),  (2, 5),  (5, 5) };
y1 = det(C);

//  Result :
//            y1:  1 + 651i
```

SEE ALSO

LUD(A)

ALGORITHM AND COMMENTS

Calculates the determinant by decomposing the matrix into Crout's LU form and multiplying the product of the diagonal elements by the sign (±1) corresponding to the number of row interchanges performed in the LU decompostion function, LUD(A); see the reference.

Returns an error message if the input parameter is invalid; a singular matrix is not considered an error.

REFERENCES

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W.T., *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, 1988, pp. 46 - 47

■ diag

FUNCTION

u = diag(A)

PURPOSE

Generate a vector containing the principal diagonal of a given square matrix A

INPUT

A (Real or Complex Matrix): a given m by m matrix

OUTPUT

u (Real or Complex Vector): the m-dimensional vector containing the elements of the principal diagonal of the matrix A, i.e., $u_i = a_{ii}$ for $1 \le i \le m$

EXAMPLES

```
//  Examples for: diag(A)
//  Compute diag(A) for a 4 by 4 real matrix A:

A = {  1,   2,   3,    0;
       4,   5,   6,    0;
       7,   8,   9,    0;
       10, 11, 12,  0};
u =diag(A);

//  Results :
//  u:  4 rows
//                        1
//                        5
//                        9
//                        0

//  Compute diag(C) for a 2 by 2 complex matrix C:

C = {  (1,1), (2, 2), (3,3);
       (4,4), (5,5), (6,6);
       (7,7), (8,8), (9,9)};
u1= diag(C);

//  Results :
//  u1:  3 rows
//                  1 + 1i
//                  5 + 5i
//                  9 + 9i
```

■ elemDivide

FUNCTION

Z = elemDivide(X, Y)

PURPOSE

Perform component-by-component division of two real vectors or matrices

INPUT

X (Real Vector or Matrix): the first (or numerator) real vector or matrix

Y (Real Vector or Matrix): the second (or denominator) real vector or matrix

OUTPUT

Z (Real Vector or Matrix): the vector or matrix resulting from dividing the input X by the input Y component-by-component. That is, $Z_i = X_i/Y_i$ for the case of vectors and $Z_{ij} = X_{ij}/Y_{ij}$ for the case of matrices.

EXAMPLES

```
// Examples for :  elemDivide(X,Y)

// Compute W =  elemDivide(U,V) where U  and V are both
//            5-dimensional vectors
U  =  {1; 2; 3; 4; 5};
V  =  {-5;-4;-3; 2; -1};
W =  elemDivide(U,V);

//  Result:     W:  5 rows
//                  -0.2
//                  -0.5
//                  -1
//                  -2
//                  -5

// Compute C=  elemDivide(A,B) where A and B are both
//            3 by 3 matrices
A = { 1.5,    2.5,   3.5;
     -4.5,   -5.5,  -6.5;
      7.5,    8.5,   9.5};
B  = {-1,  -1,  -1;
       2,   2,   2;
      -3,  -3,  -3};
C  = elemDivide(A,B);
//  Result:        C:  3 rows, 3 columns
//    -1.5         -2.5        -3.5
//    -2.25        -2.75       -3.25
//    -2.5     -2.83333333333333    -3.16666666666667
```

ALGORITHM AND COMMENTS

The two input vectors or matrices, X and Y, must have the same dimension. The components of Y must be all nonzero.

■ elemMultiply

FUNCTION

Z = elemMultiply(X, Y)

PURPOSE

Perform component-by-component multiplication of two real vectors or matrices

INPUT

X (Real Vector or Matrix): the first real vector or matrix

Y (Real Vector or Matrix): the second real vector or matrix

OUTPUT

Z (Real Vector or Matrix): the vector or matrix resulting from multiplying the input X and Y component-by-component. That is, $Z_i = X_i * Y_i$ for vectors and $Z_{ij} = X_{ij} * Y_{ij}$ for matrices.

EXAMPLES

```
// Examples for :  elemMultiply(X,Y)

// Compute W =  elemMultiply(U,V) where U  and V are both
//            5-dimensional vectors
U  =  {1; 2; 3; 4; 5};
V  =  {-5;-4;-3;-2;-1};
W =  elemMultiply(U,V);

//  Result:W:  5 rows
//                  -5
//                  -8
//                  -9
//                  -8
//                  -5

// Compute C=  elemMultiply(A,B) where A and B are both
//           3 by 3 matrices
A = { 1.5,   2.5,   3.5;
     -4.5,  -5.5,  -6.5;
      7.5,   8.5,   9.5};
B = {-1,    -1,    -1;
      2,     2,     2;
     -3,    -3,    -3};
C  = elemMultiply(A,B);

//  Result:    C:  3 rows, 3 columns
//        -1.5      -2.5      -3.5
//        -9       -11        -13
//       -22.5    -25.5      -28.5
```

ALGORITHM AND COMMENTS

The two input vectors or matrices, X and Y, must have the same dimension.

■ getColumn

FUNCTION

v = getColumn(A, k)

PURPOSE

Generate a column vector containing the elements in the kth column of a given matrix A

INPUT

A (Real or Complex Matrix): a given m by n matrix

k (Integer Scalar): a given column index of the matrix A

OUTPUT

v (Real or Complex Vector): the m-dimensional column vector whose elements are the same as those in the kth row of A, i.e., $v_j = a_{jk}$, for $1 \le j \le m$

EXAMPLE

```
//  An example for: getColumn(A, k)
//  Compute getColumn(A, k) for a 5 by 3 real  matrix A
//  with k = 2:

A = {  1,   2,   3;
       4,   5,   6;
       7,   8,   9;
       10, 11, 12;
       13, 14, 15};
v =getColumn(A,2);
//  Results :
//  v:  5 rows
//                        2
//                        5
//                        8
//                        1
//                        4
```

■ getRow

FUNCTION

v = getRow(A, k)

PURPOSE

Generate a row vector containing the elements in the kth row of a given matrix A

INPUT

A (Real or Complex Matrix): a given m by n matrix

k (Integer Scalar): a given row index of the matrix A

OUTPUT

v (Real or Complex Vector): the n-dimensional row vector whose elements are the same as those in the kth row of A, i.e., $v_j = a_{kj}$, for $1 \le j \le n$

EXAMPLE

```
//  An example for: getRow(A, k)
//  Compute getRow(A, k) for a 5 by 3 real matrix A with k = 2:
//
A = {   1,   2,   3;
        4,   5,   6;
        7,   8,   9;
       10,  11,  12;
       13,  14,  15};
v =getRow(A,2);

//  Results :
//  v:  3 rows
//                      4
//                      5
//                      6
```

■ ident

FUNCTION

I = ident(n)

PURPOSE

Construct an identity matrix of dimension n x n

INPUT

n (Integer Scalar): an integer value greater than zero

OUTPUT

I (Real Matrix): a square identity matrix of dimension n x n

EXAMPLE

```
//  An example for: ident(n)
//  Compute ident(n) for n = 5:

I = ident(5);

//  Results :
//  I:  5 rows, 5 columns
//     1 0 0 0 0
//     0 1 0 0 0
//     0 0 1 0 0
//     0 0 0 1 0
//     0 0 0 0 1
```

ALGORITHM AND COMMENTS

Returns an error message if the input parameter is invalid

■ imagPart

FUNCTION

B = imagPart(A)

PURPOSE

Extract the imaginary part of a complex scalar, vector, or matrix

INPUT

A (Complex Scalar, Vector, or Matrix): a complex argument

OUTPUT

B (Real Scalar, Vector, or Matrix): the extracted imaginary part

EXAMPLES

```
//  Examples for: imagPart(A)
//  Compute imagPart(u) for 3-dimensional complex vector u:

u = { (1,2); (3,4); (5,6)};
v = imagPart(u);

//  Results:
//  v:  3 rows
//                      2
//                      4
//                      6
```

```
//  Compute imagPart(A) for a 2 by 2 complex  matrix A:

A = {  (1.0,2),  (3,4.0);
       (5, 6 ),   (7,8)};
B = imagPart(A);

//  Results :
//  B:  2 rows, 2 columns
//                        2          4
//                        6          8
```

ALGORITHM AND COMMENTS

Returns an error message if the input parameter is invalid. Equvalent to the "`.i`" operator. In other words, `imagPart(A)` is equivalent to `A.i`.

■ inv

FUNCTION

B = inv(A)

PURPOSE

Compute the inverse of a n by n real or complex matrix A

INPUT

A (Real or Complex Matrix): a n by n matrix A

OUTPUT

B (Real or Complex Matrix): the inverse matrix A^{-1} of the matrix A

EXAMPLES

```
//  Examples for: inv(A)
//  Compute inv(A) for a 3 by 3 real  matrix A:

A = { 1.0, -1.1, -2.1;
       -1.0,  2.2,  0.0;
       -1.0,  0.0,  3.3};
B = inv(A);

//  Results :
//  B:  3 rows, 3 columns
//   -7.33333333333333    -3.66666666666667   -4.66666666666667
//   -3.33333333333333    -1.21212121212121   -2.12121212121212
//   -2.22222222222222    -1.11111111111111   -1.11111111111111
```

```
//  Compute  inv(C) for a 2 by 2 complex matrix C:

C = {(1.0,1),   (-1,1.0);
       (-1,2.0),     (2, 2) };
D = inv(C);

//  Results :
//  D:  2 rows, 2 columns
//                     0.32 -0.24i     -0.12 -0.16i
//                    -0.26 -0.18i      0.16 -0.12i
```

SEE ALSO

LUD(A), bkSv(A, iVector, bVector)

ALGORITHM AND COMMENTS

Computes the inverse of a matrix by decomposing the matrix into Crout's LU form and performing forward and backward substitution column by column; see the reference for a description of the algorithm used.

Returns an error message if the matrix is singular or an input parameter is invalid.

REFERENCES

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W.T., *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, 1988 pp. 45 - 46

■ locateMax

FUNCTION

[y, rowIndex, colIndex] = locateMax(A)

PURPOSE

Finds the largest element of a vector or rectangular matrix

INPUT

A (Real Vector or Matrix): the rectangular matrix or vector argument of locateMax(A)

OUTPUT

y (Real Scalar): the largest real element value in A

rowIndex (Integer Scalar): the row index of the largest element in A

colIndex (Integer Scalar): the column index of the largest element in A

EXAMPLES

```
//  Examples for: locateMax(A)
//  Compute the locateMax(u) for a 4-dimensional real vector u:
```

```
u = {-1.0;    2.0;   -3.0;   4.0};
y = locateMax(u);

//  Result :     y:     4

//  Compute the locateMax(A) for a 2 by 3 real  matrix A:

A = { -1,  2, 3;
      -4,  -5, 6};
y1 = locateMax(A);

//  Result :     y1:     6
```

ALGORITHM AND COMMENTS

Uses a simple iterative algorithm to locate the largest value. No prior order is assumed.

Returns an error message if the input parameter is invalid.

■ locateMin

FUNCTION

[y, rowIndex, colIndex] = locateMin(A)

PURPOSE

Finds the smallest element of a vector or rectangular matrix

INPUT

A (Real Vector or Matrix): the rectangular matrix or vector argument of locateMin(A)

OUTPUT

y (Real Scalar): the smallest real element value in A

rowIndex (Integer Scalar): the row index of the smallest element in A

colIndex (Integer Scalar): the column index of the smallest element in A

EXAMPLES

```
//  Examples for: locateMin(A)
//  Compute locateMin(u) for a 4-dimensional real vector u:

u = {-1.0;    2.0;   -3.0;   4.0};
y = locateMin(u);

//  Result :
//             y:        -3
```

```
//  Compute the locateMin(A) for a 2 by 3 real  matrix A:

A = { -1,  2, 3;
      -4,   -5, 6};
y1 = locateMin(A);

//  Result :
//             y1:      -5
```

ALGORITHM AND COMMENTS

Uses a simple iterative algorithm to locate the smallest value. No prior order is assumed.

Returns an error message if the input parameter is invalid.

■ lTriag

FUNCTION

u = lTriag(A)

PURPOSE

Return the lower triangular part of the matrix A in a vector u

INPUT

A (Real or Complex Matrix): an m by m matrix

OUTPUT

u (Real or Complex Vector): the vector containing the lower triangular portion of the matrix A

EXAMPLES

```
//  Examples for: lTriag(A)
//  Compute lTriag(A) for a 3 by 3 real matrix A:

A = {  1, -1, -1;
       2,  6,   0 ;
       3,  7,  10};
u = lTriag(A);
//  Results:
//  u:  6 rows
//                          1
//                          2
//                          3
//                          6
//                          7
//                         10
```

```
//  Compute lTriag(C) for a 3 by 3 complex matrix C:

C = {  (1.0,1),    (-1,1.0),      (-1,1);
       (2,2.0),      (6, 6),      (0, 2);
        (3, 3),      (7, 7),   (10, 10)};
v = lTriag(C);

//  Results:
//  v:  6 rows
//                        1 +  1i
//                        2 +  2i
//                        3 +  3i
//                        6 +  6i
//                        7 +  7i
//                       10 + 10i
```

■ LUD

FUNCTION

[LUD, pivot] = LUD(A)

PURPOSE

Perform the LU factorization of a matrix A

INPUT

A (Real or Complex Matrix): a given square matrix A

OUTPUT

LUD (Real or Complex Matrix): A square matrix in Crout's LU form produced by the decomposition of a row-wise permutation of matrix A

pivot (Real Vector): A vector recording the row index of the largest pivot element in each column of the matrix A

EXAMPLES

```
//  Examples for LUD(A)
//  Compute the LUD(A) for a 3 by 3 real  matrix A:

A = {  -1.0,  2.2,  0.0;
       -1.0,  0.0,  3.3;
       1.0, -1.1, -2.1};
[LUD, pivot] = LUD(A);
//
```

```
//  Results:
//  LUD:  3 rows, 3 columns
//             1              -1.1             -2.1
//            -1               1.1             -2.1
//            -1                -1             -0.9
//
//  pivot:  3 rows
//     3
//     3
//     3

//  Compute LUD(C) for a 2 by 2 complex matrix C:
C = { (1,1),      (-1,1);
      (-1,2.0),  (2, 2) };
[LUD1, pivot1] = LUD(C);
//

//  Results:
//  LUD1:  2 rows, 2 columns
//         1 +     1i               -1 +   1i
//       0.5 +   1.5i                4 +   3i
//
//  pivot1:  2 rows
//     1
//     2
```

SEE ALSO

bkSv(LUD, iVector, bVector); solve(A, b).

ALGORITHM AND COMMENTS

Uses Crout's Method with Partial Pivoting; see reference.

The algorithm has been modified to employ explicit scaling.

The second output, the pivot vector, is primarily used as the second input to the forward/backward substitution function, bkSv(A, iVector, bVector), to solve a linear system of equations Ax = B.

Returns an error message if the matrix A is singular or if the input parameter is invalid.

REFERENCES

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., 1988. *Numerical Recipes in C: The Art of Scientific Computing,* Cambridge University Press, New York, NY, pp. 39 - 44

■ norm1

FUNCTION

y = norm1(A)

PURPOSE

Compute the L_1 norm of a m by m matrix A

INPUT

A (Real or Complex Matrix): a m by m real or complex matrix

OUTPUT

y (Real Scalar): The L_1 norm of A

EXAMPLES

```
//  Examples for: norm1(A)
//  Compute norm1(A) for a 5 by 5 real  matrix A:

A = {   1, -1, -1, -1, -1;
       -1,  2,  0,  0,  0;
       -1,  0,  3,  1,  1;
       -1,  0,  1,  4,  2;
       -1,  0,  1,  2,  5 };
y = norm1(A);

//  Result :
//            y:       9

//  Compute norm1(C) for a 5 by 5 complex matrix C:
C = {   (1.0,1),   (-1,1.0),  (-1,1), (-1,1),  (-1,1);
       (-1,2.0),     (2, 2),  (0, 2), (0, 2),  (0.,2);
        (-1, 3),   (0, 3.0),  (3, 3), (1, 3),   (1,3);
        (-1, 4),     (0, 4),  (1, 4), (4, 4),   (2,4);
        (-1, 5),     (0, 5),  (1, 5), (2, 5),  (5, 5) };
y1 = norm1(C);

//  Result :
//            y1:   18.1196949894065
```

■ norm2

FUNCTION

y = norm2(A)

PURPOSE

Compute the L_2 norm of a m by m matrix A

INPUT

A (Real or Complex Matrix): a m by m real or complex matrix

OUTPUT

y (Real Scalar): The L_2 norm of A

EXAMPLES

```
//  Examples for: norm2(A)
//  Compute norm2(A) for a 5 by 5 real  matrix A:

A = { 1, -1, -1, -1, -1;
       -1,  2,  0,  0,  0;
       -1,  0,  3,  1,  1;
       -1,  0,  1,  4,  2;
       -1,  0,  1,  2,  5 };
y = norm2(A);

//  Result :
//             y:           7.48749993070499

//  Compute norm2(C) for a 5 by 5 complex matrix C:

C = {(1.0,1),  (-1,1.0),  (-1,1),  (-1,1),  (-1,1);
   (-1, 2),    (2, 2),   (0, 2),  (0, 2),   (0, 2);
   (-1, 3),    (0, 3),   (3, 3),  (1, 3),   (1, 3);
   (-1, 4),    (0, 4),   (1, 4),  (4, 4),   (2, 4);
   (-1, 5),    (0, 5),   (1, 5),  (2, 5),   (5, 5) };
y1 = norm2(C);

//  Result :
//             y1:           18.0396461653369
```

■ normF

FUNCTION

y = normF(A)

PURPOSE

Compute the Frobenius norm (also called the Schur or Euclidean norm) of a m by m matrix A

INPUT

A (Real or Complex Matrix): a m by m real or complex matrix

OUTPUT

y (Real Scalar): the Frobenius norm of A

EXAMPLES

```
//  Examples for: normF(A)
//  Compute normF(A) for a 5 by 5 real  matrix A:

A = { 1, -1, -1, -1, -1;
       -1,  2,  0,  0,  0;
       -1,  0,  3,  1,  1;
       -1,  0,  1,  4,  2;
       -1,  0,  1,  2,  5 };
y = normF(A);

//  Result :
//             y:           8.66025403784439

//  Compute normF(C) for a 5 by 5 complex matrix C:

C = {(1.0,1),   (-1,1.0),   (-1,1),  (-1,1), (-1,1);
   (-1,2),     (2, 2),   (0, 2),  (0, 2),  (0, 2);
   (-1, 3),    (0, 3),   (3, 3),  (1, 3),  (1, 3);
   (-1, 4),    (0, 4),   (1, 4),  (4, 4),  (2, 4);
   (-1, 5),    (0, 5),   (1, 5),  (2, 5),  (5, 5) };
y1 = normF(C);

//  Result :
//             y1:           18.7082869338697
```

■ normI

FUNCTION

y = normI(A)

PURPOSE

Compute the infinity norm of a m by m real matrix A

INPUT

A (Real or Complex Matrix): a m by m real or complex matrix

OUTPUT

y (Real Scalar): the infinity norm of A

EXAMPLES

```
//  Examples for: normI(A)
```

```
//  Compute  normI(A) for a 5 by 5 real  matrix A:

A = { 1, -1, -1, -1, -1;
      -1,  2,  0,  0,  0;
      -1,  0,  3,  1,  1;
      -1,  0,  1,  4,  2;
      -1,  0,  1,  2,  5 };
y = normI(A);

//  Result :
//            y:           9

//  Compute normI(C) for a 5 by 5 complex matrix C:

C = {(1.0,1),   (-1,1.0),   (-1,1),  (-1,1), (-1,1);
   (-1,2),     (2, 2),   (0, 2),  (0, 2),  (0, 2);
   (-1, 3),    (0, 3),   (3, 3),  (1, 3),  (1, 3);
   (-1, 4),    (0, 4),   (1, 4),  (4, 4),  (2, 4);
   (-1, 5),    (0, 5),   (1, 5),  (2, 5),  (5, 5) };
y1 = normI(C);

//  Result :
//            y1:           27.6542716461855
```

■ prod

FUNCTION

y = prod(A)

PURPOSE

Computes the product of the elements in a matrix or vector

INPUT

A (Real Vector or Matrix): the rectangular matrix or vector argument of prod(A)

OUTPUT

y (Real Scalar): the computed value of prod(A)

EXAMPLES

```
//  Examples for: prod(A)
//  Compute  prod(u) for a 3-dimensional real vector u:

u = {1; 2; 3};
y = prod(u);
```

```
//  Result :
//              y:         6

//  Compute  prod(A) for a 2 by 3 real matrix A:

A = { 1,  2, 3;
      4,  5, 6};
y1 = prod(A);

//  Result :
//              y1:      720
```

ALGORITHM AND COMMENTS

Computes the product of element values contained in A using a simple iterative algorithm.

Returns an error message if the input parameter is invalid.

■ randM

FUNCTION

R = randM(m, n)

PURPOSE

Construct a rectangular matrix of dimensions m by n with a pseudo-random sequence of real element values between 0 and 1.

INPUT

m (Integer Scalar): the row dimension of the generated matrix

n (Integer Scalar): the column dimension of the generated matrix

OUTPUT

R (Real Matrix): an m by n rectangular matrix of real values ranging between 0 and 1

EXAMPLES

```
//  An example for: randM(m,n)
//  Compute randM(m,n) for m=5, n=2:

R= randM(5,2);

//  Results:
//  R: 5 rows, 2 columns
//          0.000061035389081        0.500856401864439
//          0.525854131905362        0.552013588137925
//          0.263692667009309        0.345625131856650
```

```
//          0.690029110526666        0.893396073952317
//          0.422904143342748        0.371950354892761
```

SEE ALSO

SRandM(m, n, seed); rand(n)

ALGORITHM AND COMMENTS

Generates identical element values for successive restarts of the application. Successive calls to the function during the same session will, in general, produce different pseudo-random element values.

■ rank

FUNCTION

y = rank(A)

PURPOSE

Determine the rank of a m by n rectangular matrix A, i.e., the maximal number of independent rows or columns

INPUT

A (Real Matrix): the rectangular matrix argument of rank(A)

OUTPUT

y (Real Scalar): the computed value of rank(A)

EXAMPLES

```
//  An example for: rank(A)
//  Compute rank(A) for a 5 by 4 real  matrix A:

A = {   1, -1, -1, -1;
       -1,  2,  0,  0;
       -1,  0,  3,  1;
       -1,  0,  1,  4;
       -1,  0,  1,  2 };
y = rank(A);

//  Result :
//             y:  4
```

ALGORITHM AND COMMENTS

There are a number of ways to compute the rank of a matrix; the best method to use handles rank deficiency in the presence of round-off error. A robust SVD algorithm that is sensitive to the numerical characteristics of the machine is used, i.e., we compute the singular values and use a tolerance condition using the infinity norm of A: $\|A\|_\infty$.

REFERENCES

Golub, G.H and Van Loan, C.F., *Matrix Computations*, 2nd ed., Johns Hopkins, 1989

■ realPart

FUNCTION

B = realPart(A)

PURPOSE

Extract the real part of a complex scalar, vector, or matrix

INPUT

A (Complex Scalar, Vector, or Matrix): a complex argument

OUTPUT

B (Real Scalar, Vector, or Matrix): the extracted real part

EXAMPLES

```
//  Examples for: realPart(A)
//  Compute realPart(u) for 3-dimensional complex vector u:

u = { (1,2); (3,4); (5,6)};
v = realPart(u);

//  Results:
//  v:  3 rows
//                  1
//                  3
//                  5

//  Compute realPart(A) for a 2 by 2 complex  matrix A:

A = { (1.0,2),  (3,4.0);
      (5, 6 ),   (7,8)};
B = realPart(A);

//  Results:
//  B:  2 rows, 2 columns
//                        1       3
//                        5       7
```

ALGORITHM AND COMMENTS

Returns an error message if the input parameter is invalid. Equivalent to the "`.r`" operator. In other words, `realPart(A)` is equivalent to `A.r`.

■ rowDim

FUNCTION

y = rowDim(A)

PURPOSE

Return the row dimension of a given matrix

INPUT

A (Real or Complex Matrix): a given m by n matrix

OUTPUT

y (Integer Scalar): the row dimension of the matrix A

EXAMPLES

```
//  Examples for: rowDim(A)
//  Compute rowDim(A) for a 5 by 6 real  matrix A:

A = { 1, -1, -1, -1, -1, 0;
      -1,  2,  0,  0,  0, -1;
      -1,  0,  3,  1,  1, -2;
      -1,  0,  1,  4,  2, -3;
      -1,  0,  1,  2,  5, -4};
y = rowDim(A);

//  Result :
//            y:  5

//  Compute rowDim(B) for a 6 by 5 real  matrix B:

B = { 1.0, -1.0, -1.0, -1.0, -1.0;
      -1.0,  2.0,  0.0,  0.0,  0.0;
      -1.0,  0.0,  3.0,  1.0,  1.0;
      -1.0,  0.0,  1.0,  4.0,  2.0;
      -1.0,  0.0,  1.0,  2.0,  5.0;
      -1.0,  0.0,  1.0,  2.0,  3.0 };
y1 = rowDim(B);

//  Result :
//            y1:  6
```

■ scalarAdd

FUNCTION

Y = scalarAdd(X, s)

PURPOSE

Add a constant scalar to each element of a real vector or matrix

INPUT

X (Real Vector or Matrix): any real vector or matrix

s (Real Scalar): the constant scalar to be added to each element of X

OUTPUT

Y (Real Vector or Matrix): the vector or matrix resulting from adding each element of the input X by the scalar s

EXAMPLES

```
// Examples for :  scalarAdd(X,s)

// Compute V =  scalarAdd(U,s) where U is a 5-dimensional
//              vector and  s = -3
U  =  {1; 2; 3; 4; 5};
s  =  -3;
V  =  scalarAdd(U,s);

//  Result:        V:  5 rows
//                  -2
//                  -1
//                   0
//                   1
//                   2

// Compute Z=  scalarAdd(W,t) where W is a 3 by 3 matrix and
//             t = -4.5

W  =  { 1.5,   2.5,   3.5;
       -4.5,  -5.5,  -6.5;
        7.5,   8.5,   9.5};
t =  -4.5;
Z  =  scalarAdd(W,t);

//  Result:          Z:  3 rows, 3 columns
//        -3     -2     -1
//        -9    -10    -11
//         3      4      5
```

■ solve

FUNCTION

xVector = solve(A, bVector)

PURPOSE

Solve a linear system of equations in the form Ax = b

INPUT

A (Real or Complex Matrix): the n by n square matrix A of the linear system Ax = b

bVector (Real or Complex Vector): the n-dimensional vector b of right-hand side values

OUTPUT

xVector (Real or Complex Vector): the n-dimensional solution vector

EXAMPLES

```
//  Examples for: solve(A,b)
//  Compute solve(A,u) for a 5 by 5 real  matrix A and
//  a 5-dimensional real vector u:
A = { 1, -1, -1, -1, -1;
      -1,  2,  0,  0,  0;
      -1,  0,  3,  1,  1;
      -1,  0,  1,  4,  2;
      -1,  0,  1,  2,  5 };
u = {-13; 3; 17; 28; 35};
xVector = solve(A,u);

//  Results:
//  xVector:  5 rows
//                        1
//                        2
//                        3
//                        4
//                        5

//  Compute solve(C,v) for a 5 by 5 complex matrix C and
//  a 5-dimensional complex vector v:

C = {(1.0,1),       (-1,1.0),   (-1,1),  (-1,1), (-1,1);
      (-1,2.0),    (2, 2),   (0, 2),  (0, 2), (0., 2);
      (-1, 3),   (0, 3.0),   (3, 3),  (1, 3),   (1,3);
      (-1, 4),     (0, 4),    (1, 4),  (4, 4),   (2,4);
      (-1, 5),     (0, 5),   (1,  5),  (2, 5),  (5, 5)};
v = {(-4, -4); (0,0); (5,5); (10,10); (14,14)};
xVector1 = solve(C,v);
```

```
//  Results:
//  xVector1:  5 rows
//               -2 - 2i
//               -1 - 1i
//             5.73053024282714e-20 +     -2.70962109012691e-19i
//                1 + 1i
//                2 + 2i
```

SEE ALSO

LUD(A); bkSv(LUD, iVector, bVector)

ALGORITHM AND COMMENTS

Computes the solution to a system of linear equations by decomposing the matrix into Crout's LU form and then performing forward and backward substitution; see the reference for a description of the algorithm.

Returns an error message if the matrix A is singular or an input parameter is invalid.

REFERENCE

Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T., 1988. *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, NY, pp. 39 - 45

■ sRandM

FUNCTION

R = sRandM(m, n, seed)

PURPOSE

Construct a rectangular matrix of dimension m by n with a pseudo-random sequence of real element values between 0 and 1. The pseudo-random sequence to be produced is determined by specifying a seed, or initial value, for the pseudo-random number generator.

INPUT

m (Integer Scalar): the row dimension of the generated matrix

n (Integer Scalar): the column dimension of the generated matrix

seed (Integer Scalar): an integer to specify the value of the seed for system's pseudo random number generator

OUTPUT

R (Real Matrix): an m by n rectangular matrix of real values ranging between 0 and 1

Example

```
//  An example for: sRandM(m,n,seed)
```

```
//  Compute sRandM(m,n) for m=5, n=2, seed = 100:

R = sRandM(5,2,100);

//  Results:
//  R: 5 rows, 2 columns
//          0.506244640331715        0.692336864536628
//          0.674755459651351        0.050445011118426
//          0.127575733233243        0.100700062466785
//          0.509974497370422        0.319468657718971
//          0.129759316798300        0.661589191993698
```

SEE ALSO

randM(m, n) and rand(n)

ALGORITHM AND COMMENTS

Generates identical element values for successive calls to the function with the same seed value.

■ subdiag

FUNCTION

u = subdiag(A, m_1, n_1, m_2, n_2)

PURPOSE

Generate a vector whose elements are the same as those contained in a specified subdiagonal of a given matrix

INPUT

A (Real or Complex Matrix): the given m by n matrix

m_1 (Integer Scalar): the row index of A corresponding to the first element of the vector to be generated from the subdiagonal of A

n_1 (Integer Scalar): the column index of A corresponding to the first element of the vector to be generated from the subdiagonal of A

m_2 (Integer Scalar): the row index of A corresponding to the last element of the vector to be generated from the subdiagonal of A

n_2 (Integer Scalar): the column index of A corresponding to the last element of the vector to be generated from the subdiagonal of A

OUTPUT

u (Real or Complex Vector): the generated (m2-m1+1)-dimensional vector containing the subdiagonal elements of A starting from $A_{m1,n1}$

EXAMPLE

```
//  An example for: subdiag(A, m1, n1, m2, n2)
//  Compute subdiag(A,i,j,m,n) for a 3 by 3 real
//     matrix A with i= 2,
//  j = 1, m = 3, n = 2:

A = {1.1,   2.1,   3.1;
     4.2,   5.2,   6.2;
     7.3,   8.3,   9.3};
u = subdiag(A,2,1,3,2);

//  Results:
//  u:  2 rows
//                      4.2
//                      8.3
```

■ submat

FUNCTION

B = submat(A, m_1, n_1, m_2, n_2)

PURPOSE

Generates a submatrix B of a matrix A consisting of the elements of A ranging from $a_{m1,n1}$ to $a_{m2,n2}$

INPUT

A (Real or Complex Matrix): the given m by n matrix

m_1 (Integer Scalar): the row index of A corresponding to the first row of the generated matrix B

n_1 (Integer Scalar): the column index of A corresponding to the first column of the generated matrix B

m_2 (Integer Scalar): the row index of A corresponding to the last row of the generated matrix B

n_2 (Integer Scalar): the column index of A corresponding to the last column of the generated matrix B

OUTPUT

B (Real Matrix): the generated (m2-m1+1) by (n2-n1+1) submatrix of A such that: $b_{ij} = a_{i+m1-1, j+n1-1}$, where $1 \le i \le m2-m1+1$ and $1 \le j \le n2-n1+1$

EXAMPLES

```
//  An example for: submat(A, m1, n1, m2, n2)
//  Compute submat(A,i,j,m,n) for a 3 by 3 real matrix A
//  with i= 2, j = 1, m = 3, n = 2:

A = {1.1,   2.1,   3.1;
     4.2,   5.2,   6.2;
```

```
    7.3,   8.3,   9.3};
B = submat(A,2,1,3,2);

//  Results:
//  B:  2 rows, 2 columns
//                      4.2        5.2
//                      7.3        8.3
```

■ sum

FUNCTION

y = sum(A)

PURPOSE

Compute the sum of the elements in a matrix or vector

INPUT

A (Real Vector or Matrix): the rectangular matrix or vector argument of sum(A)

OUTPUT

y (Real Scalar): the computed value of sum(A)

EXAMPLES

```
//  Examples for: sum(A)
//  Compute  sum(u) for 3-dimensional real vector u:

u = {1.0; 2.0; 3.0};
y = sum(u);

//  Result :
//             y:    6

//  Compute sum(w) for 3-dimensional complex vector w:

w = { (1,4); (2,4); (3,4)};
z = sum(W);

//  Result :
//             z:    6 +12i

//  Compute sum(A) for a 2 by 2 real  matrix A:

A = { 1,  3;
      5,  7};
```

```
y1 = sum(A);

//  Result :
//             y1:    16
```

ALGORITHM AND COMMENTS

Computes the sum of the element values in A using a straightforward iterative algorithm.

Returns an error message if the input parameter is invalid.

■ tran

FUNCTION

B = tran(A)

PURPOSE

Generate the transpose of a given matrix A : A^t

INPUT

A (Real or Complex Matrix): the m by n matrix argument of tran(A)

OUTPUT

B (Real or Complex Matrix): the n by m matrix which is the transpose of A

EXAMPLES

```
//  Examples for: tran(A)
//  Compute tran(A) for a 2 by 4  real matrix A:

A = {1,   2,   3,    0;
     4,   5,   6,    0};
B = tran(A);

//  Results:
//  B:  4 rows, 2 columns
//                          1    4
//                          2    5
//                          3    6
//                          0    0

//  Compute tran(C) for a 2  by 2 complex matrix C:

C = {(1,1), (2, 2);
      (3,3),  (4,4)};
Z = tran(C);
```

```
//  Results:
//  Z:  2 rows, 2 columns
//                    1 + 1i        3 + 3i
//                    2 + 2i        4 + 4i
```

ALGORITHM AND COMMENTS

The transpose matrix B of an m by n matrix A with elements A_{ij} is the n by m matrix with:

$$B_{ji} = A_{ij}$$

for $1 \le i \le m$, $1 \le j \le n$.

■ uTriag

FUNCTION

u = uTriag(A)

PURPOSE

Return the upper triangular part of the matrix A in a vector u

INPUT

A (Real or Complex Matrix): an m by m matrix

OUTPUT

u (Real or Complex Vector): the vector containing the upper triangular portion of the matrix A

EXAMPLES

```
//  Examples for: uTriag(A)
//  Compute uTriag(A) for a 3 by 3 real  matrix A:

A = { 1, -1, -1;
      2,  6,   0 ;
      3,  7,  10};
u = uTriag(A);

//  Results:
//  u:  6 rows
//                     1
//                    -1
//                     6
//                    -1
//                     0
//                    10

//  Compute uTriag(C) for a 3 by 3 complex matrix C:
```

```
C = {(1.0,1),    (-1,1.0),      (-1,1);
   (2,2.0),      (6, 6),     (0, 2);
   (3, 3),       (7, 7),     (10, 10)};
v = uTriag(C);

//  Results:
//  v:  6 rows
//                      1 + 1i
//                     -1 + 1i
//                      6 + 6i
//                     -1 + 1i
//                      0 + 2i
//                     10 + 10i
```

■ vNorm1

FUNCTION

y = vNorm1(v)

PURPOSE

Compute the L_1 norm of a n-dimensional vector v

INPUT

v (Real or Complex Vector): the n-dimensional vector argument

OUTPUT

y (Real Scalar): the computed L_1 norm of v

EXAMPLES

```
//  Examples for: vNorm1(v)
//  Compute vNorm1(u) for a 5-dimensional real vector:

u = {-1;2;-3;4;-5};
y = vNorm1(u);

// Result:
//       y:          15

//  Compute vNorm1(v) for a 5-dimensionalcomplex vector:
v = {(1,1); (-2,-2); (3,3); (-4,-4); (5,5)};
y1 = vNorm1(v);

// Result:
//        y1:         21.2132034355964
```

ALGORITHM AND COMMENTS

The L_1 norm of the n-dimensional vector $v = (v_1,...,v_n)^t$ is defined as

$$\|v\|_1 = \sum_{i=1}^{n} |v_i|$$

REFERENCE

Golub, G.H and Van Loan, C.F., *Matrix Computations*, 2nd ed., Johns Hopkins, 1989, p.53

■ vNorm2

FUNCTION

y = vNorm2(v)

PURPOSE

Compute the L_2 norm of a n-dimensional vector v

INPUT

v (Real or Complex Vector): the n-dimensional vector argument

OUTPUT

y (Real Scalar): the computed L_2 norm of v

EXAMPLES

```
//  Examples for: vNorm2(v)
//  Compute vNorm2(u) for a 5-dimensional real vector:

u = {-1;2;-3;4;-5};
y = vNorm2(u);

// Result:
//       y:           7.41619848709566

//  Compute vNorm2(v) for a 5-dimensionalcomplex vector:

v = {(1,1); (-2,-2); (3,3); (-4,-4); (5,5)};
y1 = vNorm2(v);

// Result:
//       y1:           10.4880884817015
```

ALGORITHM AND COMMENTS

The L_2 norm of the n-dimensional vector $v = (v_1,...,v_n)^t$ is defined as

$$\|v\|_2 = \sqrt{\sum_{i=1}^{n} |v_i|^2}$$

REFERENCE

Golub, G.H and Van Loan, C.F., *Matrix Computations,* 2nd ed., Johns Hopkins, 1989, p.53

■ vNormI

FUNCTION

y = vNormI(v)

PURPOSE

Compute the infinity norm of a n-dimensional vector v

INPUT

v (Real or Complex Vector): the n-dimensional vector argument

OUTPUT

y (Real Scalar): the computed infinity norm of v

EXAMPLES

```
//  Examples for: vNormI(v)
//  Compute vNormI(u) for a 5-dimensional real vector:

u = {-1;2;-3;4;-5};
y= vNormI(u);

// Result:
//      y:          5

//  Compute vNormI(v) for a 5-dimensionalcomplex vector:

v = {(1,1); (-2,-2); (3,3); (-4,-4); (5,5)};
y1= vNormI(v);

// Result:
//       y1:        7.07106781186548
```

ALGORITHM AND COMMENTS

The infinity (or L_∞) norm of the n-dimensional vector $v = (v_1,...,v_n)^t$ is defined as

$$\|v\|_\infty = \max_{1 \le i \le n} |v_i|$$

REFERENCE

Golub, G.H and Van Loan, C.F., *Matrix Computations*, 2nd. ed., Johns Hopkins, 1989, p.53

CHAPTER 14
SPECIAL MATRICES

■ bordered

FUNCTION

B = bordered(n)

PURPOSE

Generate the n by n Bordered matrix

INPUT

n (Integer Scalar): the row and column dimension of the Bordered matrix to be generated

OUTPUT

B (Real Matrix): the generated n by n Bordered matrix with elements:

$$b_{ij} = \begin{cases} 1 & \text{if } i = j \\ 2^{1-i} & \text{if } i = n \text{ or } j = n \text{ but } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

for $1 \leq i, j \leq n$.

EXAMPLE

```
// An example for: bordered(n), generate an n by n Bordered
// matrix

// The row and column dimension is n = 5:
n = 5;
B = bordered(n);

// Results:
// B:  5 rows, 5 columns
// 1         0          0          0          1
// 0         1          0          0          0.5
// 0         0          1          0          0.25
// 0         0          0          1          0.125
// 1         0.5        0.25       0.125      1
```

ALGORITHM AND COMMENTS

The n by n Bordered matrix has n-2 eigenvalues at 1.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, 2nd ed., Adam Hilger Ltd., 1990, p. 253 - 254

■ diagonal

FUNCTION

D = diagonal(n)

PURPOSE

Generate the n by n Diagonal matrix

INPUT

n (Integer Scalar): the row and column dimension of the Diagonal matrix to be generated

OUTPUT

D (Real Matrix): the generated n by n Diagonal matrix with elements:

$$d_{ij} = \begin{cases} i & \text{if } i=j \\ 0 & \text{otherwise} \end{cases}$$

for $1 \le i, j \le n$

EXAMPLE

```
// An example for: diagonal(n), generate an n by n Diagonal
// matrix
// The row and column dimension is n = 5:
n = 5;
D = diagonal(n);
// Results:
// D:  5 rows, 5 columns
// 1        0       0       0       0
// 0        2       0       0       0
// 0        0       3       0       0
// 0        0       0       4       0
// 0        0       0       0       5
```

ALGORITHM AND COMMENTS

The eigenvalues of the generated diagonal matrix are the (diagonal) integer values i, i = 1, ... , n.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, 2nd ed., Adam Hilger Ltd., 1990, p. 254

■ dingDong

FUNCTION

D = dingDong(n)

PURPOSE

Generate the n by n DingDong matrix

INPUT

n (Integer Scalar): the row and column dimension of the DingDong matrix to be generated

OUTPUT

D (Real Matrix): the generated n by n DingDong matrix with elements

$$d_{ij} = \frac{0.5}{n - i - j + 1.5}$$

for $1 \le i, j \le n$.

EXAMPLE

```
// An example for: dingDong(n), generate an n by n DingDong
// matrix

// The row and column dimension is n = 3:
n = 3;

D = dingDong(n);

// Results:
// D:  3 rows, 3 columns
// 0.2                0.333333333333333   1
// 0.333333333333333   1                  -1
// 1                  -1                  -0.333333333333333
```

ALGORITHM AND COMMENTS

For a DingDong matrix of any dimension, there are few trailing zeros in some of the elements, so it is always represented inexactly in the machine. However, it is known to be very stable under inversion by elimination methods. Its eigenvalues have the property of clustering near $\pm \pi/2$.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, 2nd ed., Adam Hilger Ltd., 1990, pp. 253

■ frank

FUNCTION

F = frank(n)

PURPOSE

Generate the n by n Frank matrix

INPUT

n (Integer Scalar): the row and column dimension of the Frank matrix to be generated

OUTPUT

F (Real Matrix): the generated n by n Frank matrix with elements

$$f_{ij} = \min(i, j)$$

for $1 \le i, j \le n$.

EXAMPLE

```
// An example for: frank(n), generate an n by n Frank matrix

// The row and column dimension is n = 5:
n = 5;
F = frank(n);
// Results:
// F:  5 rows, 5 columns
// 1     1      1     1     1
// 1     2      2     2     2
// 1     2      3     3     3
// 1     2      3     4     4
// 1     2      3     4     5
```

ALGORITHM AND COMMENTS

The Frank matrix is a reasonably well-conditioned matrix.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, 2nd ed., Adam Hilger Ltd., 1990, p. 253

■ gram

FUNCTION

[G, det] = gram(A)

PURPOSE

Generate the Gram matrix and its determinant for a given set of n m-dimensional vectors

INPUT

A (Real Matrix): an m by n matrix with each column containing an m-dimensional vector

OUTPUT

G (Real Matrix): the n by n Gram matrix of the given set of input vectors

det (Real Scalar): the determinant of the Gram matrix

EXAMPLES

```
// An example for: gram(A), generate a Gram matrix and its
// determinant from A

// The argument matrix A is:
A = {1,  2,  3;
     4, -5,  6;
     7,  8, -9};
[ G, det ] = gram(A);
// Results :
// G:  3 rows, 3 columns
//   66                38              -36
//   38                93              -96
//  -36               -96              126
//
// det:  125316
```

ALGORITHM AND COMMENTS

For n given m-dimensional vectors u_1 , ... , u_n , we may check if $\{u_i\}_{i=1}^{n}$ forms a linearly independent set by examining whether the determinant of the Gram matrix is nonzero.

The elements, g_{ij} , of the Gram matrix are defined as

$$g_{ij} = (u_i)^t u_j$$

for $1 \le i, j \le n$ where u_i is the ith column of the input matrix A.

REFERENCE

Courant, R. and Hilbert D., *Methods of Mathematical Physics*, Wiley - Interscience, John Wiley & Sons, 1953, p. 34

■ hankel

FUNCTION

H = hankel(v)

PURPOSE

Generate the n by n Hankel matrix H with the given (2n-1)-dimensional vector v

INPUT

v (Real Vector): the (2n-1)-dimensional vector containing the desired elements of any row (or column) of the Hankel matrix

OUTPUT

H (Real Matrix): the n by n Hankel matrix generated by using the entered (2n-1)-dimensional vector v such that

$$h_{ij} = v_{i+j-1}$$

for $1 \le i, j \le n$

EXAMPLE

```
// An example for: hankel(v), generate an n by n Hankel matrix
// given a (2n-1)-dimensional vector v

// The argument vector v is:
v = {-3; 2; -1; 0; 1; -2; 3 };
H = hankel(v);

// Results:
// H:  4 rows, 4 columns
// -3        2       -1        0
//  2       -1        0        1
// -1        0        1       -2
//  0        1       -2        3
```

■ hilbert

FUNCTION

H = hilbert(n)

PURPOSE

Generate the n by n Hilbert matrix

INPUT

n (Integer Scalar): the row and column dimension of the Hilbert matrix to be generated

OUTPUT

H (Real Matrix): the generated n by n Hilbert matrix with elements:

$$h_{ij} = \frac{1}{i + j - 1}$$

for $1 \le i, j \le n$

EXAMPLE

```
// An example for: hilbert(n), generate an n by n
// Hilbert matrix

// The row and column dimension is n = 4:
n = 4;
H = hilbert(n);

// Results:  (Edited to 6 decimal place precision)
// H:  4 rows, 4 columns
//     1         0.5        0.333333     0.25
//     0.5       0.333333   0.25         0.2
//     0.333333  0.25       0.2          0.166667
//     0.25      0.2        0.166667     0.142857
```

ALGORITHM AND COMMENTS

The n by n Hilbert matrix is well-known to be positive definite. However, the matrix is so ill-conditioned for $n \ge 5$ that most eigenvalue or linear system algorithms fail for some value of $n < 20$.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, 2nd ed., Adam Hilger Ltd., 1990, pp. 253

■ kahan

FUNCTION

K = kahan(n)

PURPOSE

Generate the n by n Kahan matrix

INPUT

n (Integer Scalar): the row and column dimension of the Kahan matrix to be generated

OUTPUT

K (Real Matrix): the generated n by n Kahan matrix with elements:

$k_{ii} = 1$

$k_{ij} = -1,\ j = i+1, \ldots, n$

$k_{ij} = 0,\ j = 1, \ldots, i - 1$

for $1 \le i, j \le n$, i.e., the elements above the diagonal are equal to -1, the diagonal elements are equal to 1, and the elements below the diagonal are equal to 0

EXAMPLE

```
// An example for: kahan(n), generate an n by n Kahan matrix

// The row and column dimension is n = 5:
n = 5;
K = kahan(n);

// Results:
// K:  5 rows, 5 columns
// 1       -1       -1       -1       -1
// 0        1       -1       -1       -1
// 0        0        1       -1       -1
// 0        0        0        1       -1
// 0        0        0        0        1
```

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, 2nd ed., Adam Hilger Ltd., 1990, p. 253

■ moler

FUNCTION

M = moler(n)

PURPOSE

Generate the n by n Moler matrix

INPUT

n (Integer Scalar): the row and column dimension of the Moler matrix to be generated

OUTPUT

M (Real Matrix): the generated n by n Moler matrix with elements

$$m_{ij} = \begin{cases} i & \text{if } i=j \\ \min(i, j) - 2 & \text{otherwise} \end{cases}$$

for $1 \le i, j \le n$

EXAMPLE

```
// An example for: moler(n), generate an n by n Moler matrix

// The row and column dimension is n = 5:
n = 5;
M = moler(n);

// Results:
// M:  5 rows, 5 columns
//  1      -1      -1       -1      -1
// -1       2       0        0       0
// -1       0       3        1       1
// -1       0       1        4       2
// -1       0       1        2       5
```

ALGORITHM AND COMMENTS

The n by n Moler matrix is well-known to be positive definite. However, it has one small eigenvalue and often upsets elimination methods for solving linear systems.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, 2nd ed., Adam Hilger Ltd., 1990, pp. 253

■ toeplitz

FUNCTION

T = toeplitz(v)

PURPOSE

Generate the n by n Toeplitz matrix with the given (2n-1)-dimensional vector v

INPUT

v (Real Vector): the (2n-1)-dimensional vector containing the desired elements of any row (or column) of the Toeplitz matrix

OUTPUT

T (Real Matrix): the n by n Toeplitz matrix generated by using the input (2n-1)-dimensional vector v such that:

$$t_{ij} = v_{n+i-j}$$

for $1 \le i, j \le n$

EXAMPLE

```
// An example for: toeplitz(v), generate an n by n Toeplitz
// matrix given a (2n-1)-dimensional vector v

// The argument vector v is:
v = {-3; 2; -1; 0; 1; -2; 3};
T = toeplitz(v);

// Results:
// T:  4 rows, 4 columns
//   0       -1        2       -3
//   1        0       -1        2
//  -2        1        0       -1
//   3       -2        1        0
```

REFERENCE

Press, W.H., Flannery, B.P, Teukolsky, S.A. and Vetterling, W.T., *Numerical Recipes in C*, Cambridge University Press, 1988, p. 54

■ vandermonde

FUNCTION

A = vandermonde(v)

PURPOSE

Generate the n by n Vandermonde matrix A with a given n-dimensional vector v

INPUT

v (Real Vector): the n-dimensional vector containing the elements of any row (or column) of the desired Vandermonde matrix

OUTPUT

A (Real Matrix): the n by n Vandermonde matrix generated by using the input n-dimensional vector v such that

$$a_{1j} = 1$$

$$(a_{ij} = v_j^{i-1}) \qquad \text{for } i = 2, \ldots, n$$

where $1 \leq j \leq n$

EXAMPLE

```
// An example for: vandermonde(v), generate an n by n
// Vandermonde matrix given an n-dimensional vector v
// The argument vector v is:
v = {1.1; -2.2; 3.3; -4.4};
A = vandermonde(v);

// Result :
// A:  4 rows, 4 columns
// 1          1           1           1
// 1.1       -2.2         3.3        -4.4
// 1.21       4.84       10.89       19.36
// 1.331    -10.648      35.937     -85.184
```

ALGORITHM AND COMMENTS

Some authors refer to the matrix A^t, which results from Lagrange polynomial interpolation, as a Vandermonde matrix.

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, pp. 181-182

■ wilkinsonMinus

FUNCTION

W = wilkinsonMinus(n)

PURPOSE

Generate the n by n Wilkinson W- matrix

INPUT

n (Integer Scalar): the row and column dimension of the Wilkinson W- matrix to be generated

OUTPUT

W (Real Matrix): the generated n by n Wilkinson W- with elements

$$w_{ij} = \begin{cases} [n/2] + 1 - i & \text{if } i = j \\ 1 & \text{if } i = j+1 \text{ or } j = i+1 \\ 0 & \text{otherwise} \end{cases}$$

for $1 \le i, j \le n$, where [x] denotes the largest integer less than or equal to x

EXAMPLE

```
// An example for: wilkinsonMinus(n), generate an n by n
// Wilkinson W- matrix

// The row and column dimension is n = 5:
n = 5;
W = wilkinsonMinus(n);

// Result :
// W:  5 rows, 5 columns
// 2        1        0        0        0
// 1        1        1        0        0
// 0        1        0        1        0
// 0        0        1       -1        1
// 0        0        0        1       -2
```

ALGORITHM AND COMMENTS

For odd order, the Wilkinson W- matrix has eigenvalues which are pairs of equal magnitude but opposite sign.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, 2nd ed., Adam Hilger Ltd., 1990, p. 254

■ wilkinsonPlus

FUNCTION

W = wilkinsonPlus(n)

PURPOSE

Generate the n by n Wilkinson W+ matrix

INPUT

n (Integer Scalar): the row and column dimension of the Wilkinson W+ matrix to be generated

OUTPUT

W (Real Matrix): the generated n by n Wilkinson W+ matrix with elements:

$$w_{ij} = \begin{cases} [n/2] + 1 - \min(i, n-i+1) & \text{if } i = j \\ 1 & \text{if } i = j+1 \text{ or } j = i+1 \\ 0 & \text{otherwise} \end{cases}$$

for $1 \le i, j \le n$, where [x] denotes the largest integer less than or equal to x

EXAMPLE

```
// An example for: wilkinsonPlus(n), generate an n by n
// Wilkinson W+ matrix

// The row and column dimension is n = 5:
n = 5;
W = wilkinsonPlus(n);

// Results :
// W:  5 rows, 5 columns
// 2        1        0        0        0
// 1        1        1        0        0
// 0        1        0        1        0
// 0        0        1        1        1
// 0        0        0        1        2
```

ALGORITHM AND COMMENTS

The Wilkinson W+ matrix is normally given an odd order. This tridiagonal matrix has several pairs of close eigenvalues despite the fact that no superdiagonal element is small. The separation between the two largest eigenvalues is on the order of $(n!)^2$ so that the power method will be unable to separate them unless n is small.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, 2nd ed., Adam Hilger Ltd., 1990, p. 254

CHAPTER 15
SPECIAL MATRIX FUNCTIONS

■ band

FUNCTION

A = band(A, ml, mu)

PURPOSE

Convert a n by n band matrix A into its compact form, i.e., into a matrix *A*

INPUT

A (Real or Complex Matrix): the n by n band matrix

ml (Integer Scalar): the lower bandwidth of A

mu (Integer Scalar): the upper bandwidth of A

OUTPUT

A (Real or Complex Matrix) : a (2ml+mu+1) by n matrix

EXAMPLES

```
//  Examples for:  band(A,ml,mu)

//  Compute  band(B, ml, mu) for a 5 by 5 real band matrix B with
//   ml = 1,  mu=1:

B = { 1.1, -1.0,    0.0,  0.0,  0.0;
    -1.0,  2.2,   0.0,  0.0,  0.0;
   0.0,  -3.0,  3.3,  1.0,  0.0;
   0.0,   0.0,  5.0,  4.4,  2.0;
   0.0,   0.0,  0.0,  5.0,  5.5   };
D = band(B,1,1);

//  Results:
//  D:  4 rows, 5 columns
//     0          0          0        0         0
//     0         -1          0        1         2
//     1.1        2.2        3.3      4.4       5.5
//    -1         -3          5        5         0

//  Compute band(C, ml, mu) for a 5 by 5 complex band matrix C
//  with  ml = 0,  mu=1:
```

```
C = { (1.1, -1), (1.2, 2), (0,0), (0,0),  (0,0);
(0,0), (-1.0,  2.2),   (0.2,  -1.8), (0,0),  (0,0);
(0,0),    (0,0),          (3.3, 0),  (0, -3.4), (0,0);
(0,0),    (0,0),        (0,0),    (4.4, -4.4), (1,0);
(0,0),    (0,0),       (0,0),     (0,0),     (5, 2)  };
E = band(C,0,1);

//  Result s:
//  E:  2 rows, 5 columns
//      0+0i    1.2+2i     0.2-1.8i    0-3.4i   1+0i
//     1.1-1i    -1+2.2i      3.3+0i     4.4-4.4i   5+2i
```

ALGORITHM AND COMMENTS

This function is used to prepare matrices for use with the special band matrix routines bandLUD, bandBkSv, bandDet, bandSolve. The algorithm is similar to that provided in the reference.

REFERENCE

Dongarra, J.J., Moler, C. B., Bunch, J.R., and Stewart, G.W., *Linpack User's Guide*, SIAM, Philadelphia, 1979, p. 2.2

■ bandBkSv

FUNCTION

xVector = bandBkSv(LUD, pivot, ml, mu, bVector)

PURPOSE

Solve a n by n banded system:

$$Ax = b$$

where the information of the LU decomposition of A is available as the outputs of the function bandLUD():

[LUD, pivot] = bandLUD(*A*, ml, mu)

and *A* = band(A) is the compact form of the initial matrix A

INPUT

LUD (Real or Complex Matrix): the decomposed matrix resulting from the LU decomposition of the matrix *A* (the compact form of matrix A)

pivot (Integer Vector): a n-dimensional vector which contains the pivoting information of the LU decomposition

ml (Integer Scalar): the lower bandwidth of the given band matrix

mu (Integer Scalar): the upper bandwidth of the given band matrix

bVector (Real or Complex Vector): the n-dimensional vector of the right side containing the original system

OUTPUT

xVector (Real or Complex Vector): the computed solution vector of the banded system Ax = b

EXAMPLES

```
//  Examples for:  bandBkSv(LUD, pivot, ml, mu, b)
//

//  Compute bandBkSv(A,pivotA, ml,mu,bA) for a real banded
//  linear  system where A has been stored in its compact form
//  (i.e.,  a 5 by 6  matrix) with ml = 1 and mu = 2:

A =
{  0,      0,     0,      0,      0,      0;
   0,      0,     -1,     3,      1,    3;
   0,    -1,      -1,     1,    1.66666666666667,     1;
1,      1,    3,        3.66666666666667,    3,     -12;
-1,      0,   .333333333333333,    .545454545454545,
              1.36363636363636,         0
};
pivotA = {1; 2; 3; 4; 6; 6};
bA = { -4; 15;  18;  47; -21;  21};
xVector = bandBkSv(A,pivotA,1,2,bA);

//  Results :
// xVector:  6 rows
//                        1
//                        2
//                        3
//                        4
//                        5
//                        5.99999999999999

//  Compute bandBkSv(B,pivotB,ml,mu,bB) for a complex banded
//  linear  system where B has been stored in its compact form
//   (i.e.,
//  a 5 by 6  matrix) with ml = 0 and mu =1:

B = {     (0,0), (1.2,  2), (0.2,  -1.8),   (0, -3.4),  (1,0);
(1.1, -1),    (-1, 2.2),   (3.3, 0),  (4.4, -4.4),  (5, 2)};
pivotB = {1; 2; 3; 4; 5};
bB = {(3,5); (1.2, 3.2); (3.4, 3.4); (2,6.8); (14, -6)};
zVector = bandBkSv(B,pivotB,0,1,bB);

// Results :
//  zVector:  5 rows
//        -2 + 2i
//        1 -  1i
//        1.31418445149758e-19 + 1.31418445149758e-19i
//        -1 + 1i
```

```
//        2 -  2i
```

ALGORITHM AND COMMENTS

If the given n by n band matrix A, with lower bandwidth ml and upper bandwidth mu, is such that 2ml+mu+1 > n, then the routine bandBkSv is not suggested.

This function is usually used together with the function bandLUD. The algorithm used is similar to that provided in the reference.

REFERENCE

Dongarra, J.J., Moler, C. B., Bunch, J.R., and Stewart, G.W., *Linpack User's Guide*, SIAM, Philadelphia, 1979, p. 2.4

■ bandDet

FUNCTION

d = bandDet(*A*, ml, mu)

PURPOSE

Compute the determinant of a n by n band matrix A with its lower bandwidth ml and upper bandwidth mu in a compact storage mode (i.e., a matrix *A*)

INPUT

A (Real or Complex Matrix): the (2ml+mu+1) by n matrix which stores the element of the n by n band matrix A with lower bandwidth ml and upper bandwidth mu in a row (of the original matrix) to diagonal (of the compact matrix) fashion

ml (Integer Scalar): the lower bandwidth of the given band matrix

mu (Integer Scalar): the upper bandwidth of the given band matrix

OUTPUT

d (Real or Complex Scalar): the computed determinant of band matrix A

EXAMPLES

```
//  Examples for:  bandDet(A, ml, mu)
//

//  Compute bandDet(B, ml, mu) for a real band matrix B stored
// in its compact  form (i.e., a 4 by 5 matrix) with ml = 1,
//  mu=1:

B = { 0,      0,       0,      0,         0;
      0,     -1,       0,      1,         2;
      1.1,    2.2,     3.3,    4.4,       5.5;
```

```
     -1,     -3,       5,      5,         0};
D = bandDet(B,1,1);

//  Result :
//          D:         27.4912

//  Compute bandDet(C, ml, mu) for a complex band matrix C
// stored in its compact form (i.e., a 2 by 5 matrix) with
// ml = 0,  mu=1:

C = {     (0,0), (1.2,  2), (0.2,  -1.8),  (0, -3.4),     (1,0);
   (1.1, -1),    (-1, 2.2),   (3.3, 0), (4.4, -4.4),    (5, 2)};
E = bandDet(C,0,1);

//  Result :
//          E:   260.7792 + 299.6928i
```

ALGORITHM AND COMMENTS

The magnitude of the determinant is obtained by multiplying the diagonal elements of the upper triangular matrix resulting from the function bandLUD, which performs the LU decomposition of A in compact form. The sign of the desired determinant is then determined by the number of column interchanges performed during the LU decomposition. That is, the sign of the determinant is positive if there is an even (or zero) number of column interchanges when the LU decomposition is accomplished; otherwise the determinant has negative sign.

If the given n by n band matrix *A*, with lower bandwidth ml and upper bandwidth mu, is such that 2ml+mu+1 > n, then the routine bandDet is not suggested.

If the band matrix is not in its compact form, a conversion must first be done by using the function band() before calling this routine. The algorithm used is similar to that provided in the reference.

REFERENCE

Dongarra, J.J., Moler, C. B., Bunch, J.R., and Stewart, G.W., *Linpack User's Guide*, SIAM, Philadelphia, 1979, p. 2.4

■ bandLUD

FUNCTION

[LUD, pivot] = bandLUD(*A*, ml, mu)

PURPOSE

Perform the LU decomposition with partial pivoting for a given n by n band matrix A, with lower bandwidth ml and upper bandwidth mu, in a compact storage mode (i.e., a matrix *A*)

INPUT

A (Real or Complex Matrix): the (2ml+mu+1) by n matrix which stores the elements of a n by n band matrix with lower bandwidth ml and upper bandwidth mu in a row (of the original matrix) to diagonal (of the compact matrix) fashion

ml (Integer Scalar): the lower bandwidth of the given band matrix

mu (Integer Scalar): the upper bandwidth of the given band matrix

OUTPUT

LUD (Real or Complex Matrix): the (2ml+mu+1) by n matrix which contains the computed lower and upper triangular matrices of the LU decomposition of the input matrix *A*

pivot (Integer Vector): an integer n-dimensional vector which contains the pivoting information of the LU decomposition

EXAMPLES

```
//  Examples for: bandLUD(A, ml, mu)
//
//  Compute bandLUD(B, ml, mu) for a real band matrix B stored
// in its compact  form (i.e., a 4 by 5 matrix)
// with ml = 1,  mu=1:

B = {     0,       0,       0,        0,       0;
   0,     -1,       0,       1,       2;
   1.1,   2.2,   3.3,   4.4,    5.5;
   -1,     -3,       5,       5,        0};
[ludB, pB] = bandLUD(B,1,1);

//  Results :
//  ludB:  4 rows, 5 columns
//      0        0      0        1     2
//      0       -1      3.3      4.4   5.5
//          1.1       -3        5            5      0.333226666666667
//   -0.909090909090909  -0.43030303030303  0.284
//                           -0.163859393939394    0
//
//  pB:  5 rows
//     1
//     3
//     4
//     5
//     5

//  Compute bandLUD(C, ml, mu) for a complex band
//  matrix C stored
//  in its compact form (i.e., a 2 by 5 matrix) with ml = 0,
//  mu=1

C = {(0,0), (1.2,  2), (0.2,  -1.8), (0, -3.4), (1,0);
```

```
   (1.1, -1), (-1, 2.2), (3.3, 0),  (4.4, -4.4),  (5, 2)};
[ludC, pC] = bandLUD(C,0,1);

//  Results :
//  ludC:  2 rows, 5 columns
//      0 +  0i    1.2 + 2i      0.2 -1.8i       0 -3.4i      1 + 0i
//     1.1 -1i       -1 + 2.2i       3.3 +0i    4.4 -4.4i     5+2i
//
//  pC:  5 rows
//     1
//     2
//     3
//     4
//     5
```

ALGORITHM AND COMMENTS

If the given n by n band matrix *A*, with lower bandwidth ml and upper bandwidth mu, is such that 2ml+mu+1 > n, then the routine bandLUD is not suggested.

If the band matrix is not in its compact form, a conversion must first be done by using the function band() before calling this routine. The algorithm used is similar to that provided in the reference.

REFERENCE

Dongarra, J.J., Moler, C. B., Bunch, J.R., and Stewart, G.W., *Linpack User's Guide*, SIAM, Philadelphia, 1979, p. 2.4

■ bandSolve

FUNCTION

xVector = bandSolve(*A*, bVector, ml, mu)

PURPOSE

Solve a given n by n banded system:

$$Ax = b$$

where A is a band matrix with lower bandwidth ml and upper bandwidth mu stored in a compact storage mode (i.e., a matrix *A*)

INPUT

A (Real or Complex Matrix): the (2ml+mu+1) by n matrix which stores the element of the n by n band matrix A with lower bandwidth ml and upper bandwidth mu in a row (of the original matrix) to diagonal (of the compact matrix) fashion

bVector (Real or Complex Vector): the n-dimensional vector containing the right hand side of the system

ml (Integer Scalar): the lower bandwidth of the given band matrix

mu (Integer Scalar): the upper bandwidth of the given band matrix

OUTPUT

xVector (Real or Complex Vector): the computed n-dimensional solution vector

EXAMPLES

```
//  Examples for: bandSolve(A, b, ml, mu)
//
//  Compute bandSolve(A,bA, ml,mu) for a real banded linear
//  system where A has been stored in its compact form
//  (i.e., a 5 by 6 matrix) with ml = 1 and mu = 2:
A = {  0,  0,  0,  0,  0,  0;
   0,  0, -1,  3,  1,  3;
   0, -1,  0,  1,  2, -9;
   1,  2,  3,  4,  5,   1;
   -1,  0,  1,  2,  3,   0};
bA = { -4; 15;  18;  47; -21;  21};
xVector = bandSolve(A,bA,1,2);

//  Results :
//  xVector:  6 rows
//    1
//    2
//    3
//    4
//    5
//    6

//  Compute bandSolve(B,bB, ml,mu) for a complex banded linear
//  system where B has been stored in its
//  compact form (i.e., a 2 by 5
//  matrix)  with ml = 0 and mu =1:

B = {     (0,0), (1.2,  2), (0.2,  -1.8), (0, -3.4), (1,0);
     (1.1, -1),  (-1, 2.2), (3.3, 0),  (4.4, -4.4),   (5, 2)};
bB = {(3,5); (1.2, 3.2); (3.4, 3.4); (2,6.8); (14, -6)};
zVector = bandSolve(B,bB,0,1);

//  Results :
//  zVector:  5 rows :
//                        -2 + 2i
//                           1 -1i
//     1.31418445149758e-19  + 1.31418445149758e-19i
//                        -1  + 1i
//                          2  -2i
```

ALGORITHM AND COMMENTS

If the given n by n band matrix A, with lower bandwidth ml and upper bandwidth mu, is such that 2ml+mu+1

> n, then this routine should not be used.

REFERENCE

Dongarra, J.J., Moler, C. B., Bunch, J.R., and Stewart, G.W., *Linpack User's Guide*, SIAM, Philadelphia, 1979, p. 2.4

■ cho

FUNCTION

u = cho(*S*, n)

PURPOSE

Perform the Cholesky decomposition of an n by n symmetric positive definite matrix S stored in compact form

INPUT

S (Real Vector): the n(n+1)/2 - dimensional vector which stores the lower triangular part of the n by n symmetric matrix S in a column by column sequence (compact form)

n (Integer Scalar): the row and column dimension of A

OUTPUT

u (Real Vector): the n(n+1)/2 - dimensional vector which contains the resultant lower triangular matrix, L, of the Cholesky decomposition in a column by column sequence (compact form)

EXAMPLES

```
// An example for: cho(S, n)
//  Compute cho(A,n) for a 4 by 4 positive definite
// matrix A stored
//  in its compact form (i.e., a 10-dimensional vector):

A = {1; 1; 1; 1; 2; 2; 2; 3; 3; 4};
B = cho(A,4);

//  Results:
//  B:  10 rows
//                        1
//                        1
//                        1
//                        1
//                        1
//                        1
//                        1
//                        1
```

```
//                      1
//                      1
```

ALGORITHM AND COMMENTS

If the symmetric matrix *S* is not in its compact form, a conversion must first be done by calling function convSym() before calling this function.

REFERENCE

Golub, G.H and Van Loan, C.F., *Matrix Computations,* 2nd ed., Johns Hopkins, 1989

■ defRankLS

FUNCTION

xVector = defRankLS(A, bVector)

PURPOSE

Solve the possible rank deficient least-squares problem, i.e., to

$$\text{minimize} \quad \|Ac - b\|$$

with rank(A) ≤ min(m,n), where ||•|| denotes the Euclidean norm in R^m, A is an m by n matrix, b is an m-dimensional vector and c is the unknown n-dimensional vector

INPUT

A (Real Matrix): the given m by n coefficient matrix, i.e., rank(A) ≤ min(m,n), in the least-squares problem

bVector (Real Vector): the m-dimensional vector in the least-squares problem

OUTPUT

xVector (Real Vector): the n-dimensional vector containing the computed solution for the least-squares problem

EXAMPLES

```
//  An example for: defRankLS(A, bVector)
//  Compute defRankLS(A,b) for a 5 by 6 rank deficient matrix A
//  (i.e., rank of A is only 2) and a 5-dimensional vector b:

A= {1,   2,   3,   4;
    5,   6,   7,   8;
    9,  10,  11,  12;
    1,   1,   1,   1;
    3,   2,   1,   0};
bVector = {10; 26; 42; 4; 6};
xVector= defRankLS(A,bVector);
```

```
//  Results :
//  xVector:  4 rows
//                        1
//                        1
//                        1
//                        1
```

ALGORITHM AND COMMENTS

The method for solving possible rank deficient least-squares problems uses the well-known Householder transformation on A so that A = QR when $m \geq n$, similar to solving the full rank least-squares problem. For illustrative convenience, we only consider the case for $\text{rank}(A) < \min(m,n)$ in the following discussion. It becomes obvious when $\text{rank}(A) = \min(m,n)$.

The computation of the solution for rank deficient least-squares problems are classified by two cases:

Case 1: $(0 < n' =) \text{Rank}(A) < n \leq m$ (overdetermined problems).

Since an orthogonal transformation preserves the Euclidean norm, we can obtain the following equalities :

$$\begin{aligned}
\text{minimize } \|Ac - b\| &= \text{minimize } \|A(PP)c - b\| \\
&= \text{minimize}\|A(APPc - b)\| \\
&= \text{minimize}\|(QAP)y - Qb\| \\
&= \text{minimize}\left\|\begin{bmatrix} W \\ 0 \end{bmatrix} y - \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}\right\| \\
&= \text{minimize}\left\|\begin{bmatrix} Wy - g_1 \\ -g_2 \end{bmatrix}\right\| \\
&= \text{minimize}\left\|\begin{bmatrix} W \\ 0 \end{bmatrix} y - \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}\right\| \\
&= \text{minimize}\|Wy - g_1\|_{(n')} + \|g_2\|_{(m-n')}
\end{aligned}$$

where $\|.\|_{(n')}$ and $\|.\|_{(m-n')}$ denote the euclidean norm in $R^{n'}$ and $R^{m-n'}$ respectively, P is an n by n permutation matrix which performs the column interchanges (see comments for the function fullRankLS), Q is the m by m Householder orthogonal matrix, W is a n' by n matrix of rank n' with its n' by n' principal minor in upper triangular form, g_1 is an n'-dimensional vector , g_2 is an (m-n')-dimensional vector, such that

$$QAP = \begin{bmatrix} W \\ 0 \end{bmatrix} \quad \text{and} \quad Qb = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}$$

and y is the n-vector satisfying $c = Py$.

Since to minimize $\| Wy - g_1 \|_{(n')}$ with rank(W) = n' < n is an under-determined full rank least-squares problem, the unique minimum length solution y_{min} is computed by the method described in the function fullRankLS.

Once y_{min} is calculated, the least-squares solution is obtained immediately as:

$$y_{min} = Py_{min}$$

Note that $\| Wy - g_1 \|_{(n')} = 0$, so $\| Ac_{min} - b \| = \| g_2 \|_{(m-n')}$.

Case 2: (0 < n' =) Rank(A) < m < n (underdetermined problems).

Applying the column interchange and Householder orthogonal transformation on the transpose of A, A^t, we obtain the following relations :

$$\text{minimize } \|Ac - b\| = \text{minimize } \|PA(Q^tQ)c - Pb\|$$

$$= \text{minimize} \| (PAQ^t)(Qc) - g \|$$

$$= \text{minimize} \left\| \begin{bmatrix} W^t & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} - g \right\|$$

$$= \text{minimize} \| W^t y_1 - g \|$$

where P is an m by m permutation matrix which performs the column interchanges (see comments for the function fullRankLS) on A^t, Q is the n by n Householder orthogonal matrix of A^tP, W is an n' by m matrix of rank n' with its n' by n' principal minor in upper triangular form such that:

$$QA^tP = \begin{bmatrix} W \\ 0 \end{bmatrix}$$

and $g = Pb$ is an m-dimensional vector, y_1 is an n'-dimensional vector, y_2 is an (n-n')-dimensional vector satisfying

$$c = Q^t \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Since to minimize $\| W^t y_1 - g \|$ with $\text{rank}(W^t) = n' < m$ is an over-determined full rank least-squares problem, the unique y_{min} is computed by the method described in the function fullRankLS.

Once y_{min} is calculated, the "minimum length" least-squares solution is obtained immediately as

$$c_{min} = Q^t \begin{bmatrix} y_{min} \\ 0 \end{bmatrix}$$

Note that

$$\| Ac_{min} - b \| = \| W^t y_{min} - g \|$$

Comment :

The function defRankLS is designed to work also for the case when
rank(A) = min(m,n). However, for this case, fullRankLS is more efficient in computing the solution.

REFERENCE

Lawson, C. L. and Hanson, R. J., *Solving Least Squares Problems*, Prentice Hall, Englewood Cliffs, New Jersey, 1974, pp. 77-82

■ fastGQRD

FUNCTION

[Q, R] = fastGQRD(A)

PURPOSE

Compute the QR decomposition of a given matrix A using fast Givens rotations

INPUT

A (Real Matrix): a given m by n matrix

OUTPUT

Q (Real Matrix): a m by m orthogonal matrix

R (Real Matrix): a m by n upper triangular matrix

EXAMPLES

```
//  An example for: fastGQRD(A)
//  Compute fastGQRD(A) for a 3 by 3  real matrix A:

A = {2, 10,  8;
   1,  4, -2;
   0,  2,  3};
```

```
[QA, RA] = fastGQRD(A);
//  Results:
//  QA:  3 rows, 3 columns
// 0.894427190999916    0.182574185835055   -0.408248290463863
// 0.447213595499958   -0.365148371670111    0.816496580927726
// 0                    0.912870929175277    0.408248290463863
//
//  RA:  3 rows, 3 columns
// 2.23606797749979    10.733126291999       6.26099033699941
// 0                    2.19089023002066     4.9295030175465
// 0                    0                   -3.67423461417477
```

SEE ALSO

QRD(), GQRD(), mGS()

ALGORITHM AND COMMENTS

The result is the pair of matrices Q and R, where Q is the orthogonal m by m matrix and R is the upper triangular m by n matrix such that:

$$A = QR$$

is satisfied.

The algorithm consists of the following loop. At each step we eliminate one by one elements of R by using the fast Givens rotation g. We move from column to column from left to right, and within the column from the bottom element to the element right under the main diagonal. In what follows ,M(i) is the analog of the orthogonal matrix, D(i) is diagonal, R(i) has zero elements in some columns under the main diagonal, and at the beginning we let:

$$R(0) = a, \quad D(0) = E$$

where E is the identity matrix.

At each step we will satisfy the conditions:

$$\mu(i)^t D(i-1)\mu(i) = D(i), \quad R(i) = \mu(i)^t R(i-1)$$

and eliminate one additional non-zero element of R(i-1). It is possible to do that by choosing $\mu(i)$ as a fast Givens rotation that has only four non-trivial elements, and is equal to

$$\begin{bmatrix} \alpha & 1 \\ 1 & \beta \end{bmatrix} \text{ (type 1), or to } \begin{bmatrix} 1 & \alpha \\ \beta & 1 \end{bmatrix} \text{ (type 2)}$$

The advantage of this more complicated approach, if we compare it with the Givens algorithm, is that we do not need to compute square roots as in the case of Givens rotations. At the end of the loop, when i is equal to some q, put

$$M = \mu(1)\mu(2) \ldots \mu(q), \quad D = D(q), \quad R = R(q)$$

$$d = \frac{1}{\sqrt{R}} \qquad q = Md \qquad r = dR$$

Then

$$M^tM = D \qquad M^ta = R$$
$$qr = Mddr = M(1/D)(M^ta) = a$$

r is upper triangular, and q is orthogonal , since

$$q^tq = dM^tMd = dRd = E$$

Some special means were taken to handle the situation where the current element of R is equal or approximately equal to zero. If this occurs for all the elements of one column, this indicates a degenerate rank.

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, pp. 211-216

■ fullRankLS

FUNCTION

xVector = fullRankLS(A, bVector)

PURPOSE

Solve the full rank least-squares problem, i.e., to

$$\text{minimize } \|Ac - b\|$$

with rank(A) = min(m,n), where ||.|| denotes the Euclidean norm in R^m, A is an m by n matrix, b is an m-dimensional vector and c is the unknown n-dimensional vector

INPUT

A (Real Matrix): the given full rank m by n coefficient matrix, i.e., rank(A) = min(m,n), in the least-squares problem

bVector (Real Vector): the m-dimensional vector in the least-squares problem

OUTPUT

xVector (Real Vector): the n-dimensional vector containing the computed solution for the least-squares problem

EXAMPLES

```
// An example for: fullRankLS(A, bVector)
// Compute fullRankLS(A,bVector) for a 3 by 2 full  rank
// matrix A (i.e., rank of A is 2) and a
// 3-dimensional vector bVector :

A= {3, -2;
   0,  3;
    4,  4};
bVector = {1; 2; 4};
xVector = fullRankLS(A,bVector);
//  Results:
//  xVector:  2 rows
//                  0.5616
//                  0.496
```

ALGORITHM AND COMMENTS

The method for solving full rank least-squares problems used here is based on the orthogonal QR decomposition of the given m by n matrix A. The decomposition is accomplished by applying the well-known Householder transformation on A so that A = QR when $m \geq n$, where Q is an m by m orthogonal matrix and R is an upper triangular m by n matrix. For details about this elementary orthogonal transformation, see the reference.

The computation of solution for the full rank least-squares problems are classified by two cases:

Case 1: Rank(A) = $n \leq m$ (overdetermined problems).

Since orthogonal transformation preserves the euclidean norm, we have the following equalities :

$$\begin{aligned}
\text{minimize } \|Ac - b\| &= \text{minimize } \|Q(Ac - b)\| \\
&= \text{minimize } \|(QA)c - Qb\| \\
&= \text{minimize } \left\| \begin{bmatrix} R \\ 0 \end{bmatrix} c - \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} \right\| \\
&= \text{minimize } \left\| \begin{bmatrix} Rc - g_1 \\ -g_2 \end{bmatrix} \right\| \\
&= \text{minimize } \left(\|Rc - g_1\|_{(n)} + \|g_2\|_{(m-n)} \right)
\end{aligned}$$

where $\|.\|_{(n)}$ and $\|.\|_{(m-n)}$ denote the euclidean norm in R^n and R^{m-n}, respectively, Q is the m by m Householder orthogonal matrix, R is an n by n nonsingular upper triangular matrix, g_1 is an n-dimensional vector , g_2 is an (m-n)-dimensional vector, such that

$$QA = \begin{bmatrix} R \\ 0 \end{bmatrix} \qquad \text{and} \qquad Qb = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}$$

Since $\| Rc - g_1 \|_{(n)} \geq 0$ and R is nonsingular, so the least-squares solution is

$$c_{min} = R^{-1} g_1$$

Note that

$$\|Ac_{min} - b\| = \|g_2\|_{(m-n)}$$

Case 2: Rank(A) = m < n (underdetermined problems).

Applying the Householder orthogonal transformation on the transpose of A, i.e., A^t, we obtain the following relations :

$$\begin{aligned} \text{minimize } \|Ac - b\| &= \text{minimize } \|A\,(Q^tQ)\,c - b\| \\ &= \text{minimize } \|(AQ^t)\,(Qc) - b\| \\ &= \text{minimize } \left\| \begin{bmatrix} R^t & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} - b \right\| \\ &= \text{minimize } \|R^t y_1 - b\| \end{aligned}$$

where Q is the n by n Householder orthogonal matrix for A^t, R is a m by m nonsingular upper triangular matrix such that:

$$QA^t = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

and y_1 is an m-dimensional vector, y_2 is an (n-m)-dimensional vector satisfying

$$c = Q^t \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Since $\| R^t y_1 - b \| \geq 0$ and R is nonsingular, so $y_1 = (R^t)^{-1} b$ and

$$c_{min} = Q^t \begin{bmatrix} (Q^t)^{-1} b \\ y_2 \end{bmatrix}$$

is a least-squares solution for any (n-m)-vector y_2. In our case, we choose y_2 to be the zero vector, and c_{min} is called the "minimum length" solution.

Note that $\| Ac_{min} - b \| = 0$.

Comments :

1) To reduce the rounding error in performing the QR decomposition using the Householder transformation, a column interchange strategy is used in the function fullRankLS. For more information see the reference.

2) The function fullRankLS is designed to work only for the case when rank(A) = min(m,n). If the rank of A is known to be less than min(m,n) or even not known in advance , the function defRankLS should be used instead.

REFERENCE

Lawson, C. L. and Hanson, R. J., *Solving Least Squares Problems*, Prentice Hall, Englewood Cliffs, New Jersey, 1974, pp. 9-11, 32, 63-66, 74-76

■ GQRD

FUNCTION

[Q, R] = GQRD(A)

PURPOSE

Compute the QR decomposition of a given matrix A using Givens rotations

INPUT

A (Real Matrix): a given m by n matrix

OUTPUT

Q (Real Matrix): am by m orthogonal matrix

R (Real Matrix): a m by n upper triangular matrix

EXAMPLES

```
//  An example for: GQRD(A)
//  Compute GQRD(A) for a 3 by 3 real matrix A:

A = {2, 10,  8;
   1,  4, -2;
   0,  2,  3};
[QA, RA] = GQRD(A);
 //  Results:
//  QA:  3 rows, 3 columns
//  0.894427190999916   0.182574185835055    0.408248290463863
//  0.447213595499958  -0.365148371670111   -0.816496580927726
//  0                   0.912870929175277   -0.408248290463863
//  RA:  3 rows, 3 columns
//  2.23606797749979   10.733126291999       6.26099033699941
//  0                   2.19089023002066     4.9295030175465
// -0                   0                    3.67423461417477
```

SEE ALSO

QRD(), fastGQRD(), mGS()

ALGORITHM AND COMMENTS

This routine returns an orthogonal m by m matrix Q and upper triangular m by n matrix R, where:

$$A = QR$$

The algorithm consists of the following loop. At the beginning, R = A and G =E, where E is the identity matrix. At each step we eliminate one by one the elements of R by multiplying R by the Givens rotation G. We move from column to column from left to right, and within the column we move from the bottom element to the element right under the main diagonal. The rotation is the matrix of the form:

$$G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \qquad c^2 + s^2 = 1$$

or more precisely, this block stands in two corresponding rows and columns of the matrix m by m matrix G. The loop operation consists of finding an appropriate G, and of two multiplications:

$$G^t = GG^t, \qquad R^t = GG^t$$

$$= \text{minimize} \left\| \begin{bmatrix} Rc - g_1 \\ -g_2 \end{bmatrix} \right\|$$

Thus, GA = R is the loop invariant. Special functions for the Givens product are, of course, faster than the usual matrix multiplication. Since G is orthogonal, at the end we put $R = R^t$, $Q = G^t$. Some special means were taken to handle the situation where the current element of R^t is equal or approximately equal to zero. If this occurs for all the elements of one column, this indicates the rank of degeneracy.

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, pp. 211-216.

■ hAP

FUNCTION

B = hAP(A, u)

PURPOSE

Compute AH^u for the square matrix A and the Householder matrix H^u defined by the vector u

INPUT

A (Real Matrix): a m by m square matrix

u (Real Vector): a m-dimensional nonzero vector

OUTPUT

B (Real Matrix): the result of multiplication of H^u by A

EXAMPLES

```
//  An example for: hAP(A, u)
//  Compute hAP(A,u) for a 3 by 3 real matrix A and a
//  3-dimensional  vector u :

A= {1, 2, 3;
    4, 5, 6;
    7, 8, 9};
u = { 1; -0.2; -0.3};
B = hAP(A,u);
//  Results :
//  B:  3 rows, 3 columns
//   1.53097345132743   1.89380530973451   2.84070796460177
//   1.87610619469027   5.42477876106195   6.63716814159292
//   2.2212389380531    8.95575221238938  10.4336283185841
```

SEE ALSO

hVector(), hPA(), hPV(), QRD()

ALGORITHM AND COMMENTS

The Householder matrix defined by a vector u is the orthogonal matrix:

$$H^u = I - \frac{2uu^t}{u^t u}$$

where I is the identity matrix.

In operator form this is orthogonal transformation

$$H^u = I - \frac{2u\langle u, . \rangle}{\langle u, u \rangle}$$

In coordinate form we have:

$$H^u{}_{i,j} = \delta_{ij} - \frac{2u_i u_j}{\sum_{i,j} u_i u_j}$$

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, pp. 195-198

■ hPA

FUNCTION

B = hPA(u, A)

PURPOSE

Compute $H^u A$ for the square matrix A and the Householder matrix H^u defined by the vector u

INPUT

u (Real Vector): a m-dimensional nonzero real vector

A (Real Matrix): a m by m square matrix

OUTPUT

B (Real Matrix): the result of multiplication of A by H^u

EXAMPLE

```
//  An example for: hPA(u,A)
//  Compute the hPA(u,A) for 3-dimensional vector u
//  and a 3 by 3 matrix A:

u = { 1; -0.2; -0.3};
A= {1, 2, 3;
    4, 5, 6;
    7, 8, 9};
B = hPA(u,A);

//  Results :
//  B:  3 rows, 3 columns
//     4.36283185840708   4.47787610619469   4.5929203539823
//     3.32743362831858   4.50442477876106   5.68141592920354
//     5.99115044247788   7.25663716814159   8.52212389380531
```

SEE ALSO

hVector(), hAP(), hPV(), QRD()

ALGORITHM AND COMMENTS

The Householder matrix defined by a vector u is the orthogonal matrix:

$$H^u = I - \frac{2uu^t}{u^t u}$$

where I is the identity matrix.

In operator form this is orthogonal transformation:

$$H^u = I - \frac{2u\langle u, . \rangle}{\langle u, u \rangle}$$

In coordinate form we have:

$$H^u{}_{i,j} = \delta_{ij} - \frac{2u_i u_j}{\sum_{i,j} u_i u_j}$$

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, pp. 195-198

■ hPV

FUNCTION

w = hPV(u, v)

PURPOSE

Compute $H^u v$ for the vector v and the Householder matrix H^u defined by the vector u

INPUT

u (Real Vector): a m-dimensional vector with the first component unity

v (Real Vector): a n-dimensional

OUTPUT

w (Real Vector): the result of multiplication of H^u by v

EXAMPLES

```
//  An example for: hPV(u,v)
//  Compute hPV(u,V) for 5-dimensional vectors u and v:

u = { 1;   0.237636981003543; 0.356455471505315 ;
   0.475273962007086; 0.594092452508858};
v = { -5; 4; -3; 2; -1};
w = hPV(u,v);
```

```
//  Results :
//  w:  5 rows
//           0.404519917477946
//           5.28431379696298
//          -1.07352930455553
//           4.56862759392595
//           2.21078449240744
```

SEE ALSO

hVector(), hPA(), hPA(), QRD()

ALGORITHM AND COMMENTS

The Householder matrix defined by a vector u is the orthogonal matrix equal to:

$$H^u = I - \frac{2uu^t}{u^t u}$$

where I is the identity matrix.

In operator form this is orthogonal transformation

$$H^u = I - \frac{2u\langle u, . \rangle}{\langle u, u \rangle}$$

In coordinate form we have:

$$H^u{}_{i,j} = \delta_{ij} - \frac{2u_i u_j}{\sum_{i,j} u_i u_j}$$

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, pp. 195-198

■ hVector

FUNCTION

v = hVector(u)

PURPOSE

Compute the Householder's vector corresponding to the vector u

INPUT

u (Real Vector): a n-dimensional nonzero vector

OUTPUT

v (Real Vector): the Householder vector corresponding to the vector u

EXAMPLES

```
//  An example for: hVector(u)
//  Compute hVector(u) for a 5-dimensional vector u :

u = {1; 2; 3; 4; 5};
v = hVector(u);

//  Result :
//  v:  5 rows
//        1
//        0.237636981003543
//        0.356455471505315
//        0.475273962007086
//        0.594092452508858
```

SEE ALSO

hPA(), hAP(), hPV(), QRD()

ALGORITHM AND COMMENTS

The Householder vector h is obtained as a linear combination of u and e_1= (1, 0, 0, ..., 0):

$$h = \pm\|u\|e_1 + u$$

We normalize h by condition $h_1 = 1$, and choose the sign to avoid cancellation.

The main application of h is due to the special quality of the corresponding Housholder matrix H^h:

$$H^h u = -|u|e_1$$

which is important in inductive algebraic transformations. Here the Householder's matrix associated with u is the orthogonal matrix equal to:

$$H^u = I - \frac{2uu^t}{u^t u}$$

where I is the identity matrix.

In operator form this is orthogonal transformation:

$$H^u = I - \frac{2u\langle u, . \rangle}{\langle u, u \rangle}$$

In coordinate form we have:

$$H^u{}_{i,j} = \delta_{ij} - \frac{2u_i u_j}{\sum_{i,j} u_i u_j}$$

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, pp. 195-198

■ lTriDet

FUNCTION

d = lTriDet(*A*, n)

PURPOSE

Compute the determinant of a lower triangular matrix A stored in compact form *A*.

INPUT

A (Real or Complex Vector): a n(n+1)/2 - dimensional vector which stores the lower triangular matrix A in a column by column sequence (compact form)

n (Integer Scalar): the row and column dimension of A

OUTPUT

d (Real Scalar): the computed determinant of the lower triangular matrix A

EXAMPLES

```
//  Examples for: lTriDet(A, n)

//  Compute lTriDet(A,n) for a 4 by 4 real lower triangular
//    matrix A stored in its compact form
//     (i.e., a 10-dimensional vector):
```

```
A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
dA = lTriDet(A,4);

//  Result :
//           dA:      0.009438

//  Compute lTriDet(B,n) for a 3 by 3 complex lower triangular
//  matrix B stored in its compact form (i.e., a 6-dimensional
//  vector).

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
dB = lTriDet(B, 3);

//  Result :
//           dB:     -7.324 -4.544i
```

ALGORITHM AND COMMENTS

If the triangular matrix A is not in the vector form, a conversion must first be done by calling the routine convLTriag() before calling this function.

■ lTriInv

FUNCTION

B = lTriInv(*A*, n)

PURPOSE

Compute the inverse of a nonsingular lower triangular matrix A stored in compact form

INPUT

A (Real or Complex Vector): a n(n+1)/2 - dimensional vector which stores the lower triangular matrix A in a column by column sequence (compact form)

n (Integer Scalar): the row and column dimension of A

OUTPUT

B (Real or Complex Vector): the computed inverse of A in compact form

EXAMPLES

```
//  Examples for: lTriInv(A, n)

//  Compute lTriInv(A,n) for a 4 by 4 real lower triangular
//  matrix A stored in its compact form (i.e., a 10-dimensional
//   vector):
```

```
A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
C = lTriInv(A,4);

//  Results :
//  C:  10 rows
//         1
//        -1.51515151515152
//         0.243939393939394
//         0.0859610086882814
//         3.03030303030303
//        -3.78787878787879
//         0.185420639966095
//         5
//        -5.83916083916084
//         6.99300699300699

//  Compute lTriInv(B,n) for a 3 by 3 complex lower triangular
//  matrix B stored in its compact form (i.e., a 6-dimensional
//  vector):

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
D = lTriInv(B, 3);

//  Results:
//  D:  6 rows
//        0.5                -0.5i
//       -0.181922168950327  +0.3449828421231 96i
//       -0.0174905832515087 -0.0246481143780918i
//        0.0803134658911144 -0.486748278127966i
//       -0.0349042667363334 +0.328864689793815i
//        0.0221238938053097 -0.3318584070796 46i
```

ALGORITHM AND COMMENTS

If the triangular matrix A is not in its vector form, a conversion must first be done by calling the routine convLTriag() before calling this function.

■ lTriSolve

FUNCTION

xVector = lTriSolve(*A*, bVector)

PURPOSE

Compute the solution of a lower triangular system of linear equations:

$$Ax = b$$

where A is stored in a compact storage mode *A*

INPUT

A (Real or Complex Vector): a n(n+1)/2 - dimensional vector which stores the lower triangular matrix A in a column by column sequence (compact form)

bVector (Real Vector): the n-dimensional vector of the right side containing the linear system of equations

OUTPUT

xVector (Real or Complex Vector): the n-dimensional solution vector of Ax = b

EXAMPLES

```
//  Examples for: lTriSolve(A, b)

//  Compute lTriSolve(A, bA) for a real lower triangular
//   linear system where A is a 4 by 4 matrix stored in
//  its compact form (i.e., a 10-dimensional vector) :

A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
bA = {1; 1.16; 1.43; 1.723};
xVector = lTriSolve(A,bA);

//  Results :
//  xVector:  4 rows
//                        1
//                        2
//                        3
//                        4

//  Compute lTriSolve (B,bB) for a complex lower
//  triangular linear system where B is a 3 by 3 matrix stored
//  in its compact form (i.e., a 6-dimensional vector):

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
bB ={(2.1, 0.1); (6.276,  4.34); (15.573, 13.97)};
zVector = lTriSolve(B,bB);
//  Results:
//  zVector:  3 rows
//                      1.1 -1i
//                      2.2 -2i
//                      3.3 -3i
```

ALGORITHM AND COMMENTS

If the triangular matrix A is not in its vector form, a conversion must first be done by calling the routine convLTriag() before calling this function.

■ mGS

FUNCTION

[Q, R] = mGS(A)

PURPOSE

Compute the QR decomposition of a given matrix A using the modified Gram-Schmidt method

INPUT

A (Real Matrix): a given m by n matrix

OUTPUT

Q (Real Matrix): a m by k orthogonal matrix with rank k)

R (Real Matrix): an upper triangular k by n matrix

EXAMPLES

```
//  An example for: mGS(A)
//  Compute the mGS(A) for a 3 by 3 real matrix A:

A = {2, 10,  8;
   1,  4, -2;
   0,  2,  3};
[QA, RA] = mGS(A);

//  Results :
//  QA:  3 rows, 3 columns
// 0.894427190999916    0.182574185835055    0.408248290463863
// 0.447213595499958   -0.365148371670111   -0.816496580927726
// 0                    0.912870929175277   -0.408248290463863
//
//  RA:  3 rows, 3 columns
// 2.23606797749979    10.733126291999         6.26099033699941
// 0                    2.19089023002066       4.9295030175465
// 0                    0                      3.67423461417477
```

SEE ALSO

QRD(), GQRD(), fastGQRD()

ALGORITHM AND COMMENTS

The result is the pair of matrices Q and R where Q is an orthogonal (m by k, k = rank) matrix, and R is an upper triangular rank by n matrix:

$$A = QR$$

is satisfied.

The algorithm consists of the following loop. The main idea of the step k is to recompute columns a[k+1], ... ,a[n] of matrix A, putting the result in Q. Thus on the kth step we achieve the orthogonality to q[1], ... , q[k]. If v[j] is the jth, j>k, vector column of A that is already orthogonal to the q[1], ... , q[k-1], then on the kth step:

$$q[k] = \frac{a[k]}{\|a[k]\|}, \qquad \text{and if} \quad a[q] \neq 0$$

$$v[j] = v[j] - (v[j], q[k]), \qquad k < j \leq n$$

gives the desired reorthogonalization. The elements of R can be stored as the corresponding scalar products.

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, Ch.5. pp. 211-216

■ posBkSv

FUNCTION

xVector = posBkSv(L, bVector)

PURPOSE

Solve the symmetric positive definite linear system:

$$LL^t x = b$$

by forward and backward substitution in a compact form where L is a nonsingular lower triangular matrix

INPUT

L (Real Vector): the n(n+1)/2-dimensional vector which stores the n by n lower triangular matrix, normally resulting from using the Cholesky decomposition function cho(), in a column by column sequence

bVector (Real Vector): the right hand side n-dimensional vector for the system

OUTPUT

xVector (Real Vector): the computed n-dimensional solution vector of $LL^t x = b$

EXAMPLES

```
//  An example for: posBkSv(L,  bVector)

//  Compute posBkSv(A,n) for a positive definite linear system
// with A the lower triangular matrix such that A*A^t is the
// Cholesky decomposition of the given 4 by 4 positive
```

```
// definite  matrix. Here A is in its compact
// form (i.e., a 10-dimensional vector) :

A = {1; 1; 1; 1; 1; 1; 1; 1; 1; 1};
bVector = {10; 19; 26; 30};
xVector = posBkSv(A,bVector);

//  Results:
//  xVector:  4 rows
//                        1
//                        2
//                        3
//                        4
```

SEE ALSO

cho(A, n)

ALGORITHM AND COMMENTS

Normally, when solving a single symmetric positive definite system Ax = b with A already in its compact form, we can conveniently call the function posSolve, which is equivalent to calling the function cho followed by posBkSv, to obtain the desired solution. The advantage of using posBkSv appears when there is a need to sequentially solve $Ax_1 = b_1$, ... , $Ax_k = b_k$ for some k > 1. In such cases, posBkSv can be used repeatedly (k times) once cho has been called.

REFERENCE

Golub, G.H. and VanLoan, C.F., *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989, pp. 142

■ posDet

FUNCTION

d = posDet(*S*, n)

PURPOSE

Compute the determinant of an n by n symmetric positive definite matrix S, stored in compact form

INPUT

S (Real Vector): the n(n+1)/2 - dimensional vector which stores the lower triangular part of the n by n symmetric matrix S in a column by column sequence (compact form)

n (Integer Scalar): the row and column dimension of S

OUTPUT

d (Real Scalar): the computed determinant of the given n by n symmetric positive definite matrix S

EXAMPLE

```
//  An example for: posDet(S, n)

//  Compute posDet(A,n) for a 4  by 4 positive  definite
// matrix A  stored in its compact form
// (i.e., a 10-dimensional vector):

A = {1; 1; 1; 1; 2; 2; 2; 3; 3; 4};
d = posDet(A,4);

//  Result :    d:         1
```

ALGORITHM AND COMMENTS

If the symmetric matrix S is not in its compact form, a conversion must first be done by calling the function convSym() before calling this function.

REFERENCE

Golub, G.H and Van Loan, C.F., *Matrix Computations*, 2nd ed., Johns Hopkins, 1989

■ posInv

FUNCTION

u = posInv(*S*, n)

PURPOSE

Compute the inverse of an n by n symmetric positive definite matrix S, stored in compact form

INPUT

S (Real Vector): the n(n+1)/2 - dimensional vector which stores the lower triangular part of the n by n symmetric matrix S in a column by column sequence (compact form).

n (Integer Scalar): the row and column dimension of S

OUTPUT

u (Real Vector): the n(n+1)/2 - dimensional vector which contains the lower triangular part of the computed inverse, S^{-1}, in a column by column sequence (compact form)

EXAMPLES

```
//  An example for: posInv(S, n)

//  Compute the posInv(A,n) for a 4  by 4 positive definite
// matrix A stored in its compact form
// (i.e., a 10-dimensional vector):
```

```
A = {1; 1; 1; 1; 2; 2; 2; 3; 3; 4};
u = posInv(A,4);
//  Results:
//  u:  10 rows
//                          2
//                         -1
//                          0
//                          0
//                          2
//                         -1
//                          0
//                          2
//                         -1
//                          1
```

ALGORITHM AND COMMENTS

If the symmetric matrix S is not in its compact form, a conversion must first be done by calling the function convSym() before calling this function.

REFERENCE

Golub, G.H and Van Loan, C.F., *Matrix Computations*, 2nd ed., Johns Hopkins, 1989

■ posSolve

FUNCTION

xVector = posSolve(*A*, bVector)

PURPOSE

Solve the symmetric positive definite linear system Ax = b

INPUT

A (Real Vector): the n(n+1)/2 - dimensional vector which stores the lower triangular portion of the n by n symmetric positive definite matrix A in a column by column sequence

bVector (Real Vector): the right hand side n-dimensional vector of the positive definite system

OUTPUT

xVector (Real Vector): the computed n-dimensional solution vector of Ax = b

EXAMPLES

```
//  An example for:  posSolve(A,  b)

//  Compute posSolve(A,bA) for a positive definite linear
// system where A is a 4 by 4 matrix stored in its compact
```

```
// form (i.e., a
// 10-dimensional vector):

A = {1; 1; 1; 1; 2; 2; 2; 3; 3; 4};
bA = {10; 19; 26; 30};
xVector = posSolve(A, bA);

//  Results:
//  xVector:  4 rows
//                        1
//                        2
//                        3
//                        4
```

SEE ALSO

cho(A, n); posBkSv(L, b)

ALGORITHM AND COMMENTS

Computes the solution of a symmetric positive definite linear system by Cholesky decomposition followed by forward and backward substitution; for detailed information, see the Reference.

If the symmetric positive definite matrix A is not in its compact form, a conversion must first be done by using the function convSym() before calling the function posSolve.

REFERENCE

Golub, G.H. and VanLoan, C.F., *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989, pp. 141-145

■ pseudo

FUNCTION

P = pseudo(A, tol)

PURPOSE

Compute the pseudo (or generalized) inverse of a m by n real matrix A with $m \geq n$

INPUT

A (Real Matrix): an m x n real matrix

tol (Real Scalar): a real nonnegative number used for determining the rank of A

OUTPUT

P (Real Matrix) : the n by m real matrix which is the pseudo inverse of A of the desired rank

EXAMPLES

```
//  An example for: pseudo(A, tol)

//  Compute pseudo(A) for a real m by n matrix with m = 4
// and n = 3 and tolerance = 0.0

A = { 5,       1.0e-6,    1;
      6,       0.999999,  1;
      7,       2.00001,   1;
      8,       2.9999,    1};
P = pseudo(A, 0.0);

//  Results:
//  P:  3 rows, 4 columns
//   5322.31975981973  -1809.26632230339  -12349.4266348524
//     8836.37319733606

//  -5322.77518485512   1809.21915150261  12349.8872515601
//     -8836.33121820764

//  -26610.7854239872   9046.693075149      61746.9701216636
//     -44181.8777728254
```

ALGORITHM AND COMMENTS

If A is a n by m (with n < m) matrix, this function will return an error message to indicate that there is a dimension problem. The user should use the transpose of A in this case.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*, Adam Hilger Ltd., Bristol, 1979, pp. 19-20

■ QRD

FUNCTION

[Q, R] = QRD(A)

PURPOSE

Compute the QR decomposition of a given matrix A using the Householder reflections

INPUT

A (Real Matrix): a given m by n matrix

OUTPUT

Q (Real Matrix): the m by m orthogonal matrix in A = QR

R (Real Matrix): the m by n upper triangular matrix in A = QR

EXAMPLE

```
//  An example for: QRD(A)
//  Compute the QRD(A) for a 3 by 3  real matrix A:

A = {2, 10,  8;
   1,  4, -2;
   0,  2,  3};
[QA, RA] = QRD(A);

//  Results :
//  QA:  3 rows, 3 columns
// -0.894427190999916    0.182574185835055   -0.408248290463863
// -0.447213595499958   -0.365148371670111    0.816496580927726
//  0                    0.912870929175277    0.408248290463863
//
//  RA:  3 rows, 3 columns
// -2.23606797749979   -10.733126291999      -6.26099033699941
//  0                    2.19089023002066     4.9295030175465
//  0                    0                   -3.67423461417477
```

SEE ALSO

GQRD(), fastGQRD(), mGS()

ALGORITHM AND COMMENTS

This routine returns the matrices Q and R of the decomposition A = QR. More specifically, Q is an orthogonal m by m matrix and R is an upper triangular m by n matrix.

The algorithm consists of the following loop: at the ith step we update the m by m matrix H using H = h(i)H such that:

$$HA = R(i)$$

has zeroes under the main diagonal in 1, ... , i columns. Suppose that we have done this for i-1. Then, if R(i-1) has the ith column $<\#,v>^t$, and we are looking for A:

$$h(i) = \begin{bmatrix} E & 0 \\ 0 & h_i \end{bmatrix}$$

$$h(i)HA = h(i)R(i-1) = R(i)$$

dim(E)= i-1, dim(h_i) = (m-i+1)(m-i+1), and E is the identity matrix. Now in order that the ith column of R(i)

have zeroes under the main diagonal, we have to find h such that:

$$h(i)\,v = \lambda_i e_1, \qquad e_1 = < 1, 0, \ldots 0 >^t$$

This is the purpose of Householder transformation. In the beginning of the loop
H = E and at the end it is equal to the product of n Householder transformations of different dimensions (in block diagonal matrices). Setting $Q = H^t$, R = R(n) then yield the required decomposition. Some special handling is taken for the situation where || v || is equal or approximately equal to zero, which indicates rank degeneracy.

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, Ch.5. pp. 211-216

■ schurD

FUNCTION

[Q, R] = schurD(A)

PURPOSE

Compute the Schur decomposition of a real matrix with all real eigenvalues

INPUT

A (Real Matrix): a given m by n matrix

OUTPUT

Q (Real Matrix): an m by m orthogonal matrix

R (Real Matrix): an m by n upper triangular matrix

Q and R satisfy $Q^tRQ = A$

EXAMPLES

```
//  An example for: schurD(A)
//  Compute schurD(A) for a  3 by 3 real matrix A :

A = {2, 10,  8;
   1,  4, -2;
   8,  2,  3};
[QA, RA] = schurD(A);

//  Results:
//  QA:  3 rows, 3 columns
// -0.615526846340926     0.185357453161812      -0.766008691850792
// -0.477102857201859    -0.861253602585091       0.174971699666047
```

```
// -0.627295436842495     0.473164714017825       0.618559284406122
//
//  RA:  3 rows, 3 columns
//  8.94444934221549      8.07621530702508       -3.48315672580186
//  0                     6.69843076642959       -4.43197028782153
//  0                     0                      -6.64288010864508
```

SEE ALSO

QRD(), GQRD(), fastGQRD(), mGS()

ALGORITHM AND COMMENTS

We have a decomposition of the matrix A where there is an orthogonal m by m matrix Q, an upper triangular m by n matrix R, and Q and R satisfy

$$Q^t R Q = A$$

The algorithm is implemented as a loop operating inductively on the dimension of the matrix. On the first step, we let R[1] = A and Q = E, where E is the identity matrix. At step k, matrix R[k] is multiplied by Q[k],

$$R[k] = \begin{bmatrix} r & a_{12} \\ 0 & a_{22} \end{bmatrix}, \qquad Q[k] = \begin{bmatrix} E & 0 \\ 0 & q_k \end{bmatrix}$$

where dim(E) = (k-1)(k-1), q_k is the Householder transformation (of dimension m - k + 1), and the matrix:

$$q_k^t a_{22} q_k, \qquad (q_k^t = q_k)$$

has $v = (\lambda, 0, \ldots, 0)^t$ as its first column. Now let v be the λ-eigenvector of a_{22}, and q_k to be connected with v (see the Householder algorithm description). Then Q has to be changed into $Q[k]^t Q$, R[k+1] = Q[k]R[k]Q[k]. Since:

$$q_k a_{22} q_k (e_1) = q_k a_{22} (v) = \lambda q_k v = \lambda e_1$$

the induction condition is satisfied:

$$Q[k]^t R[k] Q[k] = \begin{bmatrix} r & a_{12} q \\ 0 & q_k a_{22} q_k \end{bmatrix}$$

All we have to note now is that equation $A = Q^t R[k+1] Q$ is the invariant of the loop.

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, Ch.5. pp. 211-216

■ SVD

FUNCTION

[U, S, V] = SVD(A)

PURPOSE

Perform the singular value decomposition on a given m by n matrix A with m ≥ n (i.e., $A=USV^t$)

INPUT

A (Real Matrix): a given m by n matrix

OUTPUT

U (Real Matrix): a m by n matrix U which contains n orthogonal columns. U overwrites A

S (Real Vector): a n-dimensional vector which contains all the singular values of the matrix S

V (Real Matrix): a n by n orthogonal matrix

EXAMPLES

```
//  An example for: SVD(A)
//  Compute SVD(A) for a real m by n matrix with m = 4 and n = 3:
//
A = {5,       1.0e-6,    1;
     6,       0.999999,  1;
     7,       2.00001,   1;
     8,       2.9999,    1};
[U, s, U] = SVD(A);

//  Results:
//  s:  3 rows
//           13.7529874373082
//            1.68960781224662
//            1.18853233030117e-05
//

//  U:  4 rows, 3 columns
//  -0.358943042062186   0.755762452356787   -0.328687304919674
//  -0.446526454047557   0.317193591607159    0.111740607768327
//  -0.5341100983257    -0.12138257880912     0.762674479694079
//  -0.621691580494195  -0.559890713097346   -0.545716345901029
//
//  V:  3rows, 3columns
```

```
//  -0.958786396551136   0.209024870993394   -0.192450640671011
//  -0.245747659695571  -0.950036144396354    0.192456260211665
//  -0.142606919687081   0.231818738773865    0.96224910434391
```

ALGORITHM AND COMMENTS

Important Note: If A is a n by m (with n < m) matrix, this function will return an error message to indicate that there is a dimension problem. The user should use the transpose of A in this case.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers, Linear Algebra and Function Minimisation*, Adam Hilger Ltd., Bristol, 1979, p. 26

■ SVDS

FUNCTION

S = SVDS(A)

PURPOSE

Compute all the singular values of a given m by n matrix with A, with m ≥ n

INPUT

A (Real Matrix): a given m by n matrix

OUTPUT

S (Real Vector): a n-dimensional vector which contains all the singular values

EXAMPLES

```
//  An example for: SVDS(A)
//
//  Compute SVDS(A) for a real m by n matrix with m = 4
//     and n = 3:

A = {5,        1.0e-6,     1;
     6,        0.999999,   1;
     7,        2.00001,    1;
     8,        2.9999,     1};
s = SVDS(A);

//  Results:
//  s:  3 rows
//       13.7529874373082
//        1.68960781224662
//        1.18853233030117e-05
```

ALGORITHM AND COMMENTS

Important Note: If A is an n by m (with n < m) matrix, this function will return an error message to indicate that there is a dimension problem. The user should use the transpose of A in this case.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*, Adam Hilger Ltd., Bristol, 1979, p. 26

■ symBkSv

FUNCTION

xVector = symBkSv(LDU, pivot, bVector)

PURPOSE

Solve the symmetric linear system:

$$Sx = b$$

by forward substitution, backward substitution and a band system solver in compact form when the LDL^t decomposition of S is available

INPUT

LDU (Real or Complex Vector): the n(n+1)/2 - dimensional vector which contains the lower triangular matrix and the tridiagonal matrix D of the LDL^t decomposition of the n by n symmetric matrix S in a column by column sequence. Normally LDU results from calling the function symLDU(*S*,n) with *S* the compact form of S

pivot (Integer Vector): the integer n-dimensional vector which contains the pivot information of the LDL^t decomposition

bVector (Real or Complex Vector): the right hand side n-dimensional vector of the symmetric system

OUTPUT

xVector (Real or Complex Vector): the computed n-dimensional solution vector of Sx = b

EXAMPLES

```
//  Examples for: symBkSv(LDU, pivot, bVector)
//  Compute symBkSv(A, pivotA, bVector) for a  4 by 4 real
// symmetric matrix A stored in its compact form
// (i.e., a 10-dimensional vector):

A = {1; 0.5; 0.5; 0.66; 0.33; 0.035; 0.92; 0.0255;  -0.00456;
    0.0005552};
pivotA = { 1; 2; 4; 3};
bVector = {3.99; 2.71; 2.098;1.723};
xVector = symBkSv(A, pivotA, bVector);
```

```
//  Results:
//  xVector:  4 rows
//                        1
//                        2
//                        3
//                        4

//  Compute symLDU(B, pivotB, bVector) for a 3 by 3 complex
// symmetric matrix B stored in its compact form
// (i.e., a 6-dimensional vector):

B = {(1, 1); (0.5,1); (0.932, 0.136);(0.33,2);
   (0.21444 , 0.09112); (0.05153424, 0.98791232)};
pivotB = {1; 2; 3};
bVector = {(-2.711, 9.59); (-8.899, 14.01); (-12.427, 16.83)};
zVector = symBkSv(B, pivotB, bVector);

//  Results:
//  zVector:  3 rows
//             1.1 +1i
//             2.2 +2i
//             3.3 +3i
```

SEE ALSO

symLDU(*S*,n), symSolve(*S*,b)

ALGORITHM AND COMMENTS

The LDL^t decomposition of S, normally resulting from calling the function symLDU, satisfies:

$$PSP^t = LDL^t$$

with P a permutation matrix, L a lower triangular matrix and D a tridiagonal matrix as described in the algorithm description of the function symLDU. The solution x, for the symmetric system Sx = b, is computed in the following manner :

Step 1: Compute c = Pb.

Step 2: Solve Lz = c by forward substitution.

Step 3: Solve tridiagonal system Dw = z using Gaussian elimination with partial pivoting in a compact band matrix form (see the function band).

Step 4: Solve $L^ty = w$ by backward substitution.

Step 5: Compute $x = P^ty$.

When S is singular, it will be detected in Step 3 when solving the tridiagonal system Dw = z, and an error message will be returned.

In the normal computation of solving a single symmetric system Sx = b, with S already in its compact form,

we can conveniently call the function symSolve(), which is equivalent to calling the function symLDU followed by symBkSv, to obtain the desired solution. The advantage of using symBkSv appears when there is a need to sequentially solve $Sx_1 = b_1$, ... , $Sx_k = b_k$ for some k > 1. In such cases, symBkSv() can be used repeatedly (k times) once symLDU() has been called.

REFERENCE

Golub, G.H. and VanLoan, C.F., *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989, pp. 159-160

■ symDet

FUNCTION

d = symDet(*S*, n)

PURPOSE

Compute the determinant of a given symmetric indefinite matrix S stored in a compact storage mode

INPUT

S (Real or Complex Vector): the n(n+1)/2 - dimensional vector which stores the lower triangular part of the n by n symmetric matrix, S, in a column by column sequence (compact form)

n (Integer Scalar): the row and column dimension of S

OUTPUT

d (Real Scalar): the computed determinant of the symmetric indefinite matrix S

EXAMPLES

```
//  Examples for: symDet(S, n)
//  Compute symDet(A,n) for a 4 by 4 real symmetric matrix
//  A stored in its compact form (i.e., a 10-dimensional
//  vector):

A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
d = symDet(A, 4);

//  Result :      d:       -1.211e-06

//  Compute symDet(B,n) for a 3 by 3 complex symmetric matrix
//  B stored in its compact form
//   (i.e., a 6-dimensional vector):

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
d1 = symDet(B,3);

//  Result :      d1:    -1.359937 -0.9171i
```

ALGORITHM AND COMMENTS

If the symmetric matrix S is not in its compact form, a conversion must be done by calling function convSym() before running the function symDet.

REFERENCE

Golub, G.H. and VanLoan, C.F., *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989

■ symInv

FUNCTION

B = symInv(*S*, n)

PURPOSE

Compute the inverse of a given nonsingular symmetric matrix S stored in a compact storage mode

INPUT

S (Real or Complex Vector): the n(n+1)/2 - dimensional vector which stores the lower triangular part of the n by n symmetric matrix S in a column by column sequence (compact form)

n (Integer Scalar): the row and column dimension of S

OUTPUT

B (Real or Complex Vector): the computed inverse of the symmetric matrix S in compact form

EXAMPLES

```
//  Examples for: symInv(S, n)
//  Compute  symInv(A,n) for a 4 by 4  real symmetric matrix
//  A stored in its compact form
//   (i.e., a 10-dimensional vector):

A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};

C = symInv(A, 4);

//  Results :
//  C:  10 rows
//        2.36994219653179
//       14.4508670520231
//      -65.8959537572254
//       52.6011560693642
//     -159.62014863749
//      417.01073492981
//     -289.017341040462
```

```
//     -672.997522708505
//      317.919075144509
//      -52.0231213872832

//  Compute symInv(B,n) for a 3 by 3 complex symmetric matrix
//  B stored in its compact form
//   (i.e., a 6-dimensional vector):

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
D = symInv(B,3);

//  Results :
//  D:  6 rows
//        0.876210480208629  -0.87766759151294i
//       -0.22732907002498   +0.734213048192607i
//       -0.096763011973531  -0.1259313054347 92i
//        0.099070668976988  -1.934543813808 14i
//        0.0215348902816923 +1.029643543872i
//        0.0116703317970013 -0.9858565957768 85i
```

ALGORITHM AND COMMENTS

If the symmetric matrix S is not in its compact form, a conversion must be done by calling function convSym() before running the function symLDU.

REFERENCE

Golub, G.H. and VanLoan, C.F., *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989

■ symLDU

FUNCTION

[LDU, pivot] = symLDU(*S*, n)

PURPOSE

Perform the LDL^t decomposition of a n by n symmetric matrix S stored in compact form

INPUT

S (Real or Complex Vector): the n(n+1)/2 - dimensional vector which stores the lower triangular portion of the n by n symmetric matrix S in a column by column sequence

n (Integer Scalar): the row and column dimension of S

OUTPUT

LDU (Real or Complex Vector): the n(n+1)/2 - dimensional vector which contains the resultant lower triangular matrix and the tridiagonal matrix D of the LDL^t decomposition in a column by column sequence (com-

pact form)

pivot (Integer Vector): the integer n-dimensional vector which contains the pivoting information of the LDL^t decomposition

EXAMPLES

```
//  Examples for: symLDU(S, n)

//  Compute symLDU(A,n) for a  4  by 4  real symmetric matrix
//  A stored in its compact form
// (i.e., a 10-dimensional vector):

A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
[lduA, pA] = symLDU(A, 4);


//  Results:
//  lduA:  10 rows
//                      1
//                      0.5
//                      0.5
//                      0.66
//                      0.33
//                      0.035
//                      0.92
//                      0.0255
//                     -0.00456
//                      0.0005552
//
//  pA:  4 rows
//     1
//     2
//     4
//     3

//  Compute symLDU(B,n) for a 3 by 3 complex symmetric matrix
//  B stored in its compact form
// (i.e., a 6-dimensional vector):

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
[lduB, pB] = symLDU(B,3);

//  Results:
//  lduB:  6 rows
//             1          +1i
//             0.5        +1i
//             0.932      +0.136i
//             0.33       +2i
//             0.21444    +0.09112i
//             0.05153424 +0.98791232i
```

```
//
//  pB:  3 rows
//     1
//     2
//     3
```

SEE ALSO

symBkSv(LDU, pivot, b)

ALGORITHM AND COMMENTS

The stable Aasen's method with (symmetric partial) pivoting is used to perform the LDL^t decomposition for a n by n symmetric matrix such that

$$PSP^t = LDL^t$$

where P is a n by n permutation matrix which corresponds to the pivoting sequence, L is a n by n lower triangular matrix in the form:

$$\begin{bmatrix} 1 & & & & & \\ 0 & 1 & & & & \\ 0 & l_{32} & 1 & & & \\ 0 & l_{42} & & 1 & & \\ \cdots & \cdots & \cdots & \cdots & 1 & \\ 0 & l_{n2} & l_{n3} & \cdots & l_{n,n-1} & 1 \end{bmatrix}$$

and

$$D = \begin{bmatrix} \alpha_1 & \beta_1 & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ \cdots & \cdots & \cdots & \cdots & & \\ \cdots & \cdots & \cdots & \cdots & \beta_{n-1} & \\ \cdots & \cdots & \cdots & \beta_{n-1} & \alpha_n & \end{bmatrix}$$

is a n by n symmetric tridiagonal matrix. Aasen's method requires only $O(n^3/3)$ arithmetic float operations in a total of n-2 symmetric Gaussian transformations which eliminate the ith off-subdiagonal elements (in both rows and columns) of A. The pivot in the ith Gaussian transformation is chosen to be the largest element in magnitude below the subdiagonal of the ith column.

Similar to the relationship between the LU decomposition and Gaussian elimination for a general square matrix, the working formulas for establishing the elements of L and D are not difficult to derive. For detailed information about generating L and D, see the reference below.

In the practical implementation of Aasen's method, a compact storage scheme is used. We stored the resultant matrices L and D in a n(n+1)/2 - dimensional vector LDU which contains the nonzero elements of the lower triangular matrix

$$\begin{bmatrix} \alpha_1 & & & & & \\ \beta_1 & \alpha_2 & & & & \\ l_{32} & \beta_2 & \alpha_3 & & & \\ l_{42} & l_{43} & \cdots & \cdots & & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \\ l_{n2} & l_{n3} & \cdots & \cdots & \beta_{n-1} & \alpha_n \end{bmatrix}$$

in a column by column sequence. The permutation matrix is stored economically by specifying the row (and column) interchange sequence in an n-dimensional integer vector pivot.

If the symmetric matrix is not in its compact form, a conversion must first be done using the function convSym() before calling this routine.

REFERENCE

Golub, G.H. and VanLoan, C.F., *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989, pp. 164-166

■ symPermu

FUNCTION

u = symPermu(*A*, n, r, p, q)

PURPOSE

Perform the complete pivoting (interchanging row p and row q, column p and column q) on the (n-r+1) by (n-r+1) submatrix starting at the rth diagonal of the given n by n symmetric matrix *A* in a compact storage mode

INPUT

A (Real or Complex Vector): a n(n+1)/2 - dimensional vector which stores the lower triangular part of A in a column by column sequence (compact form)

n (Integer Scalar): the row (and column) dimension of A

r (Integer Scalar): the index of the first row and column of the submatrix of A where the permutation is performed

p (Integer Scalar): the row and column index of the matrix A where the interchange (with row/column q) is desired

q (Integer Scalar): the row and column index of the matrix A where the interchange (with row/column p) is desired

OUTPUT

u (Real or Complex Vector): the n(n+1)/2 - dimensional vector which stores the lower triangular portion of A after permutation

EXAMPLES

```
//  Examples for: symPermu(A, n, r, p, q)

//  Compute symPermu(A,n,q,r,s) for a  4  by 4  real symmetric
//   matrix A stored in its compact form (i.e., a 10-dimensional
//   vector) with q = 2, r= 3, s=4:

A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
u = symPermu(A,4,2,3,4);

//  Results:
//  u:  10 rows
//           1
//           0.5
//           0.33
//           0.25
//           0.33
//           0.2
//           0.25
//           0.143
//           0.167
//           0.2

//  Compute  symPermu(B,n,q,r,s) for a 3 by 3 complex symmetric
//   matrix B stored in its compact form (i.e., a 6-dimensional
//  vector) with q = 1, r=1, s= 3:

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
v = symPermu(B,3,1,1,3);
//  Results:
//  v:  6 rows
//                     0.2  +3i
//                     0.25 +2i
//                     0.33 +1i
//                     0.33 +2i
//                     0.5  +1i
//                     1    +1i
```

ALGORITHM AND COMMENTS

If the symmetric matrix A is not in its compact form, a conversion must first be done by calling the function convSym() before calling this function.

■ symSolve

FUNCTION

xVector = symSolve(*S*, b)

PURPOSE

Solve the symmetric linear system Sx = b in compact form.

INPUT

S (Real or Complex Vector): the n(n+1)/2 - dimensional vector which stores the lower triangular portion of the n by n symmetric matrix S in a column by column sequence

b (Real or Complex Vector): the right hand side n-dimensional vector of the symmetric system

OUTPUT

xVector (Real or Complex Vector): the computed n-dimensional solution vector of Sx = b

EXAMPLES

```
//  Examples for: symSolve(S,  b)

//  Compute symSolve(A,bA) for a real symmetric linear system
//   where A is a 4 by 4  matrix stored in its compact form
//  (i.e., a 10-dimensional vector):

A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
bA = {3.99; 2.71; 2.098;1.723};

xVector = symSolve(A, bA);

//  Results:
//   xVector:  4 rows
//                        1
//                        2
//                        3
//                        4

//  Compute  symSolve(B,bB) for a complex symmetric linear
//  system where B is a 3 by 3 matrix stored in its compact form
//  (i.e., a 6-dimensional vector):

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
bB = {(-2.711, 9.59); (-8.899, 14.01); (-12.427, 16.83)};
zVector = symSolve(B, bB);

//  Results:
// zVector:  3 rows
//             1.1 +1i
```

```
//              2.2 +2i
//              3.3 +3i
```

SEE ALSO

symLDU(*S*, n); symBkSv(LDU, pivot, b)

ALGORITHM AND COMMENTS

Computes the solution to a symmetric linear system by decomposing the matrix into LDL^t form performing forward substitution, using a band system solver, and then backward substitution; see algorithm descriptions of symLDU, symBkSv and the Reference.

If the symmetric matrix is not in its compact form, a conversion must first be made by using the function convSym() before calling this routine.

REFERENCE

Golub, G.H. and VanLoan, C.F., *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989, pp. 159-166

■ toepSolve

FUNCTION

xVector = toepSolve(v, bVector)

PURPOSE

Efficiently solve the n-dimensional Toeplitz system of equations Tx = b using the Rybicki's method with T generated by the given (2n-1)-dimensional vector v

INPUT

v (Real Vector): the (2n-1)-dimensional vector that contains the desired elements for any row (or column) of the n by n Toeplitz matrix, T (see the definition for the output matrix T of function Toeplitz)

bVector (Real Vector): the right hand side n-dimensional vector of the given Toeplitz system

OUTPUT

xVector (Real Vector): the n-dimensional solution vector of the given Toeplitz system

EXAMPLES

```
//  An example for: toepSolve(v, b)

//  Compute toepSolve(u,b) for a Toeplitz linear linear
// system with A a 4 by 4 Toeplitz matrix generated
// by a 7-dimensional vector u:

u = {1; 2; -3; 4; 5; 6; 7};
```

```
b = {8; 12; 16; 50};
xVector = toepSolve(U,b);

//  Results:
//  xVector:  4 rows
//                        1
//                        2
//                        3
//                        4
```

ALGORITHM AND COMMENTS

Rybicki's method is an (n-1)-step iterative procedure which updates and solves $X_1, \ldots, X_n$ in a successive manner. In the ith iteration, the method updates the (already) computed $X_1, \ldots, X_i$ and establishes the value for X_{i+1} by solving the linear system with the coefficient matrix formed by the principal minors of T of order i+1 and the first i+1 components of the right hand side (i+1)-vector using a set of recursion formulas. The algorithm can be briefly summarized as follows.

(1) Start with

$$X_1^{(1)} = \frac{b_1}{A_n}, \qquad G_1^{(1)} = \frac{A_{n-1}}{A_n}, \qquad H_1^{(1)} = \frac{A_{n+1}}{A_n}$$

(2) For i = 1, ... , n-1,

$$\text{Compute} \quad X_{i+1}^{(i+1)} = \frac{\sum_{j=1}^{i} A_{n+i+1-j} X_j^{(i)} - b_{i+1}}{\sum_{j=1}^{i} A_{n+i+1-j} G_{i+1-j}^{(i)} - A_n}$$

$$\text{Update} \quad X_{i+1-j}^{(i+1)} = X_{i+1-j}^{(i)} - G_j^{(i)} X_{i+1}^{(i+1)}, \qquad \text{for} \quad 1 \le j \le i$$

$$\text{Compute} \quad H_{i+1}^{(i+1)} = \frac{\sum_{j=1}^{i} A_{n+i+1-j} H_j^{(i)} - A_{n+i+1}}{\sum_{j=1}^{i} A_{n+i+1-j} G_{i+1-j}^{(i)} - A_n} \qquad \text{and}$$

$$G_{i+1}^{(i+1)} = \frac{\sum_{j=1}^{i} A_{n+i+1-j} G_j^{(i)} - A_{n+i+1}}{\sum_{j=1}^{i} A_{n+i+1-j} H_{i+1-j}^{(i)} - A_n}$$

Update $G_j^{(i+1)} = G_j^{(i)} - G_{i+1}^{(i+1)} H_{i+1-j}^{(i)}$, and

$H_{i+1-j}^{(i+1)} = H_{i+1-j}^{(i)} - H_{i+1}^{(i+1)} G_j^{(i)}$, for $1 \le j \le i$

The solution of the Toeplitz system is then

$$X = (X_1^n, \ldots, X_n^n)$$

Comment :

The method does not require any pivoting and takes only $O(n^2)$ arithmetic operations to obtain the solution. The drawback of the method is that it will fail during the iteration if any of the diagonal principal minors of the Toeplitz matrix vanishes. If so, the given system must be solved by some more general but slower method, for example, LU decomposition with pivoting.

REFERENCE

Press, W.H., Flannery, B.P, Teukolsky, S.A. and Vetterling, W.T., *Numerical Recipes in C*, Cambridge University, 1988, p. 54

Oppenheim, A.V. and Schafer, R.W., *Digital Signal Processing*, Prentice-Hall, 1975, p. 232.

■ uTriDet

FUNCTION

d = uTriDet(*U*, n)

PURPOSE

Compute the determinant of an upper triangular matrix U

INPUT

U (Real or Complex Vector): a n(n+1)/2 - dimensional vector which stores the upper triangular matrix U in a column by column sequence (compact form)

n (Integer Scalar): the row and column dimension of U

OUTPUT

d (Real Scalar): the computed determinant of the upper triangular matrix U

EXAMPLES

```
//  Examples for: uTriDet(U, n)

//  Compute uTriDet(A,n) for a 4 by 4 real upper triangular
//   matrix A stored in its compact form
// (i.e., a 10-dimensional  vector):

A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
dA = uTriDet(A,4);

//  Result :
//         dA:         0.0117975

//  Compute uTriDet(B,n) for a 3 by 3 complex upper
// triangular matrix B stored in its compact form
// (i.e., a 6-dimensional vector):

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
dB = uTriDet(B, 3);

//  Result :
//         dB:          -4.124 -1.744i
```

ALGORITHM AND COMMENTS

If the triangular matrix U is not in the vector form, a conversion must be done by first calling the routine convUTriag() before calling this function.

■ uTriInv

FUNCTION

v = uTriInv(*U*, n)

PURPOSE

Compute the inverse of a nonsingular upper triangular matrix U

INPUT

U (Real or Complex Vector): a n(n+1)/2 - dimensional vector which stores the upper triangular matrix U in a column by column sequence (compact form)

n (Integer Scalar): the row and column dimension of U

OUTPUT

v (Real or Complex Vector): the computed inverse of U in compact form

EXAMPLES

```
//  Examples for: uTriInv(U, n)

//  Compute uTriInv(A,n) for a 4  by 4 real upper triangular
// matrix A stored in its compact form
// (i.e., a 10-dimensional  vector):

A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
C = uTriInv(A,4);

/  Results :
//  C:  10 rows
//         1
//        -1.51515151515152
//         3.03030303030303
//         1
//        -4
//         4
//        -0.447340538249629
//         0.433142614960797
//        -4.67132867132867
//         6.99300699300699

//  Compute uTriInv(B,n) for a 3 by 3 complex upper triangular
//  matrix B stored in its compact form (i.e., a 6-dimensional
//  vector):

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
D = uTriInv(B, 3);
//  Results:
//  D:  6 rows
//        0.5                -0.5i
//       -0.448642799170349  +0.601947876273785i
//        0.297592208494905  -0.901794571196681i
//       -0.0259284094817714 -0.0520807114121704i
//       -0.164248314322493  +0.615045843884197i
//        0.0221238938053097 -0.331858407079646i
```

ALGORITHM AND COMMENTS

If the triangular matrix U is not in its vector form, a conversion must first be done by calling the routine convUTriag() before calling this function.

■ uTriSolve

FUNCTION

xVector = uTriSolve(*U*, bVector)

PURPOSE

Compute the solution of an upper triangular system of linear equations:

$$Ux = b$$

where U is stored in a compact storage mode

INPUT

U (Real or Complex Vector): a n(n+1)/2 - dimensional vector which stores the upper triangular matrix U in a column by column sequence (compact form)

bVector (Real or Complex Vector): the n-dimensional vector of the right side of the linear system of equations

OUTPUT

xVector (Real or Complex Vector): the n-dimensional solution vector of Ux = b

EXAMPLES

```
//  Examples for: uTriSolve(U, b)

//  Compute  uTriSolve(A, bA) for a real upper triangular
//  linear system where A is a 4 by 4 matrix stored in its
//  compact form (i.e., a 10-dimensional vector):

A = {1; 0.5; 0.33; 0.25; 0.33; 0.25; 0.2; 0.2; 0.167; 0.143};
bA = {3.55 ;  2.45; 1.418; 0.572};
xVector = uTriSolve(A,bA);

//  Results :
//  xVector:  4 rows
//                        1
//                        2
//                        3
//                        4

//  Compute uTriSolve(B,bB) for a complex upper triangular
//  linear system where B is a 3 by 3 matrix stored in its
//  compact form (i.e., a 6-dimensional vector):

B = {(1, 1); (0.5, 1); (0.33,1); (0.33,2); (0.25,2); (0.2,3)};
bB = {(-6.91, 12.289); (-7.39 , 9.551); (-9.3, 9.66)};
zVector = uTriSolve(B,bB);
```

```
// Results:
//  zVector:  3 rows
//                        1 +1.1i
//                        2 +2.2i
//                        3 +3.3i
```

ALGORITHM AND COMMENTS

If the triangular matrix U is not in its vector form, a conversion must first be done by calling the routine convUTriag() before calling this function.

■ vanSolve

FUNCTION

xVector = vanSolve(v, bVector)

PURPOSE

Efficiently solve the n-dimensional Vandermonde system of equations Ax=b with A generated by the given n-dimensional vector v

INPUT

v (Real Vector): the n-dimensional vector containing the elements of any row (or column) of the n by n Vandermonde matrix A (see the definition for output matrix A of function Vandermonde)

bVector (Real Vector): the right hand side n-dimensional vector of the given Vandermonde system

OUTPUT

xVector (Real Vector): the n-dimensional solution vector of the given Vandermonde system

EXAMPLE

```
//  An example for: vanSolve(v, b)

//  Compute vanSolve(u,b) for a Vandermonde linear system
// Ax=b with A a 5 by 5 Vandermonde matrix  generated by the
// 5-dimensional vector u and b is a 5-dimensional vector:

u = {1; -2; 3; -4; 5};
b = {15; 15; 225; 435; 4425};
xVector = vanSolve(u,b);

//  Results :
//  xVector:  5 rows
//                        1
//                        2
```

```
//                    3
//                    4
//                    5
```

ALGORITHM AND COMMENTS

The solution of the Vandermonde system is $x = A^{-1}b = (LU)^{-1}b$ with the special simple form of the LU decomposition for A (or the UL decomposition of $(LU)^{-1}$). For detailed information on this decomposition, see the Reference. The method used here takes approximately $O(n^2)$ (roughly $5n^2/2$) arithmetic operations.

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, 1989, pp. 181-182

CHAPTER 16
MATRIX STRUCTURE FUNCTIONS

■ convLTriag

FUNCTION

u = convLTriag(A)

PURPOSE

To convert a given m by m lower triangular matrix A to a m(m+1)/2 - dimensional vector (for compact storage) in a column by column sequence, and vice versa

INPUT

A (Real, Complex Vector or Matrix) : a m by m lower triangular matrix or a m(m+1)/2 - dimensional vector

OUTPUT

u (Real, Complex Vector or Matrix): a m(m+1)/2 - dimensional vector v or a m by m lower triangular matrix A depending on whether the input is a lower triangular matrix or a vector storing a lower triangular matrix, respectively

EXAMPLES

```
//  Examples for: convLTriag(L)

//  Compute convLTriag(u) for a 10-dimensional real
//  vector u

u = { 1; 2; 3; 4; 5; 6; 7; 8; 9; 10};
U = convLTriag(u);
//  Results:
//  U:  4 rows, 4 columns
//      1    0     0     0
//      2    5     0     0
//      3    6     8     0
//      4    7     9    10

//  Compute convLTriag(B) for a 4 by 4 complex lower
//  triangular matrix B

B = { (1,1),  (0,0),  (0,0),  (0,0);
      (2,2),  (5,5),  (0,0),  (0,0);
      (3,3),  (6,6),  (8,8),  (0,0);
      (4,4),  (7,7),  (9,9),  (10,10)};
bVector = convLTriag(B);
```

```
//  Results:
//  bVector:  10 rows
//                          1 +1i
//                          2 +2i
//                          3 +3i
//                          4 +4i
//                          5 +5i
//                          6 +6i
//                          7 +7i
//                          8 +8i
//                          9 +9i
//                         10+10i
```

ALGORITHM AND COMMENTS

For a given m by m matrix A, the function convLTriag generates an m(m+1) - dimensional vector to store the lower triangular part of A in a column by column fashion i.e.,

$$v = (a_{11}, \ldots, a_{m,1}, a_{22}, \ldots, a_{m,2}, \ldots, a_{m,m})^t$$

■ convSym

FUNCTION

u = convSym(A)

PURPOSE

Convert a given m by m symmetric matrix A to a m(m+1)/2 - dimensional vector (for compact storage) in a column by column sequence, and vice versa

INPUT

A (Real, Complex Vector or Matrix): An m by m symmetric matrix or a m(m+1)/2 - dimensional vector

OUTPUT

u (Real, Complex Vector or Matrix): a m(m+1)/2 - dimensional vector v or a m by m symmetric matrix A depending on whether the input is a symmetric matrix or a vector storing a symmetric matrix, respectively

EXAMPLES

```
//  Examples for: convSym(S)

//  Compute convSym(u) for a 10-dimensional real
//  vector u
```

```
u = {1; 2; 3; 4; 5; 6; 7; 8; 9; 10};
U = convSym(U);

//  Results:
//  U:  4 rows, 4 columns
//      1    2     3     4
//      2    5     6     7
//      3    6     8     9
//      4    7     9    10

//  Compute convSym(B) for a 4 by 4 complex symmetric
//  matrix B

B = { (1,1),  (2,2),  (3,3),  (4,4);
      (2,2),  (5,5),  (6,6),  (7,7);
      (3,3),  (6,6),  (8,8),  (9,9);
      (4,4),  (7,7),  (9,9),  (10,10)};
bVector = convSym(B);

//  Results:
//  bVector:  10 rows
//                        1 +1i
//                        2 +2i
//                        3 +3i
//                        4 +4i
//                        5 +5i
//                        6 +6i
//                        7 +7i
//                        8 +8i
//                        9 +9i
//                       10+10i
```

ALGORITHM AND COMMENTS

For a given m by m symmetric matrix A, convSym(A) generates an m(m+1) - dimensional vector to store the upper triangular part of A in a column by column fashion, i.e.,

$$v = (a_{11}, a_{12}, a_{22}, a_{13}, \ldots, a_{33}, \ldots, a_{m,m})^t$$

■ convUTriag

FUNCTION

u = convUTriag(A)

PURPOSE

To convert a given m by m upper triangular matrix A to a m(m+1)/2 - dimensional vector (for compact storage) in a column by column sequence, and vice versa

INPUT

A (Real, Complex Vector or Matrix) : a m by m upper triangular matrix or a m(m+1)/2 - dimensional vector

OUTPUT

u (Real, Complex Vector or Matrix): a m(m+1)/2 - dimensional vector v or a m by m upper triangular matrix A depending on whether the input is a upper triangular matrix or a vector storing a upper triangular matrix, respectively

EXAMPLES

```
//  Examples for: convUTriag(U)

//  Compute convLTriag(u) for a 10-dimensional real
//  vector U

u = { 1; 2; 3; 4; 5; 6; 7; 8; 9; 10};
U = convUTriag(U);

//  Results:
//  U:  4 rows, 4 columns
//      1    2     4     7
//      0    3     5     8
//      0    0     6     9
//      0    0     0    10

//  Compute  convUTriag(A) for a 4 by 4 real upper
//  triangular matrix A

A = { 1,  2,  4,  7;
      0,  3,  5,  8;
      0,  0,  6,  9;
      0,  0,  0, 10};
aVector = convUTriag(A);
//  Results:
//  aVector:  10 rows
//                        1
//                        2
//                        3
//                        4
//                        5
//                        6
//                        7
//                        8
//                        9
//                       10
```

ALGORITHM AND COMMENTS

For a given m by m matrix A, convUTriag generates an m(m+1) - dimensional vector to store the upper triangular part of A in a column by column fashion i.e.,

$$v = (a_{11}, a_{12}, a_{22}, a_{13}, \ldots, a_{33}, \ldots, a_{m,m})^t$$

■ isDiagDom

FUNCTION

y = isDiagDom(A)

PURPOSE

Determine whether or not a n by n real square matrix A is diagonally dominant

INPUT

A (Real Matrix): a n by n real square matrix A

OUTPUT

y (Integer Scalar): A boolean result indicating whether or not the matrix is diagonally dominant; if true, then y = 1 = <TRUE>; if not, y = 0 = <FALSE>

EXAMPLES

```
//  An example for: isDiagDom(A)

//  Compute isDiagDom(A) for a 4 by 4 real matrix A

A = { -10,    2,     3,   -4;
        5,   26,    -7,   -8;
        1,   -3,   -11,    5;
       -2,   -4,    -4,   16};
y = isDiagDom(A);

//  Result :     y:  1
```

■ isNonSingSymIndef

FUNCTION

y = isNonSingSymIndef(A)

PURPOSE

Determine whether or not a given n by n square matrix is nonsingular, symmetric and indefinite

INPUT

A (Real Matrix): a n by n real matrix A

OUTPUT

y (Integer Scalar): A boolean result indicating whether or not the matrix is nonsingular, symmetric and indefinite; if true, then y = 1 = <TRUE>; if not, y = 0 = <FALSE>

EXAMPLE

```
//  An example for: isNonSingSymIndef(A)

//  Compute  isNonSingSymIndef(A) for a 3 by 3
//  symmetric matrix A (the three eigenvalues of A are
//  around -1.505, 0.802 and 1.570)

A= {    0.2,        0.333,        1;
        0.333,      1,           -1;
        1,         -1 ,          -0.333};
y = isNonSingSymIndef(A);

//  Result :      y:  1
```

■ isOrthogonal

FUNCTION

y = isOrthogonal(A)

PURPOSE

Determine if a given n by n real square matrix A is an orthogonal matrix

INPUT

A (Real Matrix): a n by n real matrix A

OUTPUT

y (Integer Scalar): A boolean result indicating whether or not the matrix is orthogonal; if true, then y = 1 = <TRUE>; if not, y = 0 = <FALSE>

EXAMPLE

```
//  An example for: isOrthogonal(A)

//  Compute  isOrthogonal(A) for a 4 by 4 real matrix A

A = { -1,   0,   0,   0;
       0,   0,    1,  0;
       0,  -1,    0,  0;
       0,   0,    0,  -1};
y = isOrthogonal(A);

//  Result :       y:  1
```

■ isSymmetric

FUNCTION

y = isSymmetric(A)

PURPOSE

Determine if a given n by n real square matrix A is symmetric

INPUT

A (Real Matrix): a n by n real matrix A

OUTPUT

y (Integer Scalar): A boolean result indicating whether or not the matrix is symmetric; if true, then y = 1 = <TRUE>; if not, y = 0 = <FALSE>

EXAMPLE

```
//   Examples for: isSymmetric(A)

//  Compute isSymmetric(A) for a 4 by 4 real symmetric
//  matrix A

A = { 1,  2,  3,  4;
      2,  5,  6,  7;
      3,  6,  8,  9;
      4,  7,  9, 10};
y = isSymmetric(A);
```

```
//  Result :
//               y:  1

//  Compute isSymmetric(B) for a 4 by 4 complex
//  non-symmetric matrix B

B = { (1,1), (2,-2), (-3,3),  (4,4);
      (2,2),  (5,5),  (6,6),  (7,7);
      (3,3),  (6,6),  (8,8),  (9,9);
      (4,4),  (7,7),  (9,9),  (10,10)};
z = isSymmetric(B);

//  Result :
//               z:  0
```

■ isSymNegDef

FUNCTION

y = isSymNegDef(A)

PURPOSE

Determine whether a n by n real square matrix A is symmetric and negative definite

INPUT

A (Real Matrix): a given n by n matrix A

OUTPUT

y (Integer Scalar): A boolean result indicating whether or not the matrix is symmetric and negative definite; if true, then y = 1 = <TRUE>; if not, y = 0 = <FALSE>

EXAMPLE

```
//  An example for: isSymNegDef(A)

//  Compute isSymNegDef(A) for a 4 by 4 real matrix A

A = {-1,  2,  -1,  -1;
       2,  -5,  3,   1;
      -1,  3,  -3,   1;
      -1,  1,   1,  -4};
y = isSymNegDef(A);

//  Result :     y:  1
```

■ isSymPosDef

FUNCTION

y = isSymPosDef(A)

PURPOSE

Determine the symmetry and positive definitiveness of a n by n real square matrix A

INPUT

A (Real Matrix): a given n by n real matrix A

OUTPUT

y (Integer Scalar): A boolean result indicating whether or not the matrix is symmetric and positive definite; if true, then y = 1 = <TRUE>; if not, y = 0 = <FALSE>

EXAMPLE

```
//   An example for: isSymPosDef(A)

//  Compute  isSymPosDef(A) for a 4 by 4 real matrix A

A = {1, 1, 1, 1;
     1, 2, 2, 2;
     1, 2, 3, 3;
     1, 2, 3, 4};
y = isSymPosDef(A);

//  Result :
//               y:  1
```

■ isSymSemiNegDef

FUNCTION

y = isSymSemiNegDef(A)

PURPOSE

Determine whether or not a n by n real square matrix A is symmetric and semi-negative definite

INPUT

A (Real Matrix): a given n by n real matrix A

OUTPUT

y (Integer Scalar): A boolean result indicating whether or not the matrix is symmetric and semi-negative definite; if true, then y = 1 = <TRUE>; if not, y = 0 = <FALSE>

EXAMPLE

```
//   An example for: isSymSemiNegDef(A)

//  Compute isSymSemiNegdef(A) for a 4 by 4 real matrix A

A = {-1,    -2,     1,      -6;
     -2,  -104,    52,     198;
      1,    52,   -26,     -99;
     -6,   198,   -99,    -477};
y = isSymSemiNegdef(A);
//  Result :
//               y:  1
```

■ isSymSemiPosDef

FUNCTION

y = isSymSemiPosDef(A)

PURPOSE

Determine whether or not a n by n real square matrix A is symmetric and semi-positive definite

INPUT

A (Real Matrix): a given n by n real matrix A

OUTPUT

y (Integer Scalar): A boolean result indicating whether or not the matrix is symmetric and semi-positive definite; if true, then y = 1 = <TRUE>; if not, y = 0 = <FALSE>

EXAMPLE

```
//  An example for: isSymSemiPosDef(A)

//  Compute isSymSemiPosDef(A) for a 3 by 3 real matrix A

A = {55,   130,   205;
    130,   330,   530;
    205,   530,   855};
y = isSymSemiPosDef(A);

//  Result :    y:  1
```

■ isTriangular

FUNCTION

y = isTriangular(u)

PURPOSE

Determine if a given vector u is in triangular form

INPUT

u (Real Vector): a n - dimensional vector storing the upper or lower triangular part of a square matrix

OUTPUT

y (Integer Scalar): The row or column dimension of the upper or lower triangular square matrix represented by the input vector u.

EXAMPLE

```
//  An example for:  isTriangular(u)

//  Compute  isTriangular(u) for a real vector u

u = {1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15};
y = isTriangular(u);

//  Result :
//             y:  5
```

■ lowerBand

FUNCTION

ml = lowerBand(A)

PURPOSE

Compute the lower bandwidth of a m by m square matrix A.

INPUT

A (Real, Complex Vector or Matrix): a m by m vector or matrix A

OUTPUT

ml (Real Scalar): the value indicating the lower bandwidth of the matrix A

EXAMPLE

```
//  Examples for: lowerBand(A)

//  Compute lowerBand(A) for a 6 by 6 real matrix A

A = { 1, -1, -1, 0,  0,  0;
     -1,  2,  0,  3,  0,  0;
      0,  0,  3,  1,  1,  0;
      0,  0,  1,  4,  2,  3;
      0,  0,  0,  2,  5, -9;
      0,  0,  0,  0,  3,  1 };

ml = lowerBand(A);

//  Result :
//            ml:  1

//  Compute lowerBand(B) for a 5 by 5 complex matrix B

B = { (1.1, -1),   (1.2, 2),  (0,0),   (0,0),     (0,0);
    (0,0),  (-1.0,  2.2),  (0.2,  -1.8),   (0,0),  (0,0);
    (0,0),  (0,0), (3.3, 0),  (0, -3.4),  (0,0);
     (0,0),  (0,0),  (0,0),  (4.4, -4.4),  (1,0);
     (0,0),   (0,0),  (0,0),    (0,0),   (5, 2)};
ml1 = lowerBand(B);

//  Result :
//            ml1:  0
```

■ sparsity

FUNCTION

y = sparsity(A)

PURPOSE

Compute the percentage of nonzero elements of a m by n real matrix A

INPUT

A (Real, Complex Vector or Matrix): a m by n vector or matrix A

OUTPUT

y (Real Scalar): A nonnegative real number (≤ 100) that indicates the percentage of elements in A that are not zero

EXAMPLE

```
//  Examples for: sparsity(A)

//  Compute sparisty(A) for a 5 by 5 real matrix A

A = {1, 0, 0, 0, -2;
     0, 3, 0, 0,  0;
     4, 0, 5, 6,  0;
     7, 0,-8, 0,  0;
     0, 0, 9, 0, 10};
y = sparsity(A);

//  Result :
//               y:        40

//  Compute sparsity(B) for a 4 by 4 complex matrix B

B = {(0,0), (1,1), (0,0), (0,0);
     (0,0), (2,2), (0,0), (3,3);
     (4,4), (0,0), (0,0), (0,0);
     (0,0), (0,0), (5,5), (0,0)};
z = sparsity(B);

//  Result :
//               z:      31.25
```

■ upperBand

FUNCTION

mu = upperBand(A)

PURPOSE

Compute the upper bandwidth of a m by m real square matrix A.

INPUT

A (Real, Complex Vector or Matrix): a m by m matrix A

OUTPUT

mu (Real Scalar): the value indicating the upper bandwidth of the matrix A

EXAMPLE

```
//  Examples for: upperBand(A)

//  Compute upperBand(A) for a 6 by 6 real matrix A

A = {  1, -1, -1, 0,  0,  0;
      -1,  2,  0,  3,  0,  0;
       0,  0,  3,  1,  1,  0;
       0,  0,  1,  4,  2,  3;
       0,  0,  0,  2,  5, -9;
       0,  0,  0,  0,  3,  1 };
mu = upperBand(A);

//  Result :
//             mu:  2
```

CHAPTER 17
EIGENVALUE FUNCTIONS

■ comEV

FUNCTION

[λVector, X] = comEV(A)

PURPOSE

Compute all the eigenvalues and the corresponding eigenvectors of a n by n complex matrix

INPUT

A (Complex Matrix): the n by n complex matrix

OUTPUT

λVector (Complex Vector): the n-dimensional complex vector containing all the computed eigenvalues of A

X (Complex Matrix): the n by n matrix containing all the computed eigenvectors of A. The jth column of X is the eigenvector corresponding to the jth component of the output complex vector λ. Each eigenvector is normalized so that its largest component is always unity

EXAMPLES

```
//  An example for: comEV(A)

//  Compute comEV(A) for a 4 by 4 complex matrix A:

A= {(1,3), (2,1), (3,2), (1,1);
   (3,4), (1,2), (2,1), (4,3);
   (2,3), (1,5), (3,1), (5,2);
   (1,2), (3,1), (1,4), (5,3)};
[LVector,X] =comEV(A);

//  Results :
//  LVector:  4 rows
//     9.78365812739986 +9.32251422469791i
//    -3.37100978506097 -0.7704539869710651i
//     1.36566930995022 -1.40105359746874i
//     2.22168234771089 +1.8489933597419i
//
//  X:  4 rows, 4 columns


//  column 1:
```

```
//  0.632337764496502   -0.0143780794466629i
//  0.873758591708999   +0.00810578078265854i
//  1                   +0i
//  0.943717588375998   +0.0379846348044158i
//   column 2:
// -0.506096461954582   +0.583451962603025i
//  1                   -0i
//  0.518319483940436   -0.714657072195101i
// -0.553484898199221   +0.0187563320214037i
//  column 3:
// -0.00084659871638733 +0.730203551263619i
// -0.0879417015769629  -0.387901799615492i
//  1                   -0i
// -0.432089374804352   -0.433426369347561i
//  column 4:
// -0.796620817436692   +0.304980785944659i
// -0.178834394739373   +0.4297241477445O5i
// -0.252814311193797   +0.0381734049516704i
//  1                   -0i
```

ALGORITHM AND COMMENTS

The given n by n complex matrix A is first reduced to a complex upper Hessenberg matrix H using a unitary similarity (i.e., the complex Householder) transformation P such that:

$$H = P^*AP \tag{1}$$

where [ ]* denotes the conjugate transpose of a matrix.

Then the efficient QR (unitary similarity tranformation) method with shifting is applied to further reduce the upper Hessenberg matrix H to a quasi-triangular matrix T - an upper Hessenberg matrix with principal diagonal containing 1 by 1 or 2 by 2 submatrices, such that:

$$T = Q^*HQ \tag{2}$$

where Q is an unitary matrix.

The eigenvalues of the original matrix A are obtained by collecting the (real) eigenvalues of the 1 by 1 submatrices and the complex eigenvalues (in conjugate pairs) of the 2 by 2 submatrices on the principal diagonal of the quasi-triangular matrix T.

Let Ω be the n by n diagonal matrix with its jth diagonal element being the jth eigenvalue of A (also T) and X be the n by n complex matrix with its jth column X_j being the eigenvector corresponding to the jth eigenvalue . Then, from (1) and (2), we have:

$$Q^*P^*APQX = X\Omega$$

or

$$A(PQX) = (PQX)\Omega$$

That is the eigenvectors of A are the columns of the n by n matrix PQX.

The eigenvectors of a quasi-triangular matrix T can be easily computed by a well-known backsubstitution process, see for example, the reference of Wilkinson and Reinsch.

To avoid unnecessary loss of accuracy of the computed eigenvalues, we implement a "balance" process on the matrix A. This reduces the L_1 matrix norm and makes the sums of the magnitudes of the elements in the corresponding rows and columns nearly equal, using an exact similarity transformation before the reduction to upper Hessenberg form takes place. For detailed information of how to balance a matrix, see the reference of Smith, et. al., listed below.

In addition, the maximum number of iterations for computing an eigenvalue of T (or A) using the QR method is set to 30 in our implementation. If an eigenvalue can not be determined after 30 iterations, an error message will be returned.

REFERENCES

Smith, B.T., Boyle, J.M., Dongarra, J.J., Garbow, B.S., Ikebe, Y., Klema, V.C. and Moler, C.B., *Lecture Notes in Computer Science : Matrix Eigensystem Routines - EISPACK Guide*, 2nd Edition, Springer-Verlag, New York, 1976, pp. 224-230

Wilkinson, J.H. and Reinsch, C., *Handbook for Automatic Computation*, Vol. II, Linear Algebra, Springer-Verlag, New York, 1971 , pp.372-373

■ comEVal

FUNCTION

λVector = comEVal(A)

PURPOSE

Compute all the eigenvalues of a n by n complex matrix

INPUT

A (Complex Matrix): the n by n complex matrix

OUTPUT

λVector (Complex Vector) : the n-dimensional complex vector containing all the computed eigenvalues of A

EXAMPLES

```
//  An example for: comEVal(A)
//  Compute comEVal(A) for a 4 by 4 complex matrix A:

A= {(1,3), (2,1), (3,2), (1,1);
   (3,4), (1,2), (2,1), (4,3);
   (2,3), (1,5), (3,1), (5,2);
   (1,2), (3,1), (1,4), (5,3)};
LVector =comEVal(A);
```

```
//  Results :
//  LVector:  4 rows
//      9.78365812739986 +9.32251422469791i
//     -3.37100978506097 -0.7704539869710651i
//      1.36566930995022 -1.401053597468741
//      2.22168234771089 +1.8489933597419i
```

ALGORITHM AND COMMENTS

The given n by n complex matrix A is reduced to a complex upper Hessenberg matrix using a unitary similarity (i.e., the complex Householder) transformation. Then the efficient QR method with shifting is applied to further reduce the upper Hessenberg matrix to a quasi-triangular matrix T - an upper Hessenberg matrix with principal diagonal containing 1 by 1 or 2 by 2 submatrices, such that:

$$T = Q^*AQ$$

where Q is an unitary matrix, and [ ]* denotes the conjugate transpose of a matrix.

The eigenvalues of the original complex matrix A are obtained by collecting the (real) eigenvalues of the 1 by 1 submatrices and the complex eigenvalues (in conjugate pairs) of the 2 by 2 submatrices on the principal diagonal of the quasi-triangular matrix T.

The "balance" process used in the function comEV() for computing all of the eigenvalues and eigenvectors is also implemented to avoid the unnecessary loss of accuracy of computed eigenvalues. For detailed information of this process, the reference listed below.

The maximum number of iterations for computing an eigenvalue of T (or A) using the QR method is set to 30 in our implementation. If an eigenvalue can not be determined after 30 iterations, an error message will be returned.

REFERENCE

Smith, B.T., Boyle, J.M., Dongarra, J.J., Garbow, B.S., Ikebe, Y., Klema, V.C. and Moler, C.B., *Lecture Notes in Computer Science : Matrix Eigensystem Routines - EISPACK Guide*, 2nd Edition, Springer-Verlag, New York, 1976, pp. 224 - 230

■ EV

FUNCTION

[λVector, X] = EV(A)

PURPOSE

Compute all of the eigenvalues and the corresponding eigenvectors of a n by n real matrix

INPUT

A (Real Matrix): the n by n real matrix

OUTPUT

λVector (Complex Vector): the n-dimensional complex vector containing all of the computed eigenvalues of A

X (Complex Matrix): the n by n matrix containing all of the computed eigenvectors of A. The jth column of X is the eigenvector corresponding to the jth component of the output complex vector λ. Each eigenvector is normalized so that its largest component is always unity

EXAMPLES

```
//  An example for: EV(A)
//  Compute EV(A) for a 4 by 4 real matrix:

A= {1, 2, 3, 1;
   3, 1, 2, 4;
   2, 1, 3, 5;
   1, 3, 1, 5};
[LVector,X] = EV(A);

//  Results :
//  LVector:  4 rows
//    9.70034979064718    +0i
//   -2.28150208220082    +0i
//    1.29057614577682    +1.14743439640177i
//    1.29057614577682    -1.14743439640177i
//
//  X:  4 rows, 4 columns
//  column 1:
//    0.649070218248055   -0i
//    0.870386703677542   -0i
//    1                   -0i
//    0.906364530094706   -0i
//  column 2:
//   -0.966665754267228   -0i
//    1                   -0i
//    0.506996756107969   -0i
//   -0.348874582903769   -0i
//  column 3:
//     1                 +0i
//     0.5169423398425    -0.232990405931689i
//    -0.0390786454130636 +0.592820486552752i
//    -0.626072597668989  -0.165046251393103i
//  column 4:
//     1                 +0i
//     0.5169423398425    +0.232990405931689i
//    -0.0390786454130636 -0.592820486552752i
//    -0.626072597668989  +0.165046251393103i
```

ALGORITHM AND COMMENTS

The given n by n real matrix A is first "balanced" by using a sequence of similarity transformations applying a diagonal matrix which reduces the L_1 matrix norm and makes the sums of the magnitudes of the elements in the corresponding rows and columns nearly equal. Then the resultant balanced matrix is reduced to an upper Hessenberg matrix H using a similarity (i.e., the Householder) transformation P such that

$$H = P^t D^{-1} A D P \tag{1}$$

where D is the diagonal matrix that makes $D^{-1}AD$ a balanced matrix.

Then the efficient QR similarity tranformation with shifting is applied to further reduce the upper Hessenberg matrix H to a quasi-triangular matrix T - an upper Hessenberg matrix with principal diagonal containing either 1 by 1 or 2 by 2 submatrices, such that:

$$T = Q^t H Q \tag{2}$$

where Q is an orthogonal matrix.

The eigenvalues of the original matrix A are obtained by collecting the (real) eigenvalues of the 1 by 1 submatrices and the complex eigenvalues (in conjugate pairs) of the 2 by 2 submatrices on the principal diagonal of the quasi-triangular matrix T.

Let Ω be the n by n diagonal matrix with its jth diagonal element being the jth eigenvalue of A (also T) and X be the n by n real matrix with its jth column X_j being the eigenvector corresponding to the jth eigenvalue . Then, from (1) and (2), we have:

$$Q^t P D^{-1} A D P Q X = X\Omega$$

or

$$A\,(DPQX) = (DPQX)\,\Omega$$

That is, the eigenvectors of A are the columns of the n by n matrix DPQX.

The eigenvectors of a quasi-triangular matrix T can be easily computed by a well-known backsubstution process, the reference of Wilkinson and Reinsch.

In our implementation, the maximum number of iterations for computing an eigenvalue of T (or A) using the QR method is set to 40. If an eigenvalue can not be determined after 40 iterations, an error message will be returned.

REFERENCES

Press, W.H, Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T., *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge, 1986, pp. 365-380

Wilkinson, J.H. and Reinsch, C., *Handbook for Automatic Computation*, Vol. II, Linear Algebra, Springer-Verlag, New York, 1971, pp.372-373

■ eVal

FUNCTION

λVector = eVal(A)

PURPOSE

Compute all of the eigenvalues of a n by n real matrix

INPUT

A (Real Matrix): the n by n real matrix

OUTPUT

λVector (Complex Vector): the n-dimensional complex vector containing all of the computed eigenvalues of A

EXAMPLE

```
//  An example for: eVal(A)

//  Compute eVal(A) for a 4 by 4 real matrix:

A= {1, 2, 3, 1;
    3, 1, 2, 4;
    2, 1, 3, 5;
    1, 3, 1, 5};
LVector = eVal(A);

//  Results:
//  LVector :  4 rows
//     9.70034979064718 +0i
//    -2.28150208220082 +0i
//     1.29057614577682 +1.14743439640177i
//     1.29057614577682 -1.14743439640177i
```

ALGORITHM AND COMMENTS

The given n by n real matrix A is first "balanced" by a sequence of similarity transformations with a diagonal matrix that reduces the L_1 matrix norm and makes the sums of the magnitudes of elements in the corresponding rows and columns nearly equal. Then the resultant balanced matrix is reduced to an upper Hessenberg matrix H using a similarity (i.e., the Householder) transformation such that:

$$H = P^t D^{-1} A D P$$

where D is the diagonal matrix that makes $D^{-1}AD$ a balanced matrix.

Then the efficient QR similarity tranformation with shifting is applied to further reduce the upper Hessenberg matrix H to a quasi-triangular matrix T - an upper Hessenberg matrix with principal diagonal containing

either 1 by 1 or 2 by 2 submatrices, such that:

$$T = Q^t H Q$$

The eigenvalues of the original matrix A are obtained by collecting the (real) eigenvalues of the 1 by 1 submatrices and the complex eigenvalues (in conjugate pairs) of the 2 by 2 submatrices on the principal diagonal of the quasi-triangular matrix T.

In our implementation, the maximum number of iterations for computing an eigenvalue of T (or A) using the QR method is set to 40. If an eigenvalue can not be determined after 40 iterations, an error message will be returned.

REFERENCE

Press, W.H, Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T., *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge, 1986, pp. 365-380

■ genEV

FUNCTION

[λVector, X] = genEV(A, B)

PURPOSE

Compute all the eigenvalues and their corresponding eigenvectors for the generalized eigenvalue problem:

$$Ax = \lambda Bx$$

where A and B are both n by n real matrices

INPUT

A (Real Matrix): a n by n matrix

B (Real Matrix): a n by n matrix

OUTPUT

λVector (Complex Vector): the k-dimensional complex vector containing all the k computed eigenvalues of $Ax = \lambda Bx$, where $0 \le k \le n$

X (Complex Matrix): the n by k matrix whose columns consist of the eigenvectors of $Ax = \lambda Bx$ corresponding to the computed eigenvalues λ where $0 \le k \le n$. Each eigenvector is normalized so that its largest component is always unity

EXAMPLE

```
//  An example for: genEV(A,B)
//  Compute genEV(A,B) for two 3 by 3 real matrices
```

```
//  A and B:

A= {3, 1, 2;
   2, 1, 3;
   4, 1, 3};
B ={2, 4, 4;
   0, 3, 1;
   0, 1, 3};
[LVector,X] = genEV(A,B);

//  Results:
//  LVector:  3 rows
//    -0.302581093188303 +1.047418188805899i
//    -0.302581093188303 -1.047418188805899i
//     0.105162186376605 +0i
//
//  X:  3 rows, 3 columns
//  column 1:
//    -0.547428898373756 -0.836852197956901i
//     0.390044714108891 +0.0738448714033621i
//    -0.208027428431906 +0.769263596582304i
//  column 2:
//    -0.547428898373756 +0.836852197956901i
//     0.390044714108891 -0.0738448714033621i
//    -0.208027428431906 -0.769263596582304i
//  column 3:
//     0.121221495412973 +0i
//    -1                 +0i
//     0.152709919693535 +0i
```

ALGORITHM AND COMMENTS

The eigenvalues and eigenvectors of the generalized eigenproblem are computed by using the stable QZ method (see the references for detailed information of this algorithm).

The number of eigenvalues of a general eigenproblem, $Ax = \lambda Bx$, can be any integer number between 0 and n. The complex eigenvalues are always in complex conjugate pairs.

If the QZ method cannot compute an eigenvalue after 50 similarity transformations, an error message will be returned.

REFERENCES

Golub, G.H. and VanLoan, C.F., *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989, chap. 7

Moler, C.B. and Stewart, G.W., "An Algorithm for Generalized Matrix Eigenvalue Problems," *SIAM Journal of Numerical Analysis*, 10, pp. 241 - 256

■ genEVal

FUNCTION

λVector = genEVal(A,B)

PURPOSE

Compute all the eigenvalues for the generalized eigenvalue problem:

$$Ax = \lambda Bx$$

where A and B are both n by n real matrices

INPUT

A (Real Matrix): a n by n matrix

B (Real Matrix): a n by n matrix

OUTPUT

λVector (Complex Vector): the k-dimensional complex vector containing all the k computed eigenvalues of $Ax = \lambda Bx$ where $0 \le k \le n$

EXAMPLE

```
//  An example for: genEVal(A,B)

//  Compute genEVal(A,B) for two 3 by 3 real matrices
//  A and B:

A= {3, 1, 2;
   2, 1, 3;
   4, 1, 3};
B ={2, 4, 4;
   0, 3, 1;
   0, 1, 3};
LVector= genEVal(A,B);

//  Results :
//  LVector:  3 rows
//    -0.302581093188303  1.047418188805899i
//    -0.302581093188303 -1.047418188805899i
//     0.105162186376605 +0i
```

ALGORITHM AND COMMENTS

The eigenvalues of the generalized eigenproblem, $Ax = \lambda Bx$, are computed by using the stable QZ method (see references for detailed information of this algorithm).

The number of eigenvalues of a general eigenproblem can be any integer number between 0 and n. The com-

plex eigenvalues always occur in complex conjugate pairs.

If the QZ method cannot compute an eigenvalue after 50 similarity transformations, an error message will be returned.

REFERENCES

Golub, G.H. and VanLoan, C.F., *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989, chap. 7

Moler, C.B. and Stewart, G.W., "An Algorithm for Generalized Matrix Eigenvalue Problems," *SIAM Journal of Numerical Analysis*, 10, pp. 241 - 256

■ genIter

FUNCTION

[λ, xVector] = genIter(A, B, β, maxIterations, tolerance)

PURPOSE

Compute the closest eigenvalue and its corresponding eigenvector to a given real number β for the following generalized eigenvalue/eigenvector problem:

$$Ax = \lambda Bx$$

using the inverse power method with Gaussian elimination, where A and B are both m by m matrices

INPUT

A (Real Matrix): the left hand side m by m matrix

B (Real Matrix): the right hand side m by m matrix

β (Real Scalar): the given real number for which the closest eigenvalue and its corresponding eigenvector will be computed

maxIterations (Integer Scalar): the maximum number of allowed iterations for the inverse power method.

tolerance (Real Scalar): the positive tolerance for determining the convergence to the desired eigenvalue and its corresponding eigenvector

OUTPUT

λ (Real Scalar): the computed eigenvalue that is closest to β

xVector (Real Vector): the m-dimensional vector corresponding to the (computed) eigenvalue λ

EXAMPLE

```
// An example for: genIter (A, B, e, maxIterations, tolerance)

// Compute genIter(A,B,e,iter,tol) for two 3 by 3 real
```

```
// matrices A and B with e = 1, iter = 100 and tol = 1e-16:

A= {3, 1, 2;
   2, 1, 3;
   4, 1, 3};
B ={2, 4, 4;
   0, 3, 1;
   0, 1, 3};
[Lambda,xVector] = genIter(A,B,1,100,1e-16);
//  Results:
//  Lambda:          0.105162186376605
//
//  xVector:  3 rows
//       -0.121221495412973
//        1
//       -0.152709919693535
```

ALGORITHM AND COMMENTS

Subtracting βBX from both sides of the given generalized eigenvalue/eigenvector problem, AX = λBX, we obtain

$$(A - \beta B)\,X = (\lambda - \beta)\,BX$$

If (A-βB) is nonsingular, then we have the equivalent problem:

$$(A - \beta B)^{-1} BX = cX$$

where c = 1/(λ-β). Since λ is expected to be the closest eigenvalue to β, c is expected to be the dominant eigenvalue of the matrix $(A-\beta B)^{-1}B$. Thus the power method can be applied directly to $(A-\beta B)^{-1}B$, i.e.,

$$Y_n = (A - \beta B)^{-1} BX_n$$

$$X_{n+1} = \frac{Y_n}{\|Y_n\|}$$

for n=0, 1, ..., until X_n has been shown to converge. Note that directly computing Y_n by the above formula requires the evaluation of the matrix inverse for (A-βB). This inversion can be avoided easily. In the function genIter, Y_n is computed by solving the following system of linear equations (using Gaussian elimination):

$$(A - \beta B)\,Y_n = BX_n$$

during each iteration. This is called the inverse power method.

The method converges when:

$$\|x_{n+1} - x_n\| < \text{tolerance}$$

for the user supplied positive tolerance, tolerance, where $\|\bullet\|$ denotes the maximum norm in R^m. When the method converges, the expected eigenvalue λ is such that $\lambda = \beta+1/c$, where c is the computed dominant eigenvalue of the matrix $(A-\beta B)^{-1}B$, and X_n is the expected eigenvector corresponding to λ.

Comments :

(1) Each component of the initial eigenvector X_0 is chosen automatically by a pseudo random number generator so that $\|X_0\| = 1$.

(2) The Gaussian elimination in each iteration is done by a forward substitution followed by back-substitution with the pre-established LU decomposition. Thus, it takes only $O(n^2)$ operations in each iteration.

(3) In case the matrix $A-\beta B$ is numerically singular, i.e., if during the establishment of LU decomposition some of the pivot elements are zero, we will replace such zero pivots by $\mu\|A-\beta B\|$ so that the process can be continued. Here μ is the machine precision, and $\|\bullet\|$ denotes the matrix infinity norm. For detailed information on such choices see the Reference.

(4) The inverse power method can fail to converge, as in the power method, when the desired eigenvalue is not simple.

(5) The maximum number of allowed iterations, maxIterations, has a default value of 1000 in the function. This value will be used whenever the user supplied value of maxIterations is less than 0.

(6) The value of the tolerance (for determining the convergence of inverse power method), has a default value of 10^{-8} in the function. This value will be used whenever the user-supplied tolerance is less than 0.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers : Linear Algebra and Function Minimisation*, Adam Hilger Ltd., 1979, pp. 88-91

■ hermEV

FUNCTION

[λVector, X] = hermEV(A)

PURPOSE

Compute all of the eigenvalues and the corresponding eigenvectors of a n by n complex Hermitian matrix

INPUT

A (Complex Matrix): the n by n complex Hermitian matrix

OUTPUT

λVector (Real Vector) : the n-dimensional real vector containing all of the computed eigenvalues of A

X (Complex Matrix) : the n by n matrix containing all of the computed eigenvectors of A. The jth column of X is the eigenvector corresponding to the jth component of the output real vector λ. Each eigenvector is normalized so that its largest component is always unity.

EXAMPLE

```
//  An example for: hermEV(A)
//  Compute hermEV(A) for a 4 by 4 complex  Hermitian matrix A:

A= {(1,0),  (2,1),   (-3,-2), (1,1);
   (2,-1), (-1,0),  (2,-1),  (4,3);
   (-3,2),  (2,1),   (3,0),  (5,-2);
   (1,-1), (4,-3),   (5,2),   (5,0)};
[LVector,X] =hermEV(A);

//  Results:
//  LVector:  4 rows
//      -6.31058356224902
//      -1.94035204548524
//       5.09519430300197
//      11.1557413047323
//
//  X:  4 rows, 4 columns
// column1:
//    -0.126183533119041  -0.589753545119111i
//     1                  -0i
//    -0.402978461992249  -0.686080350883919i
//    -0.233527412536373  +0.680772008625972i
// column2:
//     1                  +0i
//    -0.595131051661943  -0.746518082350704i
//     0.264580712762673  -0.595963901572561i
//     0.159249427046049  +0.670187510124963i
// column3:
//     1                  -0i
//     0.555012946371518  +0.128473931672242i
//    -0.469884678841573  +0.614536195271969i
//     0.283397166639638  -0.191518748222878i
// column4:
//    -0.0838887218525142 +0.097902375805398i
//     0.421751730389682  +0.178135379208522i
//     0.701497176252746  -0.206414355516569i
//      1                 -0i
```

ALGORITHM AND COMMENTS

The given n by n complex Hermitian matrix A is first reduced to a real symmetric tridiagonal matrix H using a unitary similarity (i.e., the complex Householder) transformation P such that:

$$H = P^*AP \tag{1}$$

where [ ]* denotes the conjugate transpose of a matrix.

Then the efficient QL (unitary similarity tranformation) method with shifting for a real symmetric tridiagonal matrix is applied to further reduce the tridiagonal matrix H to a diagonal matrix D (i.e., zeroing the off-diago-

nal elements to machine precision) such that:

$$D = Q^t H Q \quad (2)$$

where Q is an orthogonal matrix.

The eigenvalues of the original complex matrix A are the diagonal elements of the matrix D.

From (1) and (2), it is easy to see that the eigenvectors of A are the column vectors of PQ.

In our implementation, the maximum number of iterations for computing an eigenvalue of H (or A) using the QL method is set to 30. If an eigenvalue can not be determined after 30 iterations, an error message will be returned.

REFERENCE

Wilkinson, J.H. and Reinsch, C., *Handbook for Automatic Computation*, Vol. II, Linear Algebra, Springer-Verlag, New York, 1971, pp 227-240

■ hermEVal

FUNCTION

λVector = hermEVal(A)

PURPOSE

Compute all of the eigenvalues of a n by n complex Hermitian matrix

INPUT

A (Complex Matrix): the n by n complex Hermitian matrix

OUTPUT

λVector (Real Vector) : the n-dimensional real vector containing all of the computed eigenvalues of A

EXAMPLE

```
//  An example for: hermEVal(A)

//  Compute hermEVal(A) for a 4 by 4 complex  Hermitian matrix // A:

A= {(1,0),  (2,1),   (-3,-2), (1,1);
   (2,-1), (-1,0),  (2,-1),  (4,3);
   (-3,2),  (2,1),   (3,0),  (5,-2);
   (1,-1), (4,-3),   (5,2),   (5,0)};
LVector =hermEVal(A);

//  Results:
//  LVector:  4 rows
//      -6.31058356224902
```

```
//      -1.94035204548524
//       5.09519430300197
//      11.1557413047323
```

ALGORITHM AND COMMENTS

The given n by n complex Hermitian matrix A is first reduced to a real symmetric tridiagonal matrix H using a unitary similarity (i.e., the complex Householder) transformation. Then the efficient QL (plane rotation) method with shifting for a real symmetric tridiagonal matrix is applied to further reduce the tridiagonal matrix H to a diagonal matrix D (i.e., zeroing off diagonal elements to machine precision). Consequently, the eigenvalues of the original complex matrix A are the diagonal elements of the matrix D.

In our implementation, the maximum number of iterations for computing an eigenvalue of H (or A) using the QL method is set to 30. If an eigenvalue can not be determined after 30 iterations, an error message will be returned.

REFERENCE

Wilkinson, J.H. and Reinsch, C., *Handbook for Automatic Computation*, Vol. II, Linear Algebra, Springer-Verlag, New York, 1971, pp 227-240

■ powerEV

FUNCTION

[λ, xVector] = powerEV(A, maxIterations, tolerance)

PURPOSE

Compute the dominant eigenvalue (i.e., the one with maximum absolute value) and its corresponding eigenvector for a given m by m matrix A using the power method with Atkin's Δ^2 - procedure

INPUT

A (Real Matrix): a given m by m matrix

maxIterations (Integer Scalar): the maximum number of allowed iterations for the power method

tolerance (Real Scalar): the positive tolerance for determining convergence to the desired eigenvalue and its corresponding eigenvector

OUTPUT

λ (Real Scalar): the computed dominant eigenvalue

xVector (Real Vector): the m-dimensional eigenvector corresponding to the computed eigenvalue λ

EXAMPLE

```
//  An example for: powerEV(A, maxIterations, tolerance)

//  Compute powerEV(A, iter, tol) for a 4 by 4 real matrix A
```

```
//  with iter = 100 and tol = 1e-16:

A= {1, 2, 3, 1;
   3, 1, 2, 4;
   2, 1, 3, 5;
   1, 3, 1, 5};
Lambda = powerEV(A, 100, 1e-16);

//  Result :
//            Lambda:           9.70034979064718

//  Write {Lambda, xVector] = powerEV (A, 100, 1e-16) to
//  return the eigenvector corresponding to Lambda
```

ALGORITHM AND COMMENTS

The power method uses the following iterative procedure:

$$Y_n = AX_n$$

$$X_{n+1} = \frac{Y_n}{\|Y_n\|}$$

$$\lambda_{n+1} = Y_{n,p}$$

for n=0, 1, ..., with initial vector X_0, where $\|\bullet\|$ denotes the maximum norm in R^m and $X_{n,p} = \|X_n\|$. The power method converges when:

$$\|x_{n+1} - x_n\| < \text{tolerance}$$

for the user-supplied positive value of tolerance. The Atkin's Δ^2-procedure is used to speed up the convergence of the eigenvalue by replacing the computed λ_{n+1} with

$$\lambda_{n-1} - \frac{(\lambda_n - \lambda_{n-1})^2}{\lambda_{n+1} - 2\lambda_n + \lambda_{n-1}}$$

where n = 2, ...

Comments :

(1) Each component of the initial eigenvector X_0 is chosen automatically by a pseudo random number generator so that $\|X_0\| = 1$.

(2) The power method works for the case of real dominant eigenvalue. However, it can fail to converge if the dominant eigenvalue is not simple.

(3) The maximum number of allowed iterations, maxIterations, has a default value of 1000 in the function.

This value will be used whenever the user supplied maxIterations is less than 0.

(4) The value of tolerance (the value for determining the convergence of the power method) has a default value of 10^{-8} in the function. This value will be used whenever the user supplied tolerance is less than 0.

REFERENCE

Burden, R.L. and Gaires, J.D., *Numerical Analysis*, 3rd edition, Prindle, Weber and Schmidt, Boston, 1985, pp. 454-455

■ symEV

FUNCTION

[λVector, B] = symEV(A)

PURPOSE

Compute all the eigenvalues and their corresponding eigenvectors for a symmetric matrix A using the QL transformation

INPUT

A (Real Matrix): a m by m real symmetric matrix A

OUTPUT

λVector (Real Vector): the m-dimensional vector containing the eigenvalues of A

B (Real Matrix): all of the eigenvectors of A are stored in the original columns of A (i.e., they overwrote the original columns)

EXAMPLE

```
//  An example for: symEV(A)

//  Compute symEV(A) for a 4 by 4 real matrix:

A= {5, 2, 1, 1;
    2, 6, 3, 1;
   1, 3, 6, 3;
   1, 1, 3, 6};
{LVector,B] = symEV(A);

//  Results:
//  LVector:  4 rows
//         4
//         5.41079512359386
//         2.03067278171029
//        11.5585320946958
//
//  B:  4 rows, 4 columns
```

```
//  1   -0.745617936466869    0.427939919105341   0.541264120762663
// -0.5 -0.815060339430973   -0.804937937127167   0.877475449751526
// -0.5  0.323824466497235    1                   1
//  0.5  1                   -0.660817775337848   0.7949472082062
```

ALGORITHM AND COMMENTS

If the returning value of this routine is zero, the algorithm failed to converge to a computed eigenvalue after performing 30 QL iterations.

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989

■ symEVal

FUNCTION

λVector = symEVal(A)

PURPOSE

Compute all the eigenvalues of a m by m symmetric matrix A using the QL transformation

INPUT

A (Real Matrix): a m by m real symmetric matrix A

OUTPUT

λVector (Real Vector): the m-dimensional vector containing the eigenvalues of A

EXAMPLE

```
//  An example for: symEVal(A)
//  Compute symEVal(A) for a 4 by 4 real matrix A:

A= {5, 2, 1, 1;
   2, 6, 3, 1;
   1, 3, 6, 3;
   1, 1, 3, 6};
LVector = symEVal(A);

//  Results:
//  LVector:  4 rows
//         4
//         5.41079512359386
//         2.03067278171029
//        11.5585320946958
```

ALGORITHM AND COMMENTS

If the returning value of this routine is zero, the algorithm failed to converge for a computed eigenvalue after performing 30 QL interations.

REFERENCE

Golub, G.H. and Van Loan, C.F., *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989

■ symPower

FUNCTION

[λ, xVector] = symPower(A, maxIterations, tolerance)

PURPOSE

Compute the dominant eigenvalue (i.e., the one with maximum absolute value) and its corresponding eigenvector for a given m by m symmetric matrix A using the Rayleigh quotient

INPUT

A (Real Matrix): a given m by m symmetric matrix

maxIterations (Integer Scalar): the maximum number of allowable iterations for the Rayleigh quotient

tolerance (Real Scalar): the positive error tolerance used to determine convergence to an eigenvalue and its corresponding eigenvector

OUTPUT

λ (Real Scalar): the computed dominant eigenvalue

xVector (Real Vector): the computed m-dimensional eigenvector corresponding to the (computed) eigenvalue λ

EXAMPLE

```
//  An example for: symPower(A, maxIterations, tolerance)

//  Compute symPower(A, iter, tol) for a 4 by 4 real symmetric
//  matrix A with iter = 100 and tol = 1e-16:

A= {5, 2, 1, 1;
   2, 6, 3, 1;
   1, 3, 6, 3;
   1, 1, 3, 6};
Lambda = symPower(A, 100, 1e-16);

//  Result :
//            Lambda:           11.5585320946958

//  Use the call: [Lambda, xVector] to also return the
```

```
//  eigenvector xVector corresponding to Lambda
```

ALGORITHM AND COMMENTS

The Rayleigh quotient for computing the dominant eigenvalue and its corresponding eigenvector is:

$$Y_n - AX_n$$

$$X_{n+1} = \frac{Y_n}{\sqrt{Y_n^t Y_n}}$$

$$\lambda_{n+1} = x_n^t AX_n$$

for n=0, 1, ... , with initial vector X_0. The method converges when

$$\|x_{n+1} - x_n\| < \text{tol}$$

for the user-supplied positive value of tolerance, where $\|\bullet\|$ denotes the L_2 norm in R^m.

Comments :

(1) Each component of the initial eigenvector X_0 is chosen automatically by a pseudo random number generator so that $\|X_0\| = 1$.

(2) The Rayleigh quotient procedure is, in general, considered more suitable for computing the dominant eigenvalue and its corresponding eigenvector of a symmetric matrix than is the power method, because it converges faster than the power method.

(3) The Rayleigh quotient also works for non-symmetric matrices, however its convergence need not be faster than power method in this case. The Rayleigh quotient procedure may fail to converge as in the power method if the dominant eigenvalue is not simple.

(4) The maximum number of allowable iterations, maxIterations, has a default value 1000 in the function. This value will be used whenever the user supplied maxIterations is less than 0.

(5) The value of tolerance (the value for determining the convergence of the Rayleigh quotient procedure) has a default value of 10^{-8} in the function. This value will be used whenever the user supplied tolerance is less than 0.

REFERENCE

Burden, R.L. and Faires, J.D., *Numerical Analysis*, 3rd edition, Prindle, Weber and Schmidt, Boston, 1985, pp. 457

CHAPTER 18
FOURIER ANALYSIS FUNCTIONS

■ bilinear

FUNCTION

[c, d] = bilinear(a, b)

PURPOSE

Generate a digital filter function from an analog filter function, which is described by the ratio of two polynomials in s, using the bilinear transform method

INPUT

a (Real Vector): a real vector storing the coefficients of the polynomial in s for the numerator of the analog filter function

b (Real Vector): a real vector storing the coefficients of the polynomial in s for the denominator of the analog filter function

OUTPUT

c (Real Vector): a real vector storing the coefficients of the polynomial in z^{-1} for the numerator of the digital filter function

d (Real Vector): a real vector storing the coefficients of the polynomial in z^{-1} for the denominator of the digital filter function. The first element of d is always unity on output

EXAMPLES

```
//  An example for: bilinear(v1, v2)
//

//  Compute bilinear(U,V) where U is a 5-dimensional real
//  vector and V is a 6-dimensional real vector:

U = { 5; 4; 3; 2; 1};
V = {8; 4; 9; 3; 2; 1};

EXbilinear = bilinear(U,V);

//  Results :
//  EXbilinear:  6 rows
//     0.555555555555556
//     1.2962962962963
//     1.85185185185185
```

```
//     1.55555555555556
//     0.555555555555556
//     0.111111111111111
```

ALGORITHM AND COMMENTS

The original analog filter function

$$H_a(s) = \frac{\sum_{k=0}^{M} a_k s^k}{\sum_{j=0}^{N} b_j s^j} \qquad (1)$$

is specified by the given input vectors

$$a = (a_0, a_1, \ldots, a_M)$$
$$b = (b_0, b_1, \ldots, b_N)$$

The bilinearly transformed digital filter function is

$$H_d(z) = \frac{\sum_{k=0}^{K} c_k z^{-k}}{\sum_{j=0}^{L} d_j z^{-j}} \qquad (2)$$

The coefficients of numerator and denomintor in (2), i.e.,

$$c = (c_0, c_1, \ldots, c_K)$$
$$d = (1, d_1, \ldots, d_L)$$

are obtained by substituting $s = (z - 1)/(z + 1)$ in (1) and then normalizing to make the constant term in the denominator equal to one.

REFERENCES

Oppenheim, A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice-Hall, 1975

Bose, N.K., *Digital Filters Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ convolve

FUNCTION

w = convolve(u, v)

PURPOSE

Compute the convolution between a real data sequence and a response sequence

INPUT

u (Real Vector): a real vector representing the data sequence

v (Real Vector): a real vector representing the response sequence

OUTPUT

w (Real Vector): a real vector representing the result of convolving the data sequences u with v. If the lengths of u and v are L and M respectively, then the length of u is L+M-1

EXAMPLES

```
//  An example for: convolve(v1, v2)
//

//  Compute convolve(U,V) where U is a 5-dimensional real
//  vector and V is a 6-dimensional real vector:

U = { 1; 2; 3; 4; 5};
V = {1; 2; 3; 9; 4; 8};

EXconvolve = convolve(U,V);

//  Results :
//  EXconvolve:  10 rows
//                        1
//                        4
//                       10
//                       25
//                       44
//                       65
//                       79
//                       85
//                       52
//                       40
```

ALGORITHM AND COMMENTS

The convolution between two real sequences u and v of length L and M, respectively, is the vector w = (w_0,..., w_{L+M-2}) of length L+M-1 with

$$w_n = \sum_{\substack{k = 0 \\ n - M + 1 \le k \le n}}^{L - 1} u_k v_{n-k} \qquad (1)$$

where

$$u = (u_0, u_1, \ldots, u_{L-1})$$
$$v = (v_0, v_1, \ldots, v_{M-1})$$

In order to speed up the computation indicated in (1), the fast Fourier transform (FFT) algorithm is applied. First the FFT algorithm is used to obtain the Fourier coefficients of u_n and v_n. Then the Fourier coefficients of w_n is calculated by multiplying the Fourier coefficients of u_n and v_n. Finally the inverse FFT algorithm is used to obtain w_n. Both the FFT algorithm and its inverse require the input array length equal to a power of two and no less than four. The lengths of the input vectors u and v are in general not a power of two. This problem is resolved by padding zeros to the end of each vector to make them so.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filters Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ correl

FUNCTION

w = correl(u)

PURPOSE

Compute the auto-correlation of a real data sequence

INPUT

u (Real Vector): a real vector representing the data sequence

OUTPUT

w (Real Vector): a real vector representing the auto-correlation results. The lengths of v and u are equal

EXAMPLES

```
//  An example for: correl(v)
//

//  Compute correl(U) where U is a 5-dimensional real vector:
```

```
U = { 1; 2; 3; 4; 5};

EXcorrel = correl(U);

//  Results :
//  EXcorrel:  5 rows
//                      55
//                      40
//                      26
//                      14
//                       5
```

ALGORITHM AND COMMENTS

The auto-correlation of a real data sequences u of length L is the vector $w = (w_0,..., w_{L-1})$ of the same length with

$$w_n = \sum_{k=0}^{L-n-1} u_k u_{n+k} \qquad (1)$$

where

$$u = (u_0, u_1, \ldots, u_{L-1})$$

In order to speed up the computation indicated in (1), the FFT algorithm is applied. First the FFT algorithm is used to obtain the Fourier coefficients of u_n. Then the Fourier coefficients of w_n is calculated by multiplying the complex conjugates of the Fourier coefficients of u_n with the Fourier coefficients of u_n. Finally the inverse FFT algorithm is used to obtain w_n. Both the FFT algorithm and its inverse require the input array length equal to a power of two and no less than four. The lengths of the input vectors u and v are in general not a power of two. This problem is resolved by padding zeros to the end of each vector to make them so.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filters Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ cosFT

FUNCTION

w = cosFT(u)

PURPOSE

Compute the discrete cosine transform of a real data sequence of length equal to a power of two and no less than eight

INPUT

u (Real Vector): a real vector of length equal to a power of two and no less than eight that representis the data sequence

OUTPUT

w (Real Vector): a real vector with the same length as the input vector u representing the result of the discrete cosine transform

EXAMPLES

```
//  An example for: cosFT(v)
//

//  Compute cosFT(U) for an 8-dimensional real vector U:

U = { 1;   0.707106781186548; 0;   -0.707106781186548;
              -1;  -0.707106781186548; 0;   0.707106781186548};

EXcosFT = cosFT(U);

//  Results:
//  EXcosFT:  8 rows
//     0
//     1
//     4
//     1
//     0
//     1
//     -1.34506121518552e-15
//     1
```

ALGORITHM AND COMMENTS

The discrete cosine transform of a real data sequence u of length N is the vector w of the same length with

$$w_n = \sum_{k=0}^{L-n-1} u_k \cos\left(\frac{\pi nk}{N}\right) \qquad (1)$$

where $8 \le N = 2P$, $p \ge 3$, and

$$u = (u_0, u_1, \ldots, u_{N-1})$$

To speed up the computation indicated in (1), the FFT algorithm is used. For detailed information, see the references.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filters Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ crossCorrel

FUNCTION

w = crossCorrel(u, v)

PURPOSE

Compute the cross-correlation of two real data sequences

INPUT

u (Real Vector): a real vector representing the first input data sequence

v (Real Vector): a real vector representing the second input data sequence

OUTPUT

w (Real Vector): a real vector representing the result of cross correlating u with v. If the lengths of u and v are L and M, respectively, then the length of w is L+M-1

EXAMPLES

```
//  An example for: crossCorrel(v1, v2)
//

//  Compute crossCorrel(U,V) where U is a 5-dimensional real
//  vector and V is a 6-dimensional real vector:

U = { 1; 2; 3; 4; 5};
V = {1; 2; 3; 9; 4; 8};

EXcrossCorrel = crossCorrel(U,V);

//  Results:
//  EXcrossCorrel:  10 rows
//                5
//               14
//               26
//               65
//               70
//               91
//               65
//               41
//               20
//                8
```

ALGORITHM AND COMMENTS

The cross-correlation between two real sequences u and v of lengths L and M, respectivly, is the vector w = $(w_{-L+1},...,w_{M-1})$ of length L+M-1 with

$$w_n = \sum_{\substack{k=0 \\ -n \le k \le M-1-n}}^{L-1} u_k v_{n+k} \qquad (1)$$

where

$$u = (u_0, u_1, \ldots, u_{L-1})$$
$$v = (v_0, v_1, \ldots, v_{M-1})$$

In order to speed up the computation indicated in (1), the FFT algorithm is applied. First the Fourier coefficients of u_n, v_n are computed by the FFT. Then the complex conjugates of the Fourier coefficients of u_n are multiplied by the Fourier coefficients of v_n to obtain the Fourier coefficients of w_n. Finally the inverse FFT algorithm is used to obtain w_n. Both the FFT algorithm and its inverse require the input array length equal to a power of two and no less than four. The lengths of the input vectors u and v are in general not a power of two. This problem is resolved by padding zeros to the end of each vector to make them so.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filters Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ csd

FUNCTION

w = csd(u1, u2, N, type, r)

PURPOSE

Compute the cross power spectral density of two real data sequences using the periodogram method with FFT

INPUT

u1 (Real Vector): a real vector representing the first data sequence

u2 (Real Vector): a real vector representing the second data sequence

N (Integer Scalar): an integer equal to a power of two and no less than four that specifies the half length of a FFT to be performed on segments of u1 and u2

type (Integer Scalar): an integer specifying the window type: 1 = Bartlett, 2 = Parzen, 3 = Hanning, 4 = Hamming, 5 = Blackman, 6 = Rectangular, 7 = Chebyshev, 8= Gaussian, 9= Kaiser

r (Real Scalar): a real number specifying the redundancy (i.e., overlapping) ratio of any two consecutive data segments: $1.0 \le r \le 2.0$ (1.0 = disjoint segments, 2.0 = totally redundant segments)

OUTPUT

w (Real Vector): a real vector of length N+1 storing the computed cross power spectral density of the input vectors u1 and u2

EXAMPLES

```
//  An example for: csd(v1, v2, i1, i2, r)
//

//  Compute csd(U,V,l,t,r) where U and V are 8-dimensional
//  real vectors, l = 4, t = 3 and r = 1.0:

U = {0; 0.866025403784439;    -0.866025403784439; 0;
                         0.866025403784439;     -0.866025403784439; 0;
                          0.866025403784439};
V = {0.707106781186548;    0.258819045102521;
                  -0.965925826289068;   0.707106781186548;
                     0.258819045102521;  -0.965925826289068;
                     0.707106781186548;    0.258819045102521};
l = 4;
t = 3;
r = 10;

EXcsd = csd(U,V,l,t,r);

//  Results:
//  EXcsd:  5 rows
//     0.000274924142535783
//     0.00785744561501351
//     0.519972168933765
//     0.8331179285605
//     0.084053950515755
```

ALGORITHM AND COMMENTS

This function computes the cross power spectral density using the periodogram method with FFT as follows. For two given real data sequences u1 = $\{u1_0,..., u1_{n-1}\}$ and u2 = $\{u2_0,..., u2_{n-1}\}$ of equal length n = 2N with N an integer of power of 2, their cross power spectral density is represented by a vector w = $(w_0, ..., w_N)$ of length N+1 with

$$w_k = \frac{1}{\sum_{j=0}^{n-1} v_j^2} \left| \text{Real}\left(D1_k D2_k\right) \right|$$

where Real() denotes the real part of a complex number and

$$v = (v_0, \ldots, v_{n-1})$$

represents the weights of specified window of length n+1, which can be obtained by the function getWind(), and

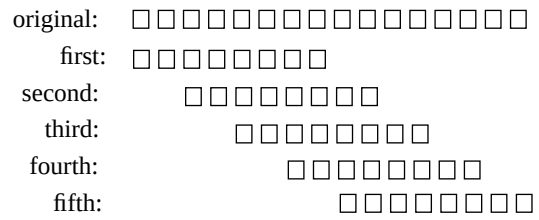
$$D1_k = \sum_{j=0}^{n-1} v_j u1_j e^{\frac{-i2\pi jk}{n}}$$

$$D2_k = \sum_{j=0}^{n-1} v_j u2_j e^{\frac{-i2\pi jk}{n}}$$

are both the results of FFT.

For a pair of real data sequences u and v with the same length n > 2N, a segmentation scheme is used to break u and v into a consecutive equally sized segments with each a length of 2N. The cross power spectral density for each segment of corresponding u and v is computed with FFT as described above. The cross power spectral density of u and v is taken as the average of those of the segments.

For illustrative convience, let us described the segmentation scheme with a simple example on data sequence u. Suppose u is a 16-point data sequence , N = 4, and r = 1.75. It means that r - 1 = 0.75 or 3/4 of the current segment should be contained in the preceding segment. Since the FFT size is 2N (= 8), each segment is comprised of the last 6 points of the preceding segment and 2 points from the unprocessed data. This is illustrated graphically below.



(Identical segmentation is appplied to the data sequence v while the length of v is 16.)

In dealing with cross power spectral density of two data sequences of different length, we append zeros to the

end of the shorter sequence so that it has the same length as the longer one.

REFERENCE

Stearns, S.D. and David, R. A., *Digital Signal Processing Algorithms*, Prentice-Hall, 1988.

■ DFT

FUNCTION

w = DFT(u)

PURPOSE

Compute the discrete Fourier transform (DFT) of a real or complex data sequence of arbitrary length no less than four

INPUT

u (Real or Complex Vector): a real or complex vector of length no less than 4 representing the data sequence

OUTPUT

w (Complex Vector): a complex vector with the same length as the input vector u representing the result of the discrete Fourier transform

EXAMPLES

```
//  Examples for: DFT(v)
//

//  Compute DFT(U) for an 8-dimensional real vector U:

U = { 0;   0.707106781186548; 1;   0.707106781186548;
                     0;  -0.707106781186548; -1;  -0.707106781186548};

EXDFTU = DFT(U);

//  Result :
//  EXDFTU:  8 rows
//    -5.42101086242752e-20 +0i
//     2.11419423634673e-18 -4i
//    -1.58213532772343e-19 +9.75781955236954e-19i
//    -1.46367293285543e-18 -1.34506121518552e-15i
//     0                    -5.7159983545887e-19i
//     2.00577401909818e-18 +1.34506121518552e-15i
//    -1.14847106720592e-18 +2.71050543121376e-18i
//    -7.96888596776846e-18 +4i


//  Compute DFT(V) for an 8-dimensional complex vector V:
```

```
V = {  (1, 0); (0.707106781186548, 0.707106781186548);
                     (0, 1); (-0.707106781186548,0.707106781186548);
                   (-1,0); (-0.707106781186548,-0.707106781186548);
                   (0,-1); (0.707106781186548, -0.707106781186548)};

EXDFTV = DFT(V);

//  Results:
//  EXDFTV:  8 rows
//    -5.42101086242752e-20 -5.42101086242752e-20i
//     8                    +3.84891771232354e-18i
//    -1.57209315010398e-18 -1.0842021724855e-18i
//     0                    -1.30104260698261e-18i
//     5.42101086242752e-19 -1.02999206386123e-18i
//    -2.69012243037103e-15 +2.43945488809238e-18i
//     1.0842021724855e-18  -3.90312782094782e-18i
//     6.88468379528295e-18 -1.30104260698261e-18i
```

ALGORITHM AND COMMENTS

The discrete Fourier transform of a real or complex data sequence u of length N is the vector $w = (w_0, ..., w_{N-1})$ of the same length with

$$w_n = \sum_{k=0}^{N-1} u_k e^{\frac{-2\pi nk}{N} i}$$

where

$$u = (u_0, \ldots, u_{N-1})$$

The factor $2\pi/N$ may be interpreted as a base frequency. The characteristic of the DFT is that if a data sequence consists of samples of a signal at an integer multiple of this base frequency, the DFT output contains a single nonzero point at the location corresponding to this multiple and is zero at all other locations. However, if the frequency of the input signal is a noninteger multiple of this base frequency, the DFT output is nonzero at all locations.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ FFT

FUNCTION

w = FFT(u)

PURPOSE

Compute the one-dimensional discrete fast Fourier transform (FFT) of a real or complex data sequence of length equal to a power of two and no less than four

INPUT

u (Real or Complex Vector): a real or complex vector of length equal to a power of two and no less than four that represents the data sequence

OUTPUT

w (Complex Vector): a complex vector of the same length as the input vector u representing the result of the FFT

EXAMPLES

```
//  Examples for: FFT(v)
//

//  Compute FFT(U) for an 8-dimensional real vector U:

U = { 0;   0.707106781186548; 1;   0.707106781186548;
                     0;  -0.707106781186548; -1;  -0.707106781186548};

EXFFTU = FFT(U);

//  Results:
//  EXFFTU:  8 rows
//     0                     +0i
//     1.89735380184963e-18  -4i
//     0                     +0i
//     1.02999206386123e-18  -1.34484437475102e-15i
//     0                     +0i
//    -1.13841228110978e-18  +1.34527805562001e-15i
//     0                     +0i
//    -1.78893358460108e-18  +4i


//  Compute FFT(V) for an 8-dimensional complex vector V:

V = {(1, 0); (0.707106781186548, 0.707106781186548);
                    (0, 1); (-0.707106781186548,0.707106781186548);
                   (-1,0); (-0.707106781186548,-0.707106781186548);
                    (0,-1); (0.707106781186548, -0.707106781186548)};
```

```
EXFFTV = FFT(V);

//  Results:
//  EXFFTV:  8 rows
//     0                    +0i
//     8                    +3.19839640883224e-18i
//     0                    +0i
//    -3.06658683336675e-19 -3.79470760369927e-19i
//     0                    +0i
//    -2.69055611124003e-15 -2.43945488809238e-18i
//     0                    +0i
//     3.06658683336675e-19 -3.79470760369927e-19i
```

ALGORITHM AND COMMENTS

The function FFT computes the discrete Fourier transform

$$w_n = \sum_{k=0}^{N-1} u_k e^{\frac{-2\pi nk}{N} i}$$

for n = 0, ..., N-1, of a given data sequence u= (u_0, ..., u_{N-1}) using the (fast) Cooley-Tukey algorithm. With the fast algorithm, the discrete Fourier transform reduces the number of arithmetic operations (i.e., multiplications and additions) from an order of N^2 for a standard discrete Fourier transform algorithm to the order of $N\log_2 N$.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., Digital Signal Processing, Prentice Hall, 1975

Bose, N.K., Digital Filters Theory and Applications, Elsevier Science, NewYork, N.Y., 1985

■ FFTn

FUNCTION

w = FFTn(u, L)

PURPOSE

Compute the multi-dimensional discrete fast Fouier transform of a real or complex data sequence of length equal to a power of two and no less than four

INPUT

u (Real or Complex Vector): a real or complex vector of length equal to a power of two and no less than four that represents the multi-dimensional input data sequence. For a N-dimensional discrete transform, if the

input data set is

$$\{u_{n_1, n_2, \ldots, n_N} \mid 0 \le n_j \le L_j - 1, \quad 1 \le j \le N\}$$

then

$$u = \left(u_{0\ldots0}, \ldots, u_{0\ldots0\,(L_N-1)}, u_{0\ldots010}, \ldots, u_{0\ldots0\,(L_{N-1}-1)\;(L_N-1)} \right.$$
$$\left. u_{0\ldots0100}, \ldots, u_{0\ldots0\,(L_{N-2}-1)\;(L_{N-1}-1)\;(L_N-1)}, \ldots, u_{(L_1-1)\;(L_2-1)\ldots(L_N-1)} \right)$$

is a vector of length $L_1L_2...L_N$

L (Integer Scalar): the integer vector (L_1, L_2, ..., L_N)

OUTPUT

w (Complex Vector): a complex vector of length $L_1L_2...L_N$ representing the N-dimensional FFT results which are stored in the same order as that of the input vector u

EXAMPLES

```
//  Examples for: FFTn(v, i)
//

//  Compute FFTn(U,iu) where U is a 8-dimensional real
//  vector and iu is a two-dimensional integer vector:

U = {0; 1;1; 0; 0; -1; -1; 0};
iu = {4; 2};

EXFFTnU = FFTn(U,iu);

//  Results:
//  EXFFTnU:  8 rows
//     0 +0i
//     0 +0i
//     2 -2i
//    -2 -2i
//     0 +0i
//     0 +0i
//     2 +2i
//    -2 +2i


//  Compute FFTn(V,iv) where V is a 8-dimensional complex
//  vector and iv is a two-dimensional integer vector:

V = {(1, 0);(0, 1);(0, 1);(-1, 0);(-1, 0);(0, -1);
                        (0, -1);(1, 0)};
iv = {4; 2};
```

```
EXFFTnV = FFTn(V,iv);

//  Results:
//  EXFFTnV:  8 rows
//     0                    +0i
//     0                    +0i
//     4                    +4i
//     4                    -4i
//     0                    +0i
//     0                    +0i
//     2.60208521396521e-18 -2.60208521396521e-18i
//    -2.60208521396521e-18 -2.60208521396521e-18i
```

ALGORITHM AND COMMENTS

For a N-dimensional data set

$$\{u_{n_1, n_2, \ldots, n_N} \mid 0 \le n_j \le L_j - 1, \quad 1 \le j \le N\}$$

stored in a vector as

$$u = \left(u_{0\ldots0}, \ldots, u_{0\ldots0\,(L_N-1)}, u_{0\ldots010}, \ldots, u_{0\ldots0\,(L_{N-1}-1)\,(L_N-1)} \right.$$
$$\left. u_{0\ldots0100}, \ldots, u_{0\ldots0\,(L_{N-2}-1)\,(L_{N-1}-1)\,(L_N-1)}, \ldots, u_{(L_1-1)\,(L_2-1)\ldots(L_N-1)} \right)$$

of length $L_1L_2...L_N$, the discrete N-dimensional Fourier transform of u is the vector

$$w = \left(w_{0\ldots0}, \ldots, w_{0\ldots0\,(L_N-1)}\, w_{0\ldots010}, \ldots, w_{(L_1-1)\ldots(L_{N-1}-1)\,(L_N-1)} \right)$$

of the same length with

$$w_{n_1, n_2, \ldots, n_N} = \sum_{k_1=0}^{L_1-1} \sum_{k_2=0}^{L_2-1} \cdots \sum_{k_N=0}^{L_N-1} u_{k_1, k_2, \ldots, k_N} e^{-i2\pi \sum_{j=1}^{N} \frac{n_j k_j}{L_j}}$$

The N-dimensional FFT is computed by repeatedly applying the one-dimensional FFT. For detailed information, see the references.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ filter

FUNCTION

y = filter(x, a, b)

PURPOSE

Compute the output of a linear system described by the ratio of two polynomials in z^{-1} which is excited by a finite data sequence

INPUT

x (Real Vector): a real vector storing the input data sequence to be filtered

a (Real Vector): a real vector storing the coefficients of the polynomial in z^{-1} for the numerator of the system function

b (Real Vector): a real vector storing the coefficients of the polynomial in z^{-1} for the denominator of the system function. The first element of b must be unity

OUTPUT

y (Real Vector): a real vector of the same length as the input vector x representing the filtered signal sequence

EXAMPLES

```
//  An example for: filter(v1, v2, v3)
//

//  Compute filter(U,V,W) where U is a 4-dimensional real
//  vector,V is a 5-dimensional real vector, and W is a
//  3-dimensional vector:

U = {1; 2; 3; 4};
V = { 0.5;  0.4;  0.3;  0.2;  0.1};
W = {1; 3; 5};

EXfilter = filter(U,V,W);

//  Results:
//  EXfilter:  4 rows
//        0.5
//       -0.1
//        0.4
//        3.3
```

ALGORITHM AND COMMENTS

The linear, causal, shift-invariant system

$$H_d(z) = \frac{\sum_{k=0}^{M} a_k z^{-k}}{\sum_{j=0}^{N} b_j z^{-j}}$$

is specified by the given input vectors

$$a = (a_0, a_1, \ldots, a_M)$$
$$b = (b_0, b_1, \ldots, b_N)$$

with $b_0 = 1$.

The filtered signal sequence with input sequence x of length L is the real vector $y = (y_0,\ldots,y_{L-1})$ of the same length with

$$y_N = \sum_{j=0}^{\min(n, M)} a_j x_{n-j} - \sum_{k=1}^{\min(n, N)} b_k y_{n-k}$$

where $x = (x_0, \ldots, x_{L-1})$.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ FIR

FUNCTION

h = FIR(N, fl, fh, btype, type)

PURPOSE

Compute the coefficients of frequecy response function for a general finite impulse response (FIR) filter using the windowing technique

INPUT

N (Integer Scalar): an integer specifying the length of the coefficients for the FIR filter

fl (Real Scalar): a real number specifying the normalized lower cutoff frequency ratio, $0 < fl < 0.5$

fh (Real Scalar): a real number specifying the normalized higher cutoff frequency ratio,
$0 < fl < fh \leq 0.5$

btype (Integer Scalar): an integer specifying the band type: 1 = low pass, 2 = high pass, 3 = band pass, 4 = band stop

type (Integer Scalar): an integer specifying the window type: 1 = Bartlett, 2 = Parzen, 3 = Hanning, 4 = Hamming, 5 = Blackman, 6 = Rectangular, 7 = Chebyshev, 8= Gaussian, 9= Kaiser

OUTPUT

h (Real Vector): a real vector of length N storing the computed filter coefficients

EXAMPLES

```
//  An example for: FIR(i1, r1, r2, i2, i3)
//

//  Compute FIR(n,f1,f2,bt,t) for n = 8, f1 = 0.3, f2 = 0.4,
//  bt = 1 and t = 3:

n = 8;
f1 = 0.3;
f2 = 0.4;
bt = 1;
t = 3;

EXFIR = FIR(n,f1,f2,bt,t);

//  Results:
//  EXFIR:  8 rows
//     0
//    -0.0239693836634766
//     0.0400836758323794
//     0.489533905089011
//     0.489533905089011
//     0.0400836758323794
//    -0.0239693836634766
//     0
```

ALGORITHM AND COMMENTS

The frequency response function for a general FIR filter of length N has the form

$$H(e^{i\theta}) = \sum_{k=-(N-1)/2}^{(N-1)/2} h_k e^{-i\theta}$$

The coefficients of the function H, computed by this function are represented by the vector $h = (h_{-(N-1)/2}, h_{-(N-1)/2+1}, ..., h_{(N-1)/2})$ of length N with

Low pass filter:

$$h_k = \begin{cases} \dfrac{\sin(2\pi k f_l)}{\pi k} v_k & \text{if } k \neq 0 \\ 2 v_0 f_l & \text{if } k = 0 \end{cases}$$

High pass filter:

$$h_k = \begin{cases} \dfrac{\sin(\pi k) - \sin(2\pi k f_h)}{\pi k} v_k & \text{if } k \neq 0 \\ v_0 (1 - 2 f_h) & \text{if } k = 0 \end{cases}$$

Band pass filter:

$$h_k = \begin{cases} \dfrac{\sin(2\pi k f_h) - \sin(2\pi k f_l)}{\pi k} v_k & \text{if } k \neq 0 \\ 2 v_0 (f_h - f_l) & \text{if } k = 0 \end{cases}$$

Band stop filter:

$$h_k = \begin{cases} \dfrac{\sin(2\pi k f_l) + \sin(\pi k) - \sin(2\pi k f_h)}{\pi k} v_k & \text{if } k \neq 0 \\ v_0 (1 - 2 f_h) & \text{if } k = 0 \end{cases}$$

where fl and fh denote the normalized lower and higher cutoff frequency ratios, respectively, and

$$v = (v_{-(N-1)/2}, v_{-(N-1)/2+1}, \cdots, v_{(N-1)/2})$$

represents the weights of specified window of length N, which can be obtained by the function getWind().

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ FIRlow

FUNCTION

h = FIRlow(N, fl, type)

PURPOSE

Compute the coefficients of frequency response function for a low pass filter using the windowing technique

INPUT

N (Integer Scalar): an integer specifying the length of the coefficients for the low pass filter

fl (Real Scalar): a real number specifying the normalized lower cutoff frequency ratio, $0 < fl < 0.5$

type (Integer Scalar): an integer specifying the window type: 1 = Bartlett, 2 = Parzen, 3 = Hanning, 4 = Hamming, 5 = Blackman, 6 = Rectangular, 7 = Chebyshev, 8= Gaussian, 9= Kaiser

OUTPUT

h (Real Vector): a real vector of length N storing the computed filter coefficients

EXAMPLES

```
//  An example for: FIRlow(i1, r, i2)
//

//  Compute FIRlow(n,f,t) for n = 8, f = 0.3 and t = 3:

n = 8;
f = 0.3;
t = 3;

EXFIRlow = FIRlow(n,f,t);

//  Results :
//  EXFIR:  8 rows
//     0
//    -0.0239693836634766
//     0.0400836758323794
//     0.489533905089011
//     0.489533905089011
//     0.0400836758323794
//    -0.0239693836634766
//     0
```

ALGORITHM AND COMMENTS

The frequency response function of a low pass filter of length N has the form

$$H(e^{i\theta}) = \sum_{k=-(N-1)/2}^{(N-1)/2} h_k e^{-i\theta}$$

The coefficients of the function H, computed by this function are represented by the vector $h = (h_{-(N-1)/2}, h_{-(N-1)/2+1}, ..., h_{(N-1)/2})$ of length N with

$$h_k = \begin{cases} \dfrac{\sin(2\pi k f_l)}{\pi k} v_k & \text{if } k \neq 0 \\ 2 v_0 f_l & \text{if } k = 0 \end{cases}$$

where fl is the normalized lowercutoff frequency ratio, and

$$v = (v_{-(N-1)/2}, v_{-(N-1)/2+1}, \cdots, v_{(N-1)/2})$$

represents the weights of specified window of length N, which can be obtained by the function getWind().

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ gain

FUNCTION

[u, phi] = gain(a, b, f)

PURPOSE

Compute the gain of a linear, causal, shift-invariant system described by the ratio of two polynomials in z^{-1}

INPUT

a (Real Vector): a real vector storing the coefficients of the polynomial in z^{-1} for the numerator of the given system function

b (Real Vector): a real vector storing the coefficients of the polynomial in z^{-1} for the denominator of the given system function. The first element of b must be unity

f(Real Scalar): a real number representing the frequency at which the gain is to be evaluated, $0 < r \leq 1$

OUTPUT

u (Real Scalar): the magnitude of the gain

phi (Real Scalar): the phase angle of the gain (in degrees)

EXAMPLES

```
//  An example for: gain(v1, v2, r)
//

//  Compute gain (vin1,vin2,f);) where U is a 5-dimensional real
//  vector and V is a 6-dimensional real vector, and f = 0.5:
```

```
U = { 5; 4; 3; 2; 1};
V = {8; 4; 9; 3; 2; 1};
f = 0.5;

[EXgain, EXgainphase] = gain(U,V,f);

//  Results:
//     EXgain:          0.75
//     EXgainphase:   -4.55548196563785e-17
```

ALGORITHM AND COMMENTS

The linear, causal, shift-invariant system

$$H_d(z) = \frac{\sum_{k=0}^{M} a_k z^{-k}}{\sum_{j=0}^{N} b_j z^{-j}} \qquad (1)$$

is specified by the given input vectors

$$a = (a_0, a_1, \ldots, a_M)$$
$$b = (b_0, b_1, \ldots, b_N)$$

with $b_0 = 1$.

The gain of the system at frequency f is the value of (1) at $z=2i\pi f$, i.e.,

$$ue^{\phi i} = H_d(e^{2i\pi f}) = \frac{\sum_{k=0}^{M} a_k d^{-2\pi f k i}}{\sum_{j=0}^{N} b_j e^{-2\pi f j i}}$$

where u is the magnitude and ϕ is the phase angle with $-\pi < \phi \leq \pi$ (or equivalently, $-180 < \phi \leq 180$ in degrees).

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ getWind

FUNCTION

w = getWind(N, type)

PURPOSE

Compute the weights of a window function with specified length of window

INPUT

N (Integer Scalar): an integer specifying the length of the window

type (Integer Scalar): an integer specifying the window type: 1 = Bartlett, 2 = Parzen, 3 = Hanning, 4 = Hamming, 5 = Blackman, 6 = Rectangular, 7 = Chebyshev, 8= Gaussian, 9= Kaiser

OUTPUT

w (Real Vector): a real vector of length N storing the computed weights of the window

EXAMPLES

```
//  An example for: getWind(i1, i2)
//

//  Compute getWind(n,t) for n = 8 and t = 4:

n = 8;
t = 4;

EXgetWind = getWind(n,t);

//  Results:
//  EXgetWind:  8 rows
//     0.08
//     0.253194691144983
//     0.642359629619905
//     0.954445679235113
//     0.954445679235113
//     0.642359629619905
//     0.253194691144983
//     0.08
```

ALGORITHM AND COMMENTS

The weights of various windows of length N computed by this function are represented by the vector $w = (w_{-(N-1)/2}, w_{-(N-1)/2+1}, \ldots, w_{(N-1)/2})$ of length N with

Bartlett (or Triangular) window:

$$w_n = 1 - \frac{2|n|}{N-1}$$

Parzen (or Parabolic) window:

$$w_n = 1 - \left(\frac{2n}{N-1}\right)^2$$

Hanning window:

$$w_n = \frac{1}{2}\left[1 + \cos\left(\frac{2\pi n}{N-1}\right)\right]$$

Hamming window:

$$w_n = 0.54 + 0.46\cos\left(\frac{2\pi n}{N-1}\right)$$

Blackman window:

$$w_n = 0.42 + 0.5\cos\left(\frac{2\pi n}{N-1}\right) + 0.08\cos\left(\frac{4\pi n}{N-1}\right)$$

Rectangular window:

$$w_n = 1$$

Gaussian window:

$$w_n = e^{-\frac{1}{2}\left(\frac{2n}{N-1}\right)^2}$$

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ iCosFT

FUNCTION

w = iCosFT(u)

PURPOSE

Compute the inverse discrete cosine transform of a real data sequence of length equal to a power of two and no less than eight

INPUT

u (Real Vector): a real vector of length equal to a power of two and no less than eight that represents the data sequence

OUTPUT

w (Real Vector): a real vector with the same length as the input vector u representing the result of the inverse discrete cosine transform

EXAMPLES

```
//  An example for: iCosFT(v)
//

//  Compute iCosFT(U) for an 8-dimensional real vector U:

U = { 0; 1; 4; 1; 0; 1; 0; 1};

EXiCosFT = iCosFT(U);

//  Results:
//   EXiCosFT:  8 rows
//     1
//     0.707106781186548
//    -5.93600689435814e-18
//    -0.707106781186548
//    -1
//    -0.707106781186548
//    -5.55653613398821e-18
//     0.707106781186548
```

ALGORITHM AND COMMENTS

The inverse discrete cosine transform of a real data sequence u of length N is the vector $w= (w_0, ..., w_{N-1})$ of the same length satisfying

$$u_n = \sum_{k=0}^{N-1} w_k \cos\left(\frac{\pi n k}{N}\right) \qquad (1)$$

where $8 \le N = 2P$, $p \ge 3$, and

$$u = (u_0, \ldots, u_{N-1})$$

To speed up the computation indicated in (1), the FFT algorithm is used. For detailed information, see the references.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ iDFT

FUNCTION

w = iDFT(u)

PURPOSE

Compute the inverse discrete Fourier transform (DFT) of a data sequence of arbitrary length no less than four

INPUT

u (Real or Complex Vector): a real or complex vector of length no less than four representing the data sequence

OUTPUT

w (Complex Vector): a complex vector with the same length as the input vector u representing the result of the inverse DFT

EXAMPLES

```
//  Examples for: iDFT(v)
//

//  Compute iDFT(U) for an 8-dimensional real vector U:

U = { 0; -4; 0; 0; 0; 0; 0; 4};

EXiDFTU = iDFT(U);

//  Results:
//  EXiDFTU:  8 rows
//     0                    +0i
//    -4.60785923306339e-19 -0.707106781186548i
//    -5.96311194867027e-19 -1i
//    -4.60785923306339e-19 -0.707106781186548i
//     0                    +8.13151629364128e-19i
//     4.60785923306339e-19 +0.707106781186548i
//     3.14418630020796e-18 +1i
//     2.90024081139872e-18 +0.707106781186548i
```

```
//  Compute iDFT(V) for an 8-dimensional complex vector V:

V = {(0, 0);(8, 0);(0,0);(0,0);(0,0);(0,0);(0,0);(0,0)};

EXiDFTV = iDFT(V);

//  Results:
//  EXiDFTV:  8 rows
//     1                    +0i
//     0.707106781186548    +0.707106781186548i
//     1.89735380184963e-19 +1i
//    -0.707106781186548    +0.707106781186548i
//    -1                    +3.79470760369927e-19i
//    -0.707106781186548    -0.707106781186548i
//    -6.7762635780344e-19  -1i
//     0.707106781186548    -0.707106781186548i
```

ALGORITHM AND COMMENTS

The inverse DFT of a given real or complex data sequence u of length N is the vector $w = (w_0, ..., w_{N-1})$ of the same length with

$$w_n = \frac{1}{N}\sum_{k=0}^{N-1} u_k e^{\frac{2\pi nk}{N}i}$$

where

$$u = (u_0, \ldots, u_{N-1})$$

The factor $2\pi/N$ may be interpreted as a base "frequency". The characteristic of the inverse DFT is that if a sequence consists of samples of a signal at an integer multiple of this base "frequency", the inverse DFT output contains a single nonzero point at a location corresponding to this multiple and is zero at all other locations. However, if the "frequency" of the input signal is a noninteger multiple of this base "frequency", the inverse DFT output is nonzero at all locations.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ iFFT

FUNCTION

w = iFFT(u)

PURPOSE

Compute the one-dimensional inverse discrete FFT (fast Fourier transform) of a real or complex data sequence

INPUT

u (Real or Complex Vector): a real or complex vector of length equal to a power of two and no less than four that represents the data sequence

OUTPUT

w (Complex Vector): a complex vector of the same length as the input vector u representing the result of the inverse FFT

EXAMPLES

```
//  Examples for: iFFT(v)
//

//  Compute iFFT(U) for an 8-dimensional real vector U:

U = { 0; -4; 0; 0; 0; 0; 0; 4};

EXiFFTU = iFFT(U);

//  Results:
//  EXiFFTU:  8 rows
//     0                    +0i
//    -5.42101086242752e-19 -0.707106781186548i
//    -1.89735380184963e-19 -1i
//    -5.42101086242752e-19 -0.707106781186548i
//     0                    +0i
//     5.42101086242752e-19 +0.707106781186548i
//     1.89735380184963e-19 +1i
//     5.42101086242752e-19 +0.707106781186548i


//  Compute FFT(V) for an 8-dimensional complex vector V:

V = {(0,0);(8,0);(0,0);(0,0);(0,0);(0,0);(0,0);(0,0)};

EXiFFTV = iFFT(V);

//  Results:
```

```
//  EXiFFTV:  8 rows
//     1                    +0i
//     0.707106781186548    +0.707106781186548i
//     1.89735380184963e-19 +1i
//    -0.707106781186548    +0.707106781186548i
//    -1                    +0i
//    -0.707106781186548    -0.707106781186548i
//    -1.89735380184963e-19 -1i
//     0.707106781186548    -0.707106781186548i
```

ALGORITHM AND COMMENTS

The function iFFT computes the inverse discrete Fourier transform

$$w_n = \frac{1}{N}\sum_{k=0}^{N-1} u_k e^{\frac{2\pi nk}{N}i}$$

for n = 0, ..., N-1, of a given data sequence u= (u_0, ..., u_{N-1}) using the (fast) Cooley-Tukey algorithm.

With the fast algorithm, the inverse discrete Fourier transform reduces the number of arithmetic operations (i.e., multiplications and additions) from an order of N^2 for a standard inverse DFT algorithm to the order of $N\log_2 N$.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ iFFTn

FUNCTION

w = iFFTn(u, L)

PURPOSE

Compute the multi-dimensional discrete inverse fast Fouier transform of a real or complex data sequence of length equal to a power of two and no less than four

INPUT

u (Real or Complex Vector): a real or complex vector of length equal to a power of two and no less than four that represents the multi-dimensional input data sequence. For a N-dimensional discrete inverse transform, if the input data set is

$$\{u_{n_1, n_2, \ldots, n_N} \mid 0 \le n_j \le L_j - 1, \quad 1 \le j \le N\}$$

then

$$u = \left(u_{0\ldots0}, \ldots, u_{0\ldots0\,(L_N-1)}, u_{0\ldots010}, \ldots, u_{0\ldots0\,(L_{N-1}-1)\;(L_N-1)} \right.$$
$$\left. u_{0\ldots0100}, \ldots, u_{0\ldots0\,(L_{N-2}-1)\;(L_{N-1}-1)\;(L_N-1)}, \ldots, u_{(L_1-1)\;(L_2-1)\ldots(L_N-1)} \right)$$

is a vector of length $L_1L_2...L_N$

L (Integer Scalar): the integer vector (L_1, L_2, ..., L_N)

OUTPUT

w (Complex Vector): a complex vector of length $L_1L_2...L_N$ representing the N-dimensional inverse FFT results which are stored in the same order as that of the input vector u

EXAMPLES

```
//  Examples for: iFFTn(v, i)
//

//  Compute iFFTn(U,iu) where U is a 8-dimensional real
//  vector and iu is a two-dimensional integer vector:

U = {0; 0; 2; -2; 0; 0; 2; -2};
iu = {4; 2};

EXiFFTnU = iFFTn(U,iu);

//  Results:
//  EXiFFTnU:  8 rows
//                        0 +0i
//                        1 +0i
//                        0 +0i
//                        0 +0i
//                        0 +0i
//                       -1 +0i
//                        0 +0i
//                        0 +0i


//  Compute iFFTn(V,iv) where V is a 8-dimensional complex
//   vector and iv is a two-dimensional integer vector:

V = {(0, 0);(0, 0);(4, 4);(4, -4);(0, 0);(0, 0);
                    (0, 0);(0, 0)};
iv = {4; 2};

EXiFFTnV = iFFTn(V,iv);

//  Results:
//  EXiFFTnV:  8 rows
```

```
//     1                     +0i
//     0                     +1i
//     1.30104260698261e-18  +1i
//    -1                     +1.30104260698261e-18i
//    -1                     +0i
//     0                     -1i
//    -1.30104260698261e-18  -1i
//     1                     -1.30104260698261e-18i
```

ALGORITHM AND COMMENTS

For a N-dimensional data set

$$\{u_{n_1, n_2, \ldots, n_N} \mid 0 \le n_j \le L_j - 1, \quad 1 \le j \le N\}$$

stored in a vector as

$$u = \left(u_{0\ldots0}, \ldots, u_{0\ldots0\,(L_N-1)}, u_{0\ldots010}, \ldots, u_{0\ldots0\,(L_{N-1}-1)\,(L_N-1)} \right.$$
$$\left. u_{0\ldots0100}, \ldots, u_{0\ldots0\,(L_{N-2}-1)\,(L_{N-1}-1)\,(L_N-1)}, \ldots, u_{(L_1-1)\,(L_2-1)\ldots(L_N-1)} \right)$$

of length $L_1L_2...L_N$, the discrete N-dimensional Fourier transform of u is the vector

$$w = \left(w_{0\ldots0}, \ldots, w_{0\ldots0\,(L_N-1)}\, w_{0\ldots010}, \ldots, w_{(L_1-1)\ldots(L_{N-1}-1)\,(L_N-1)} \right)$$

of the same length with

$$w_{n_1, n_2, \ldots, n_N} = \frac{1}{L_1 \ldots L_n} \sum_{k_1=0}^{L_1-1} \sum_{k_2=0}^{L_2-1} \cdots \sum_{k_N=0}^{L_N-1} u_{k_1, k_2, \ldots, k_N} e^{i2\pi \sum_{j=1}^{N} \frac{n_j k_j}{L_j}}$$

The N-dimensional inverse FFT is computed by repeatedly applying the one-dimensional inverse FFT. For detailed information, see the references.

REFERENCES

Oppenheim, A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ iSinFT

FUNCTION

w = iSinFT(u)

PURPOSE

Compute the inverse discrete sine transform of a real data sequence of length equal to a power of two and no less than eight

INPUT

u (Real Vector): a real vector of length equal to a power of two and no less than eight that represents the data sequence

OUTPUT

w (Real Vector): a real vector with the same length as the input vector u representing the result of the inverse discrete sine transform

EXAMPLES

```
//  An example for: iSinFT(v)
//

//  Compute  iSinFT(U) for an 8-dimensional real vector U:

U = { 0; 0; 4; 0; 0; 0; 0; 0};

EXiSinFT = iSinFT(U);

//  Results:
//  EXiSinFT:  8 rows
//    -0
//     0.707106781186548
//     1
//     0.707106781186548
//     0
//    -0.707106781186548
//    -1
//    -0.707106781186548
```

ALGORITHM AND COMMENTS

The inverse discrete sine transform of a real data sequence u of length N is the vector $w= (w_0, ..., w_{N-1})$ of the same length with

$$w_n = \frac{2}{N}\sum_{k=1}^{N-1} u_k \sin\left(\frac{\pi nk}{N}\right) \quad (1)$$

where $8 \le N = 2^p$, $p \ge 3$, and

$$u = (u_0, \ldots, u_{N-1})$$

To speed up the computation indicated in (1), the FFT algorithm is used. For detailed information, see the references.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ iTwoRealFFT

FUNCTION

[v1, v2] = iTwoRealFFT(u1, u2)

PURPOSE

Compute the inverse discrete fast Fourier transforms of two real data sequences of the same length (of a power of two) simultaneously

INPUT

u1 (Real Vector): a real vector of length equal to a power of two and no less than four that represents the first data sequence

u2 (Real Vector): a real vector of the same length as u1 representing the second data sequence

OUTPUT

v1 (Complex Vector): a complex vector of the same length as u1 storing the inverse FFT output of u1

v2 (Complex Vector): a complex vector of the same length as u2 storing the inverse FFT output of u2

EXAMPLES

```
//  An example for: iTwoRealFFT(v1, v2)
//

//  Compute iTwoRealFFT(U,V) where U and V are both
//  8-dimensional real vectors :
```

```
U = {0; 0; 4; 0; 0; 0; 4; 0};
V = {0; 0; 0; 4; 0; 4; 0; 0};

[EXiTwoRealFFTVec1, EXiTwoRealFFTVec2] = iTwoRealFFT(U,V);

//  Results:
//  EXiTwoRealFFTVec1:  8 rows
//     1                    +0i
//    -5.42101086242752e-19 +      8.13151629364128e-20i
//    -1                    +0i
//     5.42101086242752e-19 +8.13151629364128e-20i
//     1                    -0i
//     5.42101086242752e-19 -8.13151629364128e-20i
//    -1                    -0i
//    -5.42101086242752e-19 -8.13151629364128e-20i
//
//  EXiTwoRealFFTVec2:  8 rows
//     1                    +0i
//    -0.707106781186548    -0i
//     0                    -0i
//     0.707106781186548    -0i
//    -1                    +0i
//     0.707106781186548    +0i
//     0                    +0i
//    -0.707106781186548    +0i
```

ALGORITHM AND COMMENTS

The inverse discrete Fourier transform of two real data sequences u1 and u2 of the same length N are the complex vectors v1 = ($v1_0$,...,$v1_{N-1}$) and v2 = ($v2_0$,...,$v2_{N-1}$) of length N, respectively, with

$$v1_n = \frac{1}{N}\sum_{k=0}^{N-1} u1_k e^{\frac{2\pi kn}{N}i}$$

and

$$v2_n = \frac{1}{N}\sum_{k=0}^{N-1} u2_k e^{\frac{2\pi kn}{N}i}$$

where

$$u1 = (u1_0, \ldots, u1_{N-1})$$
$$u2 = (u2_0, \ldots, u2_{N-1})$$

The function iTwoRealFFT() behaves as if it were two separate calls of the function iFFT() with two complex data sets in which the real parts of the results are identically zero. This condition is satisfied if the upper halves of u1 and u2 are complex conjugates of their respective lower halves, not counting the first and the N/2 + 1st points, where N is the length of u1 or u2. The output appears as two separate complex data sets, each in the same format as for the function iFFT().

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ psd

FUNCTION

w = psd(u, N, type, r)

PURPOSE

Compute the power spectral density of a real data sequence using the periodogram method with FFT

INPUT

u (Real Vector): a real vector representing the given data sequence

N (Integer Scalar): an integer equal to a power of two and no less than 4 that specifies the half length of a FFT to be performed on segments of u

type (Integer Scalar): an integer specifying the window type: 1 = Bartlett, 2 = Parzen, 3 = Hanning, 4 = Hamming, 5 = Blackman, 6 = Rectangular, 7 = Chebyshev, 8= Gaussian, 9= Kaiser

r (Real Scalar): a real number specifying the redundancy (i.e., overlapping) ratio of any two consecutive data segments: 1.0 = disjoint segments, 2.0 = totally redundant segments

OUTPUT

w (Real Vector): a real vector of length N+1 storing the computed power spectral density of the input vector u

EXAMPLES

```
//  An example for: psd(v, i1, i2, r)
//

//  Compute psd(U,l,t,r) where U is an 8-dimensional
//  real vector, l = 4, t = 3 and r = 1.0:

U = {0; 0.866025403784439; -0.866025403784439; 0;
                    0.866025403784439; -0.866025403784439; 0;
                    0.866025403784439};
l = 4;
t = 3;
```

```
r = 10;

EXpsd = psd(U,l,t,r);

//  Results:
//  EXpsd:  5 rows
//     0.000919914110089357
//     0.0116432188134525
//     0.744638347648319
//     1.12408008588991
//     0.28125
```

ALGORITHM AND COMMENTS

This function computes the power spectral density using the periodogram method with FFT as follows. For a given real data sequence $u = \{u_0,..., u_{n-1}\}$ of length both $n = 2N$ with N an integer of power of 2, the power spectral density is represented by a vector $w = (w_0, ..., w_N)$ of length N+1 with

$$w_k = \frac{1}{\sum_{j=0}^{n-1} v_j^2} |D_k|^2$$

where

$$v = (v_0, \ldots, v_n)$$

represents the weights of specified window of length n+1, which, for example, can be obtained by the function getWind(), and

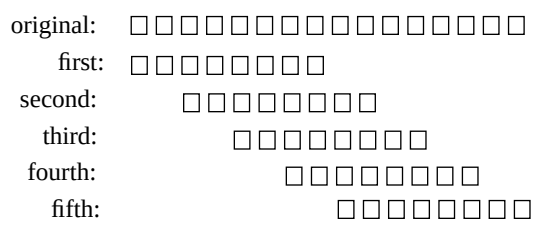
$$D_k = \sum_{j=0}^{n-1} u_j v_j e^{\frac{-i2\pi jk}{n}}$$

are the results of FFT.

For a given real data sequence u of length $n > 2N$, the following segmentation scheme is applied first on consecutive segments to obtain the power spectral density for each segment. The power spectral density of u is then computed as the average of those of the segments.

Let us illustrate the segmentation of an input data sequence of length larger than 2N. Assume u is a 16-point data set , N = 4, and r = 1.75. It means that r - 1 = 0.75 or 3/4 of the current segment should be contained in the preceding segment. Since the FFT size is 2N (= 8), each segment is comprised of the last 6 points of the

preceding segment and 2 points from the unprocessed data. This is illustrated graphically below.



REFERENCE

Stearns, S.D. and David, R. A., *Digital Signal Processing Algorithms*, Prentice-Hall, 1988.

■ realFFT

FUNCTION

w = realFFT(u)

PURPOSE

Compute the fast Fourier transform of a real data sequence in compact storage mode

INPUT

u (Real Vector): a real vector of length equal to a power of two and no less than eight that represents the data sequence

OUTPUT

w (Real Vector): a real vector of the same length as the input vector u which stores the real and imaginary parts of the Fourier transform

EXAMPLES

```
//  An example for: realFFT(v)
//

//  Compute realFFT(U) where U is an  8-dimensional real vector :

U = {1; 0; -1; 0; 1; 0; -1; 0};

EXrealFFT = realFFT(U);

//  Results:
//  EXrealFFT:  8 rows
//                        0
//                        0
```

```
//                    0
//                    0
//                    4
//                   -0
//                    0
//                   -0
```

ALGORITHM AND COMMENTS

When the data to be processed by FFT algorithm is purely real, computational time can be saved by stuffing the even-numbered input data points into the real part memory locations, and the odd-numbered data points into the imaginary part memory locations, respectively. An FFT of half the size of the input data points is taken on the "made-up" complex input data. The result can be reshuffled to obtain the FFT output corresponding to the original input real data sequence. The computational time saved in this way increases with the length of the input real data sequence.

Also, it is known that the Fourier transform of a real data sequence has a complex conjugate symmetry property. By taking the advantage of symmetry, the storage required for the output of FFT can be saved. The function, realFFT(), implements a compact storage scheme for storing the output in the following manner.

Let $u = (u_0,..., u_{N-1})$ be the input data sequence of length $N = 2^p$ with $p \geq 3$. The output real vector $w = (w_0, ..., w_{N-1})$ represents the (complex-numbered) results of the Fourier transform
$v= \{v_0, ..., v_{N-1})$ that satisfies

$$v_n = w_{2n} + iw_{2n+1}$$

and

$$v_{N-n} = v^*_n$$

for n = 1, ..., N/2-1 with $v_0 = w_0$, $v_{N/2} = w_1$. Here the asterisk * denotes the complex conjugate.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ response

FUNCTION

y = response(x, a, b, m)

PURPOSE

Compute the time response function of a system described by a ratio of two polynomials in z^{-1} which is excited by a finite data sequence

INPUT

x (Real Vector): a real vector storing the input data sequence

a (Real Vector): a real vector storing the coefficients of the polynomial in z^{-1} for the numerator of the system function

b (Real Vector): a real vector storing the coefficients of the polynomial in z^{-1} for the denominator of the system function. The first element of b must be unity

m (Integer Scalar): a integer specified the length of the desired output points

OUTPUT

y (Real Vector): a real vector of length m representing the system response

EXAMPLES

```
//  An example for: response(v1, v2, v3, i)
//

//  Compute response(U,V,W,n) where U is a 2-dimensional real
//  vector,V is a 5-dimensional real vector, and W is a
//  3-dimensional vector and n = 10:

U = {1; 2};
V = { 0.5; 0.4; 0.3; 0.2; 0.1};
W = {1; 3; 5};
n = 10;

EXresponse = response(U,V,W,n);

//  Results:
//  EXresponse:  10 rows
//                        0.5
//                       -0.1
//                       -0.1
//                        3.4
//                       -6.8
//                        6.4
//                       17.8
//                      -82.4
//                      161.2
//                      -68.6
```

ALGORITHM AND COMMENTS

The linear, causal, shift-invariant system

$$H_d(z) = \frac{\sum_{k=0}^{M} a_k z^{-k}}{\sum_{j=0}^{N} b_j z^{-j}}$$

is specified by the given input vectors

$$a = (a_0, a_1, \ldots, a_M)$$
$$b = (b_0, b_1, \ldots, b_N)$$

with $b_0 = 1$.

The system response with the specified length of output, m, and input sequence x is the real vector $y = (y_0,\ldots,y_{m-1})$ with

$$y_n = \sum_{j=0}^{\min(n, M)} a_j x_{n-j} - \sum_{k=1}^{\min(n, N)} b_k y_{n-k}$$

where $x = (x_0, \ldots, x_{L-1})$ and $x_p = x_{L-1}$ for $p > L-1$.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ sinFT

FUNCTION

w = sinFT(u)

PURPOSE

Compute the discrete sine transform of a real data sequence of length equal to a power of two and no less than eight

INPUT

u (Real Vector): a real vector of length equal to a power of two and no less than eight that represents the data sequence

OUTPUT

w (Real Vector): a real vector with the same length as the input vector u representing the result of the discrete sine transform

EXAMPLES

```
//  An example for: sinFT(v)
//

//  Compute  sinFT(U) for an 8-dimensional real vector U:

U = { 0;   0.707106781186548; 1;   0.707106781186548;
                      0;  -0.707106781186548; -1;  -0.707106781186548};

EXsinFT = sinFT(U);

//  Results:
//  EXsinFT:  8 rows
//     -0
//     -0
//      4
//     -1.19262238973405e-18
//      0
//     -1.19262238973405e-18
//      1.34560331627176e-15
//     -0
```

ALGORITHM AND COMMENTS

The discrete sine transform of a given data sequence u of length N is the vector $w = (w_0, ..., w_{N-1})$ of the same length with

$$w_n = \sum_{k=1}^{N-1} u_k \sin\left(\frac{\pi nk}{N}\right) \qquad (1)$$

where $8 \le N = 2^p$, $p \ge 3$, and

$$u = (u_0, \ldots, u_{N-1})$$

To speed up the computation indicated in (1), the FFT algorithm is used. For detailed information, see the references.

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ twoRealFFT

FUNCTION

[v1, v2] = twoRealFFT(u1, u2)

PURPOSE

Compute the discrete fast Fourier transforms of two real data sequences of the same length (of a power of two) simultaneously

INPUT

u1 (Real Vector): a real vector of length equal to a power of two and no less than four that represents the first data sequence

u2 (Real Vector): a real vector of the same length as u1 representing the second data sequence

OUTPUT

v1 (Complex Vector): a complex vector of the same length as u1 storing the FFT output of u1

v2 (Complex Vector): a complex vector of the same length as u2 storing the FFT output of u2

EXAMPLES

```
//  An example for: twoRealFFT(v1, v2)
//

//  Compute  twoRealFFT(U,V) where U and V are both
//  8-dimensional real vectors :

U = {1; 0; -1; 0; 1; 0; -1; 0};
V = {1;  -0.707106781186548;  0;   0.707106781186548;
                -1;  0.707106781186548;   0;  -0.707106781186548};

[EXtwoRealFFtVec1, EXtwoRealFFTVec2] = twoRealFFT(U,V);

//  Results:
//  EXtwoRealFFtVec1:  8 rows
//     0                    +0i
//     1.46367293285543e-18 +2.16840434497101e-19i
//     4                    +0i
//    -1.46367293285543e-18 +2.16840434497101e-19i
//     0                    -0i
//    -1.46367293285543e-18 -2.16840434497101e-19i
//     4                    -0i
//     1.46367293285543e-18 -2.16840434497101e-19i
//
//  EXtworeallFFTVec2:  8 rows
//     0                    +0i
//    -1.34506121518552e-15 +5.42101086242752e-20i
//     0                    -0i
```

```
//     4                    +5.42101086242752e-20i
//     0                    +0i
//     4                    -5.42101086242752e-20i
//     0                    +0i
//    -1.34506121518552e-15 -5.42101086242752e-20i
```

ALGORITHM AND COMMENTS

The discrete Fourier transform of two real data sequences u1 and u2 of the same length N are the complex vectors v1 = ($v1_0$,...,$v1_{N-1}$) and v2 = ($v2_0$,...,$v2_{N-1}$) of length N, respectively, with

$$v1_n = \sum_{k=0}^{N-1} u1_k e^{\frac{-2\pi kn}{N} i}$$

and

$$v2_n = \sum_{k=0}^{N-1} u2_k e^{\frac{-2\pi kn}{N} i}$$

where

$$u1 = (u1_0, \ldots, u1_{N-1})$$
$$u2 = (u2_0, \ldots, u2_{N-1})$$

The function twoRealFFT() behaves as if it were two separate calls of the function FFT() with two complex data sets in which the real parts of the results are identically zero. This condition is satisfied if the upper halves of u1 and u2 are complex conjugates of their respective lower halves, not counting the first and the N/2 + 1st points, where N is the length of u1 or u2. The output appears as two separate complex data sets, each in the same format as for the function FFT().

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ window

FUNCTION

w =window(u, type)

PURPOSE

Compute the component-by-component product of a real data sequence with corresponding window weights

INPUT

u (Real Vector): a real vector representing the data sequence

type (Integer Scalar): an integer specifying the window type: 1 = Bartlett, 2 = Parzen, 3 = Hanning, 4 = Hamming, 5 = Blackman, 6 = Rectangular, 7 = Chebyshev, 8= Gaussian, 9= Kaiser

OUTPUT

w (Real Vector): a real vector of the same length as the input vector u storing the computed component-by-component product of u and the corresponding window weights

EXAMPLES

```
//  An example for: window(v, i)
//
//  Compute window(U,t) for an 8-dimensional real vector
//   and t = 3:

U = {1;1;1;1;1;1;1;1};
t = 3;

EXwindow = window(U,t);

//  Results:
//  EXwindow:  8 rows
//     0
//     0.188255099070633
//     0.611260466978157
//     0.95048443395121
//     0.95048443395121
//     0.611260466978157
//     0.188255099070633
//     0
```

ALGORITHM AND COMMENTS

For a given real data sequence u of length N, the output vector $w = (w_0, ..., w_{N-1})$ is of the same length with

$$w_n = v_n u_n$$

where

$$u = (u_0, \ldots, u_{N-1})$$

$$v = (v_0, \ldots, v_{N-1})$$

is the vector containing the weights of the specified window (of length N) generated by the function getWind().

REFERENCES

Oppenheim, A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ winSum

FUNCTION

s = winSum(N, type)

PURPOSE

Compute the sum of squares of the first N-1 weights for a window of length N

INPUT

N (Integer Scalar): an integer specifying the length of the window

type (Integer Scalar): an integer specifying the window type: 1 = Bartlett, 2 = Parzen, 3 = Hanning, 4 = Hamming, 5 = Blackman, 6 = Rectangular, 7 = Chebyshev, 8= Gaussian, 9= Kaiser

OUTPUT

s (Real Scalar): the computed sum of squares of the first N-1 weights for the specified window of length N

EXAMPLES

```
//  An example for: winSum(i1, i2)
//
//  Compute  winSum(n,t) for n = 8 and t = 3:

n = 8;
t = 3;

EXwinSum = winSum(n,t);

//  Result :
//    EXwinSum:   2.625
```

ALGORITHM AND COMMENTS

The output of this function, s, is such that

$$s = \sum_{n=0}^{N-2} v_n^2$$

where the vector v = (v_0,..., v_{N-1}) of length N containing the weights of the desired window which is generated by the function getWind().

REFERENCES

Oppenheim, A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

■ zTrans

FUNCTION

[v, w] = zTrans(a, α)

PURPOSE

Compute the z-transform of real coefficients for a complex number representing in its polar form

INPUT

a (Real Vector): a real vector storing the coefficients of a polynomial in z^{-1} that represents the z-transform

α (Complex Scalar): a complex number for which the real part is the magnitude and the imaginary part is the phase angle in radians at which the z-transform is to be taken. Suppose α = r+i c , the corresponding z value is z= r e^{ic}

OUTPUT

v (Complex Scalar): the z-transform result in Cartesian form

w (Complex Scalar): the z-transform result in polar form for which the real part is the magnitude and the imaginary part is the phase angle in radians

EXAMPLES

```
//  Examples for: zTrans(v, a)
//

//  Compute  zTrans(U,z) for an 5-dimensional real vector
//  and z = 1.0:

U= {1; 4; 3; 2; 1};
z = 1.0;

[EXzTransU1,EXzTransU2] = zTrans(U,z);

//  Results:
//           EXzTransU1:    11 +       0i
//           EXzTransU2:    11 +       0i
```

```
//  Compute  zTrans(V,z) for an 4-dimensional complex vector
//  and z = 1.0:

V= {(1,1); (4, 4); (3, 3); (2, 2)};
z = 1.0;

[EXzTransV1,EXzTransV2] = zTrans(V,z);

//  Results:
//            EXzTransV1:   10              + 10i
//            EXzTransV2:   14.142135623731 + 0.785398163397448i
```

ALGORITHM AND COMMENTS

The z-transform, described by the given real vector a = (a_0, ..., a_{N-1}), at the input α = r+i c is

$$v = v_r + iv_r = \sum_{n=0}^{N-1} a_n z^{-n}$$

where $z = r\, e^{ic}$.

The polar form of v which stored in the complex number w = w_r + i w_i on output is such that

$$w_r = \sqrt{v_r^2 + v_i^2}$$

and

$$w_i = \tan^{-1}\left(\frac{v_i}{v_r}\right)$$

REFERENCES

Oppenheim,A.V. & Schafer, R.W., *Digital Signal Processing*, Prentice Hall, 1975

Bose, N.K., *Digital Filter Theory and Applications*, Elsevier Science, NewYork, N.Y., 1985

CHAPTER 19
STATISTICAL ANALYSIS FUNCTIONS

■ avg

FUNCTION

y = avg(u)

PURPOSE

Compute the average (or arithmetic mean) of a given set of n data points

INPUT

u (Real Vector): the n-dimensional vector containing the n data values $u_1, \ldots, u_n$

OUTPUT

y (Real Scalar): the computed average of $u_1, \ldots, u_n$

EXAMPLES

```
//  Example for avg(u), the average of the data set
//  contained in u

//  The data vector is:
u = {8; 3; 5; 12; 10};
y = avg(u);

//  Result :
//      y:    7.6
```

ALGORITHM AND COMMENTS

The average of $u_1, \ldots, u_n$ is defined by

$$u_{avg} = \frac{1}{n}\sum_{i=1}^{n} u_i$$

REFERENCES

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 65

■ avgDev

FUNCTION

y = avgDev(u)

PURPOSE

Compute the average (or mean) deviation of a set of n data points

INPUT

u (Real Vector): the n-dimensional vector containing the n data values $u_1, \ldots, u_n$

OUTPUT

y (Real Scalar): the computed average deviation of $u_1, \ldots, u_n$

EXAMPLES

```
//  Example for avgDev(u), the average deviation of the
//  data set in u.
//
//  The data vector is:
u = {8; 3; 5; 12; 10};
y = avgDev(u);

//  Result :
//      y:     2.88
```

ALGORITHM AND COMMENTS

The average deviation of $u_1, \ldots, u_n$ is defined by

$$\text{avgDev}(u) = \frac{1}{n}\sum_{i=1}^{n}\left|u_i - u_{avg}\right|$$

where u_{avg} is the average of $u_i, \ldots, u_n$

REFERENCE

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 87

■ betaDF

FUNCTION

y = betaDF(a, b, x)

PURPOSE

Compute the density function of the beta distribution with parameters a, b at x:

$$\text{betaf}(a, b, x) = \frac{1}{B(a, b)} x^{a-1} (1-x)^{b-1}$$

for $1 \le a, b < \infty$ and $0 \le x \le 1$ where B(,) denotes the beta function

INPUT

a (Real Scalar): the first parameter for the density function of the beta distribution

b (Real Scalar): the second parameter for the density function of the beta distribution

x (Real Scalar): the point where the density function of beta distribution is evaluated

OUTPUT

y (Real Scalar): the evaluated density function at x

EXAMPLES

```
//  Example for betaDF(a, b, x), the beta distribution
//  density function.

//  The parameters are: a = 1.5, b = 1; the evaluation
//  point is x = 0.5:

a = 1.5;
b = 1;
x = 0.5;
y = betaDF(a,b,x);

//  Result :
//      y:          1.06066017177982
```

ALGORITHM AND COMMENTS

Domain is $0 \le x \le 1$. Parameters have to satisfy the conditions $a \ge 1$ and $b \ge 1$.

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p.930

■ betaDist

FUNCTION

z = betaDist(a, x, y)

PURPOSE

Compute the beta distribution function of x, y with parameter a:

$$\text{beta}(a, x, y) = \frac{\Gamma(x+y)}{\Gamma(x)\,\Gamma(y)} \int_0^a z^{x-1}(1-z)^{y-1}dz$$

where Γ() denotes the gamma function

INPUT

a (Real Scalar): the argument of the distribution function

x (Real Scalar): the first parameter of the beta distribution function

y (Real Scalar): the second parameter of the beta distribution function

OUTPUT

z (Real Scalar): the value of the beta distribution function

EXAMPLES

```
//  Example for betaDist(a, x, y), the beta distribution
//  function.
//  The distribution parameters are: x = 1 and y = 2;
//  the evaluation parameter is a = 0.5:

a = 0.5;
x = 1;
y = 2;
z = betaDist(a,x,y);

//  Result :
//     z:       0.75
```

ALGORITHM AND COMMENTS

Domain: $0 < x, y < \infty$; $0 \le a \le 1$; Range: $0 \le z \le 1$

The algorithm for this function uses its definition in terms of the incomplete beta integral function.

■ bin

FUNCTION

y = bin(m,n)

PURPOSE

Compute the binomial coefficient of m and n:

$$\text{bin}(m, n) = \frac{m!}{n!\,(m-n)!}$$

INPUT

m (Integer Scalar): the first or upper element of the binomial coefficient

n (Integer Scalar): the second or lower element of the binomial coefficient

OUTPUT

y (Real Scalar): the computed value of the binomial coefficien.

EXAMPLES

```
//  Example for bin(m, n), the binomial coefficient of m and n.

//  The parameters are m = 8 and n = 4:

m = 8;
n = 4;
y = bin(m, n);

//  Result :
//    y:     70
```

ALGORITHM AND COMMENTS

Domain is $0 \le n \le m$.

An algorithm is implemented which computes the natural logarithm of the factorial (lnfact()) to a high degree of precision, then exponentiation is used to compute the coefficient value. The formula used is:

floor(1/2 + exp(lnfact(m) - lnfact(n) - lnfact(m -n))),

using a high precision form of the function lnfact() that calls a function that computes the logarithm of gamma() function.

■ binDF

FUNCTION

y = binDF(n, a, k)

PURPOSE

Compute the density function of the binomial distribution with parameters n, a at k:

$$f_{n,a}(k) = \frac{n!a^k(1-a)^{n-k}}{k!(n-k)!}$$

INPUT

n (Integer Scalar): the first parameter for the density function of the binomial distribution

a (Real Scalar): the second parameter for the density function of the binomial distribution

k (Integer Scalar): the point where the density function of the binomial distribution is evaluated

OUTPUT

y (Real Scalar): the evaluated density function at k

EXAMPLES

```
//  Example for binDF(n, a, k), the binomial distribution
//  density function.

//  The parameters are n = 9 and a = 0.4, the evaluation
//  point is k = 8:

n = 9;
a = 0.4;
k = 8;
y = binDF(n,a,k);

//  Result :
//      y:       0.003538944
```

ALGORITHM AND COMMENTS

The density function of the binomial distribution with parameters n and a at k is defined for n = 0, 1, ... ; 0 < a < 1, and k = 0, ... , n.

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p.929

■ binDist

FUNCTION

y = binDist(k,n,p)

PURPOSE

Compute the binomial distribution function, i.e., the cumulative sum of the terms 0 through k of the binomial probability density:

$$\text{binDist}(k, n, p) = \sum_{m=0}^{k} \text{bin}(n, m)\, p^{m} (1-p)^{n-m}$$

INPUT

k (Integer Scalar): the number of terms minus one in the cumulative sum

n (Integer Scalar): the first term in the binomial function coefficient in each term of the sum

p (Real Scalar): the binomial probability value

OUTPUT

y (Real Scalar): the value of the binomial distribution function

EXAMPLES

```
//  Example for binDist(k, n, p), the binomial distribution
//  function.
//
//  The number of terms minus one is k = 8, the first
//  coefficient term is n = 9, and the probability value
//  is p = 0.4:

k = 8;
n = 9;
p = 0.4;
y = binDist(k,n,p);

//  Result :
                y:     0.999737856
```

ALGORITHM AND COMMENTS

Domain: n = 0, 1, …; $0 \le k \le n$; $0 < p < 1$; Range: $0 \le y \le 1$.

The incomplete beta integral is used via the identity:

$$\sum_{m=0}^{k} bin(n, m)\, p^{m} (1-p)^{n-m} = iBeta(1-p, n-k, k+1)$$

for the general case.

■ cauchyDF

FUNCTION

y = cauchyDF(a, b, x)

PURPOSE

Compute the density function of the Cauchy distribution with parameters a, b at x:

$$cauchyDF(a, b, x) = f_{a,b}(x) \frac{1}{\pi b\left[1 + \left(\frac{x-a}{b}\right)^2\right]}$$

INPUT

a (Real Scalar): the first parameter for the density function of the Cauchy distribution

b (Real Scalar): the second parameter for the density function of the Cauchy distribution

x (Real Scalar): the point where the density function of the Cauchy distribution is evaluated

OUTPUT

y (Real Scalar): the evaluated density function at x

EXAMPLES

```
//  Example for cauchyDF(a, b, x), the Cauchy distribution
//  density function.

//  The parameters a = 0 and b = 1; the evaluation point
//  is x = 1:

a = 0;
b = 1;
x = 1;
y = cauchyDF(a,b,x);
//  Result :
//     y:      0.159154943091895
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$, $-\infty < a < \infty$, $0 < b < \infty$

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p.930

■ chiSq

FUNCTION

y = chiSq(n, χ^2)

PURPOSE

Compute the chi-square probability distribution function for n degrees of freedom:

$$\text{chiSq}(n, \chi^2) = \frac{1}{2^{n/2}\Gamma(n/2)} \int_0^{\chi^2} z^{n/2-1} e^{-z/2} dz$$

INPUT

n (Real Scalar): the number of degrees of freedom for the chi-square probability function

χ^2 (Real Scalar): the chi-square argument of chiSq(n, χ^2)

OUTPUT

y (Real Scalar): the value of the chiSq(n, χ^2) function

EXAMPLES

```
//  Example for chiSq(n, x), the chi-square distribution
//  function.

//  The number of degrees of freedom n = 2 and the chi-square
//  argument x = 5.1:

n = 2;
x = 5.1;
y = chiSq(n,x);

//  Result :
//      y:       0.921918333998847
```

ALGORITHM AND COMMENTS

Domain: $0 \le \chi^2 \le \infty$, $1 \le n$.

The chi-square probability distribution is computed using the incomplete gamma integral function via the relation:

$$\text{chiSq}(n, \chi^2) = \text{iGamma}(n/2, \chi^2/2)$$

■ comChiSq

FUNCTION

y = comChiSq(n, χ^2)

PURPOSE

Compute the complement of the function chiSq(n, χ^2):

$$\text{comChiSq}(n, \chi^2) = 1 - \text{chiSq}(n, \chi^2)$$

INPUT

n (Real Scalar): the number of degrees of freedom for the chi-square probability function corresponding to the complement

χ^2 (Real Scalar): the chi-square argument of chiSq(n, χ^2) corresponding to the complement

OUTPUT

y (Real Scalar): the value of the comChiSq(n, χ^2) function

EXAMPLES

```
//  Example for comChiSq(n, x), the complement of the
//  chi-square distribution function.

//  The number of degrees of freedom n = 2 and the chi-square
//  argument x = 5.1:

x = 5.1;
n = 2;
y = comChiSq(n,x);

//  Result :
//      y:      0.85793594736096
```

ALGORITHM AND COMMENTS

Domain: $0 \le \chi^2 < \infty$, $1 \le n$.

The complement of the chi-square probability distribution is computed using the complement of the incomplete gamma integral function via the relation:

$$\text{ComChiSq}(n, \chi^2) = \text{comIGamma}(n/2, \chi^2/2)$$

where comIGamma(n/2, χ^2/2) is the complement of the iGamma function.

■ correlate

FUNCTION

y = correlate(u, v)

PURPOSE

Compute the correlation coefficient of two sets of n data points

INPUT

u (Real Vector): the n-dimensional vector containing the n data values $u_1, \ldots, u_n$ for the first set

v (Real Vector): the n-dimensional vector containing the n data values $v_1, \ldots, v_n$ for the second set

OUTPUT

y (Real Scalar): the computed correlation coefficient for u_i and v_i,
where i = 1, ... , n

EXAMPLES

```
//  Example for correlate(u, v), the correlation coefficient
//  of the two sets of data contained in u and v

//  The first data set is:
u = {8; 3; 5; 12; 10};
//  The second data set is:
v = {5; -2; 7; 0; 3};
y = correlate(u, v);

//  Result :
//     y:      0.0037593984962406
```

ALGORITHM AND COMMENTS

The correlation coefficient for the sets u = $\{u_1, \ldots, u_n\}$ and v = $\{v_1, \ldots, v_n\}$ is defined by:

$$\text{correlate}(u, v) = \frac{\text{cov}(u, v)}{s_u s_v}$$

where cov(u,v) is the covariance of u and v and s_u, s_v are the standard deviations of u and v respectively.

REFERENCE

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 298

■ COV

FUNCTION

y = cov(u, v)

PURPOSE

Compute the covariance between two sets of n data points

INPUT

u (Real Vector): the n-dimensional vector containing the n data values $u_1, \ldots, u_n$ for the first set

v (Real Vector): the n-dimensional vector containing the n data values $v_1, \ldots, v_n$ for the second set

OUTPUT

y (Real Scalar): the computed covariance for the sets u_i and v_i, where i = 1, ... , n

EXAMPLES

```
//  Example for cov(u, v), the covariance between the two data
//  sets contained in the vectors u and v

//  The first data set is:
u = {8; 3; 5; 12; 10};
//  The second data set is:
v = {5; -2; 7; 0; 3};
y = cov(u, v);

//  Result :
//    y:    0.04
```

ALGORITHM AND COMMENTS

The covariance of the sets u = {u1, ... , un} and v = {v1, ... , vn} is defined by:

$$\mathrm{cov}(u, v) = \sum_{i=1}^{n} \frac{(u_i - u_{avg})(v_i - v_{avg})}{n}$$

where u_{avg} and v_{avg} are the averages of $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$, respectively.

REFERENCE

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 298

■ covMatrix

FUNCTION

C = covMatrix(A)

PURPOSE

Compute the covariance matrix for n data sets, each containing m values

INPUT

A (Real Matrix): the m by n matrix storing the values of the n data sets where each column of the matrix contains the m values of a data set

OUTPUT

C (Real Matrix): the computed n by n covariance matrix

EXAMPLES

```
//  Example for covMatrix(A), the covariance matrix for
//  n data sets, each of length m

//  The data matrix is:
A = {1, 2;
     3, 4};
C = covMatrix(A);

//  Result :
//  C:  2 rows, 2 columns
//             1          1
//             1          1
```

ALGORITHM AND COMMENTS

The n by n symmetric covariance matrix has elements

$$C_{ij} = \frac{\sum_{k=1}^{m} (X_{ki} - \bar{X}_{i,\,avg})\,(X_{kj} - \bar{X}_{j,\,avg})}{m} \qquad \text{for } 1 \le i, j \le n$$

where

$$\bar{X}_{i,\,avg} = \frac{\sum_{k=1}^{m} X_{ki}}{m} \quad \text{and} \quad \bar{X}_{j,\,avg} = \frac{\sum_{k=1}^{m} X_{kj}}{m}$$

REFERENCE

Brandt, S., *Statistical and Computational Methods in Data Analysis*, North-Holland, 1976, p. 372

■ cumeDF

FUNCTION

y = cumeDF(a,b,x)

PURPOSE

Compute the cumulative distribution function for the exponential distribution with parameters a and b at x:

$$\text{cumeDF}(a, b, x) = \int_{a}^{x} \frac{1}{b} e^{-\left(\frac{t-a}{b}\right)} dt$$

$$= 1 - e^{-\left(\frac{x-a}{b}\right)} \qquad \text{for } x \geq a$$

INPUT

a (Real Scalar): the first parameter of the probability density function for the exponential distribution

b (Real Scalar): the second parameter of the probability density function for the exponential distribution

x (Real Scalar): the point where the cumulative distribution function is evaluated

OUTPUT

y (Real Scalar): the computed cumulative distribution function at x

EXAMPLES

```
//  Example for cumeDF(a, b, x), the exponential cumulative
//  distribution function

//  The first and second parameters are a = -4 and b = 1; the
//  evaluation point is x = 0.0:
a = -4;
b = 1;
x = 0;
```

```
y = cumeDF(a,b,x);

//  Result :
    y:       0.981684361111266
```

ALGORITHM AND COMMENTS

Domain: $a \le x < \infty$, $-\infty < a < \infty$, $0 < b < \infty$

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, pp. 927, 930

■ eDF

FUNCTION

y = eDF(a,b,x)

PURPOSE

Compute the probability density function for the exponential distribution with parameters a and b at x:

$$eDF(a, b, x) = \frac{1}{b} e^{-\left(\frac{x-a}{b}\right)} \qquad \text{for } x \ge a$$

INPUT

a (Real Scalar): the first parameter of the probability density function for the exponential distribution

b (Real Scalar): the second parameter of the probability density function for the exponential distribution

x (Real Scalar): the point where the probability density function is evaluated

OUTPUT

y (RealScalar): the computed density function at x

EXAMPLES

```
//  Example for eDF(a, b, x), the exponential distribution
//  density function

//  The first and second parameters are a = -4 and b = 1; the
//  evaluation point is x = 0.0:

a = -4;
b = 1;
x = 0;
y = eDF(a,b,x);
```

```
//  Result :
//     y:      0.0183156388887342
```

ALGORITHM AND COMMENTS

Domain: $a \leq x < \infty$, $-\infty < a < \infty$, $0 < b < \infty$

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p. 930

■ errDF

FUNCTION

y = errDF(a, x)

PURPOSE

Compute the density function of the error distribution function with parameter a at x:

$$\text{errDF}(a, x) = \frac{a}{\sqrt{\pi}} e^{-a^2x^2}$$

INPUT

a (Real Scalar): the parameter for the density function of the error distribution function

x (Real Scalar): the point where the density function of the error distribution function is evaluated

OUTPUT

y (Real Scalar): the evaluated density function at x

EXAMPLES

```
//  Example for errDF(a, x), the error distribution
//  density function

//  The parameter value is a = 1, the evaluation
//  point is x = -2:
a = 1;
x = -2;
y = errDF(a,x);

//  Result :
//    y:     0.010333492677046
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$, $0 < a < \infty$

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p.930

■ fact

FUNCTION

y = fact(n)

PURPOSE

Compute the factorial of n, n! = n (n-1) … (2) (1)

INPUT

n (Integer Scalar): the integer argument for the factorial function

OUTPUT

y (Integer or Real Scalar): the value of the factorial of n, n!

EXAMPLES

```
//  Example for fact(n), the factorial function n!

//  The argument for the factorial function is n = 6:

n = 6;
y = fact(n);

//  Result :
//     y:      720
```

ALGORITHM AND COMMENTS

When the value of y exceeds the largest integer on the system, a real value is returned that approximates the value of n! to a high degree of precision.

■ fDist

FUNCTION

y = fDist(f, n_1, n_2)

PURPOSE

Compute the F distribution function (Snedecor's density function) for the probability density function F = $(X_1/n_1)/(X_2/n_2)$, where X_1 and X_2 are random variables with chi-square distributions with n_1 and n_2 (degrees of freedom) parameters

INPUT

f (Real Scalar): the variance ratio value F

n_1 (Integer Scalar): the first degree of freedom parameter

n_2 (Integer Scalar): the second degree of freedom parameter

OUTPUT

y (Real Scalar): the value of the F-distribution probability distribution function of f with parameters n_1 and n_2

EXAMPLES

```
//  Example for fDist(f, n1, n2), the F distribution function
//
//  The variance ratio f = 10.2, and the degree of freedom
//  parameters n1 = 1 and n2 = 3:

f = 10.2;
n1 = 1;
n2 = 3;
y = fDist(f,n1,n2);

//  Result :
//    y:       0.0495686721739861
```

ALGORITHM AND COMMENTS

Domain: $0 \le f < \infty$, $1 \le n_1$, $1 \le n_2$. The algorithm for this function uses the relation to the incomplete beta integral function given by:

$$fDist(f, n_1, n_2) = iBeta(n_1 f/(n_2 + n_1 f), n_1/2, n_2/2).$$

■ gammaDF

FUNCTION

y = gammaDF(a, x)

PURPOSE

Compute the density function of the gamma distribution with parameter a at x:

$$\text{gammaDF}(a, x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}$$

INPUT

a (Real Scalar): the parameter for the density function of the gamma distribution

x (Real Scalar): the point where the density function ofthe gamma distribution is evaluated

OUTPUT

y (Real Scalar): the evaluated density function at x

EXAMPLES

```
//  Example for gammaDF(a, x), the gamma distribution
//  density function

//  The distribution parameter is a = 1; the evaluation
//  point is x = 10.0:

a = 1;
x = 10;
y = gammaDF(a,x);

//  Result :
//     y:       4.53999297624849e-05
```

ALGORITHM AND COMMENTS

Domain: $0 \le x < \infty$, $0 < a < \infty$

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p.930

■ gammaDist

FUNCTION

y = gammaDist(a,b,x)

PURPOSE

Compute the value of the gamma density distribution function at x:

$$\text{gammaDist}(a, b, x) = \frac{a^b}{\Gamma(b)} \int_0^x z^{b-1} e^{-az} dz$$

INPUT

a (Real Scalar): the scale parameter of the distribution function

b (Real Scalar): the parameter corresponding to the (related function) incomplete gamma integral

x (Real Scalar): the argument of the gammaDist() function

OUTPUT

y (Real Scalar): the value of the gamma distribution function

EXAMPLE

```
//  Example for gammaDist(a, b, x), the gamma distribution
//  function

//  The distribution parameters are a = 1 and b = 2; the
//  argument of the distribution is evaluated at x = 10.0:

a = 1;
b = 2;
x = 10;
y = gammaDist(a,b,x);

//  Result :
//     y:          0.999500600772613
```

ALGORITHM AND COMMENTS

Domain: a, b > 0; $0 \le x < \infty$; Range: $0 \le y \le 1$

This function is computed via its relation to the incomplete gamma integral:

$$\text{igamma}(a, x) = \frac{1}{\Gamma(a)} \int_0^x z^{a-1} e^{-z} dz$$

which is computed either via a continued fraction expansion or the power series expansion for various ranges of its arguments.

■ gaussDF

FUNCTION

y = gaussDF(x)

PURPOSE

Compute the (normalized) density function of the standard normal (or Gaussian) distribution function at x:

$$\text{gaussDF}(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

INPUT

x (Real Scalar): the point where the density function of the standard normal distribution function is evaluated

OUTPUT

y (Real Scalar): the value of the density function gaussDF(x) at x

EXAMPLES

```
//  Example for gaussDF(x), the standard normal (Gaussian)
//  distribution density function

//  The evaluation point is x = 0.0:
x = 0;
y = gaussDF(x);

//  Result :
//      y:        0.398942280401433
```

ALGORITHM AND COMMENTS

Domain: $-\infty < x < \infty$, $0 < a < \infty$

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p.930

■ gaussDist

FUNCTION

y = gaussDist(x)

PURPOSE

Compute the value of the normal (or Gaussian) distribution function at x:

$$\text{gaussDist}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{z^2/2} dz$$

INPUT

x (Real Scalar): the argument at which the normal distribution is to be computed

OUTPUT

y (Real Scalar): the value of the normal probability distribution function at x

EXAMPLE

```
//  Example for gaussDist(x), the normal (Gaussian)
//  distribution function

//  The evaluation point is x = 1:

x = 1;
y = gaussDist(x);

//  Result :
//     y:          0.841344746068543
```

ALGORITHM AND COMMENTS

Domain: $0 < x < \infty$; Range: $0 \leq y \leq 1$

This function is related directly to the error functions erf(x) and erfc(x); in fact, gaussDist (x) = (1 + erf(x/sqrt(2)))/2 = (1 - erfc(x/sqrt(2)))/2. For small x, we use the erf(x) function; for larger values of x, the erfc(x) function is used.

■ geoDF

FUNCTION

y = geoDF(a, k)

PURPOSE

Compute the density function of the geometric distribution with parameter a at k:

$$\text{geoDF}(a, k) = a(1 - a)^k$$

for $0 < a < 1$, and $k = 0, 1, \ldots$

INPUT

a (Real Scalar): the parameter for the density function of the geometric distribution

k (Integer Scalar): the point where the density function of the geometric distribution is evaluated

OUTPUT

y (Real Scalar): the evaluated density function at k

EXAMPLES

```
//  Example for geoDF(a, k), the geometric distribution
```

```
//  density function

//  The distribution parameter is a = 0.5; the evaluation
//  point is k = 4.0:
a = 0.5;
k = 4;
y = geoDF(a,k);
//  Result :
         y:      0.03125
```

ALGORITHM AND COMMENTS

Domain: k = 0, 1, … , 0 < a < 1

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p.929

■ kurt

FUNCTION

y = kurt(u)

PURPOSE

Compute the moment coefficient of kurtosis for a set of n data points

INPUT

u (Real Vector): the n-dimensional vector containing the n data values $u_1, \ldots, u_n$

OUTPUT

y (Real Scalar): the moment coefficient of kurtosis of $u_1, \ldots, u_n$

EXAMPLES

```
//  Example for kurt(u), the moment coefficient of
//  kurtosis for the data set contained in u

//  The data set is:
u = {8; 3; 5; 12; 10};
y = kurt(u);

//  Result :
//     y:      1.59254338854655
```

ALGORITHM AND COMMENTS

The moment coefficient of kurtosis of $u_1, \ldots, u_n$ is defined by

$$\text{kurt}(u) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{u_i - u_{avg}}{s_u}\right)^4$$

where u_{avg} and s_u are the average and standard deviation of u_1, …, u_n, respectively.

COMMENT:

For the reason that KT=3 for the normal distribution (see the reference), the moment coefficient of kurtosis is sometimes defined by the value KT - 3.

REFERENCE

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 112

■ median

FUNCTION

y = median(u)

PURPOSE

Compute the median (or middle value) of a set of n data points

INPUT

u (Real Vector): the n-dimensional vector containing the n data values u_1, ...,u_n

OUTPUT

y (Real Scalar): the median of u_1, ... , u_n

EXAMPLES

```
//  Example for median(u), the median of the data
//  values contained in u

//  The data set is:
u = {8; 3; 5; 12; 10};
y = median(u);

//  Result :
//      y:      8
```

ALGORITHM AND COMMENTS

The median of u_1, ... , u_n is defined as follows:

$$\text{median} = u_{(n+1)/2} \qquad \text{if n is an odd integer, and}$$

$$\text{median} = \frac{(u_{n/2} + u_{n/2+1})}{2} \qquad \text{if n is an even integer}$$

The computation of the median is done by combining a (modified) Quicksort algorithm (see the reference for details) together with a Bubble sort (for data number no larger than 9).

REFERENCE

Knuth, D.E., *Sorting and Searching: The Art of Computer Programming*, Vol. 3, Addison-Wesley, 1973, p. 114

■ mult

FUNCTION

y = mult(v)

PURPOSE

Compute the multi-nomial coefficient (specified by an integer vector):

$$\text{mult}(v) = \frac{(v_1 + \ldots + v_n)!}{v_1! \ldots v_n!}$$

INPUT

v (Integer Vector): the n-dimensional integer vector, $(v_1, \ldots, v_n)^t$, with $v_i \geq 0$ for $1 \leq i \leq n$

OUTPUT

m (Integer Scalar): the computed multi-nomial coefficient specified by the input vector v, i. e.,

EXAMPLE

```
//  Example for mult(U), the multi-nomial coefficient
//  of the data values contained in U

//  The data set is:
U = {1;2;3;4};
y = mult(U);

//  Result :
//   y:   12600
```

ALGORITHM AND COMMENTS

All components of v must be nonnegative: $v_i \geq 0$.

■ negBinDist

FUNCTION

y = negBinDist(k,n,p)

PURPOSE

Compute the negative binomial distribution function, i.e., the cumulative sum of the terms 0 through k of the negative binomial probability density:

$$\text{negBinDist}(k, n, p) = \sum_{m=0}^{k} \text{bin}(n + m - 1, m)\, p^n (1 - p)^m$$

INPUT

k (Integer Scalar): the number of terms minus one in the cumulative sum

n (Integer Scalar): the term indicating which success (i.e., the nth success) occurs at the (n + m)th trial

p (Real Scalar): the binomial probability value

OUTPUT

y (Real Scalar): the value of the negative binomial distribution function

EXAMPLES

```
//  Example for negBinDist(k, n, p), the negative binomial
//  distribution function

//  The number of terms minus one is k = 8, the success
//  parameter is n = 9, and the binomial probability
//  is p = 0.4:

k = 8;
n = 9;
p = 0.4;
y = negBinDist(k,n,p);

//  Result :
//   y:          0.19893648967598
```

ALGORITHM AND COMMENTS

Domain: $0 \le p \le 1; 0 < k; 1 < n$

The incomplete beta integral is used via the identity:

$$\sum_{m=0}^{k} bin(n+m-1, m)\, p^n (1-p)^m = iBeta(p, n, k+1)$$

for the general case.

This distribution function computes the probability that the nth success is preceded by k or less failures. Each term of the sum provides the probability that the nth success occurs on the (n + m)th trial in a sequence of Bernoulli trials.

■ poi

FUNCTION

y = poi(k, λ)

PURPOSE

Compute the Poisson distribution function with parameters k and λ:

$$poi(k, \lambda) = \sum_{m=0}^{k} \frac{e^{-\lambda}\lambda^m}{m!}$$

INPUT

k (Integer Scalar): the number of terms minus one in the distribution function

λ (Real Scalar): the Poisson parameter value

OUTPUT

y (Real Scalar): the value of the Poisson distribution function

EXAMPLES

```
//  Example for poi(k, p), the Poisson distribution function
//  The number of terms minus one is k = 2, and the
//  Poisson parameter is p = 0.9:
k = 2;
p = 0.9;
y = poi(k,p);
//  Result :
//    y:        0.937143065702081
```

ALGORITHM AND COMMENTS

Domain: $k = 0, 1, \ldots, 0 < \lambda < \infty$.

The Poisson distribution is computed using the complement of the incomplete gamma integral function via the relation:

$$poi(k, \lambda) = 1 - iGamma(k + 1, \lambda)$$

■ poiDF

FUNCTION

y = poiDF(a, k)

PURPOSE

Compute the density function of the Poisson distribution with parameter a at k:

$$poiDF(a, k) = \frac{e^{-a}a^{k}}{k!}$$

INPUT

a (Real Scalar): the parameter for the density function of Poisson distribution

k (Integer Scalar): the point where the density function of Poisson distribution is evaluated

OUTPUT

y (Real Scalar) : the evaluated density function at k

EXAMPLES

```
//  Example for poiDF(k,p), the Poisson distribution
//  density function

//  The distribution parameter is k = 2, the evaluation
//  point is p = 0.9:

k = 2;
p = 0.9;
y = poi(k,p);

//  Result :
//     y:      0.164660712194943
```

ALGORITHM AND COMMENTS

Domain: $0 < a < \infty$, and k = 0, 1,...

REFERENCE

Abramowitz, M. and Stegun, I.A., *Handbook of Mathematical Functions*, Dover Publ., 1972, p.929

■ rand

FUNCTION

y = rand(n)

PURPOSE

Return a uniformly distributed random number seeded with an integer

INPUT

n (Integer Scalar): the integer seed value. If n = 0, the next number in a random sequence is returned. If n = 1, the random sequence is reset to the default sequence and the first number in the sequence is returned. For all other n, the random sequence is seeded with n and the first number in the sequence is returned.

OUTPUT

y (Real Scalar): a uniformly distributed random real value in the range from 0 to 1

EXAMPLES

```
//  Example for rand(n), the integer seeded random
//  number function

y1 = rand(0);
y2 = rand(0);
y3 = rand(0);

y4 = rand(1); // Reset to default sequence
y5 = rand(0);
y6 = rand(0);

// Result:
//     y1:    0.000061035389081
//     y2:    0.500856401864439
//     y3:    0.525854131905362
//
//     y4:    0.000061035389081
//     y5:    0.500856401864439
//     y6:    0.525854131905362

y1 = rand(2); // Set seed to 2
```

```
y2 = rand(0);
y3 = rand(0);

y4 = rand(3); // Set seed to 3
y5 = rand(3); // Set seed to 3
y6 = rand(3); // Set seed to 3

// Result:
//     y1:    0.000553135527298
//     y2:    0.540964166633785
//     y3:    0.489183786557987
//
//     y4:    0.000799185596406
//     y5:    0.000799185596406
//     y6:    0.000799185596406
```

SEE ALSO

randM(m, n); sRandM(m, n, seed)

ALGORITHM AND COMMENTS

For seed values other than zero, rand will return the same random number each time that seed is used. The default sequence is the random sequence that occurs when calling rand(0) without first setting the seed.

REFERENCE

Knuth, D.E., *Seminumerical Algorithms*, 2nd. Ed., Vol. 2 of *The Art of Computer Programming*, Addison-Wesley, 1981, sections, 3.2, 3.3

■ RMS

FUNCTION

y = RMS(u)

PURPOSE

Compute the root mean square (or quadratic mean) of a set of n data points:

$$\mathrm{RMS}(u) = \sqrt{\sum_{i=1}^{n} \frac{u_i^2}{n}}$$

INPUT

u (Real Vector): the n-dimensional vector containing the n data values $u_1, \ldots, u_n$

OUTPUT

y (Real Scalar): the computed root mean square of $u_1, \ldots, u_n$

EXAMPLES

```
//  Example for RMS(u), the root mean square of the
//  values contained in u

//  The data set is:
u = {8; 3; 5; 12; 10};
y = RMS(u);

//  Result :
//   y:        8.27042925125413
```

REFERENCE

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 79

■ skew

FUNCTION

y = skew(u)

PURPOSE

Compute the moment coefficient of skewness for a set of n data points:

$$\text{skew}(u) = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{u_i - u_{avg}}{s_u}\right)^3$$

where u_{avg} and s_u are the average and standard deviation of $u_i, \ldots, u_n$, respectively

INPUT

u (Real Vector): the n-dimensional vector containing the n data values $u_1, \ldots, u_n$

OUTPUT

y (Real Scalar): the moment coefficient of skewness of $u_1, \ldots, u_n$

EXAMPLES

```
//  Example for skew(u), the moment coefficient of skewness
```

```
//  for the data contained in u
//  The data set is:
u = {8; 3; 5; 12; 10};
y = skew(u);

//  Result :
//     y:      -0.0912793903894912
```

REFERENCE

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 111

■ stanDev

FUNCTION

y = stanDev(u)

PURPOSE

Compute the standard deviation of a set of n data points

INPUT

u (Real Vector): the n-dimensional vector containing the n data values $u_1, \ldots, u_n$

OUTPUT

y (Real Scalar): the computed standard deviation of $u_1, \ldots, u_n$

EXAMPLES

```
//  Example for stanDev(u), the standard deviation of the
//  data set contained in u

//  The data set is:
u = {8; 3; 5; 12; 10};
y = stanDev(u);

//  Result :
//     y:        3.26190128606002
```

ALGORITHM AND COMMENTS

The standard deviation of $u_1, \ldots, u_n$ is defined by:

$$s_u = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\frac{(u_i - u_{avg})^2}{n}}$$

where u_{avg} is the average of u_i, …, u_n.

REFERENCE

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 88

■ stanForm

FUNCTION

y = stanForm(u)

PURPOSE

Convert a set of n data points to standardized variable representation (i.e., a set of n data points with zero average and unity standard deviation)

INPUT

u (Real Vector): the n-dimensional vector containing the n data values u_1, ... , u_n

OUTPUT

y (Real Vector): the n-dimensional vector containing the standardized set of variables z_1, ... , z_n

EXAMPLES

```
//  Example for stanForm(u), the standardized form of
//  of the data set contained in u
//  The data set is:
u = {8; 3; 5; 12; 10};
y = stanForm(u);

//  Result : y: 5 rows
//        0.122627867896993
//       -1.41022048081542
//       -0.797081141330456
//        1.34890654686692
//        0.735767207381959
```

ALGORITHM AND COMMENTS

The standardized variable z_i corresponding to u_i, i =1, ... , n, is defined by

$$z_i = \frac{u_i - u_{avg}}{s_u}$$

where u_{avg} and s_u are the average and standard deviation of u_i, …, u_n, respectively.

REFERENCE

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 91

■ student

FUNCTION

y = student(t, n)

PURPOSE

Compute the Student t distribution function:

$$\text{student}(t, n) = \frac{\Gamma\left(\frac{n+1}{2}\right)}{(n\pi)^{1/2}\Gamma(n/2)} \int_{-\infty}^{t} \left(1 + \frac{x^2}{n}\right)^{-\frac{n+1}{2}} dx$$

INPUT

t (Real Scalar): the distribution parameter for student(t,n)

n (Integer Scalar): the number of degrees of freedom of the student t density function

OUTPUT

y (Real Scalar): the value of the Student t distribution function

EXAMPLES

```
//  Example for student(t, n), the Student t
//  distribution function
//  The student t parameter is t = 1.2, and the number
//  of degrees of freedom is n = 4:

t = 1.2;
n = 4;
y = student(t,n);

//  Result :
            y:     0.851824303343823
```

ALGORITHM AND COMMENTS

Domain: $-\infty < t < \infty$, n = 1, 2, … If t = 0, then the value of student(t,n) is 1/2.

The algorithm for this function consists of performing an integration by parts for the domain $-1 < t < 1$ and by using the relation to the incomplete beta integral function given by:

$$\int_{-\infty}^{|t|} \left(1 + \frac{x^2}{n}\right)^{-\frac{n+1}{2}} dx = \frac{1}{2}\text{ibeta}\left(\frac{n}{n+t^2}, \frac{n}{2}, \frac{1}{2}\right)$$

■ var

FUNCTION

y = var(u)

PURPOSE

Compute the variance of a set of n data points

INPUT

u (Real Vector): the n-dimensional vector containing the n data values $u_1, \ldots, u_n$

OUTPUT

y (Real Scalar): the computed variance of $u_1, \ldots, u_n$

EXAMPLES

```
//  Example for var(u), the variance of the data
//  contained in u

//  The data set is:
u = {8; 3; 5; 12; 10};
y = var(u);

//  Result :
//      y:      10.64
```

ALGORITHM AND COMMENTS

The variance of $u_1, \ldots, u_n$. is defined by:

$$\text{var}(u) = \frac{1}{n}\sum_{i=1}^{n} (u_i - u_{avg})^2$$

where u_{avg} is the average of $u_1, \ldots, u_n$.

Note that n should be replaced by n-1 in this formula when the average is based on a set of sample data, i.e., we assume the mean u_{avg} is known. A simple HiQ-Function can be written for this other case.

REFERENCE

Spiegel, M.R., *Schaum's Outline Series of Theory and Problems of Statistics*, McGraw-Hill, 1990, p. 89

■ weibull

FUNCTION

y = weibull(a,b,x)

PURPOSE

Compute the density function of the Weibull distribution with parameters a, b at x:

$$\text{weibull}(a, b, x) = abx^{a-1}e^{-bx^a}$$

for a,b > 0 and x > 0

INPUT

a (Real Scalar): the first parameter for the density function of Weibull distribution

b (Real Scalar): the second parameter for the density function of Weibull distribution

x (Real Scalar): the point where the density function of Weibull distribution is evaluated

OUTPUT

y (Real Scalar) : the evaluated density function at x

EXAMPLES

```
//  Example for weibull(a, b, x), the Weibell distribution
//  density function

//  The distribution parameters are a = 0.2 and b = 1, the
//  evaluation point is x = 0.5:
a = 0.2;
b = 1;
x = 0.5;
y = weibull(a,b,x);
//  Result :
//     y:      0.145807104733769
```

ALGORITHM AND COMMENTS

Domain: a > 0; b > 0; x > 0; Range: 0 < y < ∞

REFERENCE

Bickel, P.J. and Doksum, K.A., *Mathematical Statistics : Basic Ideals and Selected Topics*, Holden-Day, Inc., San Francisco, 1977, P. 82

CHAPTER 20
DATA FITTING FUNCTIONS

■ bSplineBasis

FUNCTION

[coefMatrix, intervalMatrix, basisVector] = bSplineBasis(k, u)

PURPOSE

Generate the (normalized) B-spline basis of order k (= 2, 3 or 4) from a given set of n montonically increasing knots

INPUT

k (Integer Scalar): the order (=2, 3 or 4) of the desired B-spline basis

u (Real Vector): the n-dimensional vector containing the n monotonically increasing knots, i.e., u[i] < u[i+1] for i=1, ... , n-1

OUTPUT

coefMatrix (Real Matrix): the k by k(n-1) matrix containing the coefficients of the polynomials of order k (or degree k-1) for the nonzero supports of the B-spline basis

intervalMatrix (Real Matrix): the 2 by k(n-1) matrix containing the intervals of the kth order polynomial for the nonzero supports of the B-spline basis corresponding to the columns of the output matrix coefMatrix

basisVector (Integer Vector): the integer vector of dimension k(n-1) containing the indices of the B-spline basis corresponding to the columns of coefMatrix

EXAMPLES

```
//  An example for: bSplineBasis(k, u), a normalized B-spline
//  basis function

//  The order of the basis is k = 2:
k = 2;
//  The knot vector is:
u = { 1, 2, 3, 4, 5 };

[coefMatrix, intervalMatrix, basisVector] = bSplineBasis(k, u);

//  Results:
//  coefMatrix:  2 rows, 8 columns
//      2    -1    3    -2     4     -3     5      -4
//     -1     1   -1     1    -1      1    -1       1
//  basisVector:  8 rows
```

```
//  1
//  2
//  2
//  3
//  3
//  4
//  4
//  5
//
//  intervalMatrix:  2 rows, 8 columns
//       1     1    2     2      3     3     4      4
//       2     2    3     3      4     4     5      5
```

ALGORITHM AND COMMENTS

Algorithm Description:

For a given set of n knots, u[1] < ... < u[n], we generate a new sequence of n+2(k-1) knots such that t_i=u[i+1], for i=0, ... , n-1 and t_{-j}=u[1] - jh_1, t_{n+j-1} = u[n] + jh_{n-1} for j=1, ... , k-1 where h_1=u[2] - u[1] and h_{n-1} = u[n] - u[n-1]. The B-spline basis for the original n knots, u[1], ... , u[n] contains n+k-2 splines (of order k) and can be constructed in terms of the new knot sequence, t_{1-k}, ... , t_{n+k-2}, as follows:

The ith (normalized) B-spline of any order k (≥2) for a set of monotonically increasing knots t_{1-k} < ... < t_{n+k} is defined by the finite differences

$$B_i(x) = (t_i - t_{i-k})\,[t_{i-k}, ..., t_i]\,(t-x)_+^{k-1} \qquad \text{for all } x$$

where

$$(t-x)_+^{k-1} = (t-x)^{k-1} \qquad \text{if } t > x, \text{ and}$$

$$= 0 \qquad \text{if } t \le x$$

is called the truncated power function of order k. Note that B_i has only finite nonzero support, that is, $B_i(x)$ is nonzero only for x in the interval [t_{i-k} , t_i]. In our program, the following explicit piecewise polynomial expressions of the B-spline basis of orders 2, 3 or 4 are used.

1) B-spline basis for order 2 (or linear B-spline).

$$B = \begin{cases} A_i + C_i + D_i & \text{if } t_{i-3} < x < t_{i-2} \\ A_i + C_i & \text{if } t_{i-2} \le x < t_{i-1} \\ A_i & \text{if } t_{i-1} \le x < t_i \\ 0 & \text{otherwise} \end{cases}$$

where

$$A_i = \frac{t_i - x}{t_i - t_{i-1}}$$

$$C_i = -\left(\frac{1}{t_i - t_{i-1}} + \frac{1}{t_{i-1} - t_{i-2}}\right)(t_{i-1} - x)$$

for i=1, ... , n.

2) B-spline basis for order 3 (or quadratic B-spline).

$$B = \begin{cases} A_i + C_i + D_i & \text{if } t_{i-3} < x < t_{i-2} \\ A_i + C_i & \text{if } t_{i-2} < x < t_{i-1} \\ A_i & \text{if } t_{i-1} < x < t_i \\ 0 & \text{otherwise} \end{cases}$$

where

$$A_i = \frac{(t_i - x)^2}{(t_i - t_{i-2})(t_i - t_{i-1})}$$

$$C_i = \frac{-(t_i - t_{i-3})}{(t_i - t_{i-1})(t_{i-1} - t_{i-2})(t_{i-1} - t_{i-3})}(t_{i-1} - x)^2$$

$$D_i = \frac{-(t_1 - t_{i-3})}{(t_i - t_{i-2})(t_{i-1} - t_{i-2})(t_{i-2} - t_{i-3})}(t_{i-2} - x)^2$$

for i = 1, ... , n+1.

3) B-spline basis for order 4 (or cubic B-spline).

$$B = \begin{cases} A_i + C_i + D_i + E_i & \text{if } t_{i-4} < x < t_{i-3} \\ A_i + C_i + D_i & \text{if } t_{i-3} \le x < t_{i-2} \\ A_i + C_i & \text{if } t_{i-2} \le x < t_{i-1} \\ A_i & \text{if } t_{i-1} \le x < t_i \\ 0 & \text{otherwise} \end{cases}$$

where

$$A_i = \frac{(t_i - x)^3}{(t_i - t_{i-3})\,(t_i - t_{i-2})\,(t_i - t_{i-1})}$$

$$C_i = \frac{-(t_i - t_{i-4})}{(t_i - t_{i-1})\,(t_{i-1} - t_{i-4})\,(t_{i-1} - t_{i-3})\,(t_{i-1} - t_{i-2})}(t_{i-1} - x)^3$$

$$D_i = \frac{(t_i - t_{i-4})}{(t_i - t_{i-2})\,(t_{i-1} - t_{i-2})\,(t_{i-2} - t_{i-4})\,(t_{i-2} - t_{i-3})}(t_{i-2} - x)^3$$

$$E_i = \frac{-(t_i - t_{i-4})}{(t_i - t_{i-3})\,(t_{i-1} - t_{i-3})\,(t_{i-2} - t_{i-4})\,(t_{i-3} - t_{i-4})}(t_{i-3} - x)^3$$

for i = 1, ... , n+2.

Comments :

BSplineBasis returns two matrices, coefMatrix and intervalMatrix, and an integer vector basisVector. They specify only the nonzero supports of the established B-spline basis. In other words, any subinterval composed of two adjacent knots from the original knot sequence, u[1],...,u[n], having zero function values over it will not be shown in coefMatrix, intervalMatrix and basisVector.

Here is a simple example. Suppose that the input arguments are k=2, u[1] = -1, u[2] = 0 and u[3] = 1; then by calling the function bSplineBasis(k,u), we will obtain:

$$\text{CoefMatrix} = \begin{bmatrix} 0 & 1.0 & 1.0 & 0 \\ -1.0 & 1.0 & -1.0 & 1.0 \end{bmatrix}$$

$$\text{CoefInterval} = \begin{bmatrix} -1.0 & -1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 1.0 \end{bmatrix}$$

and

$$\text{basisVector} = \begin{bmatrix} 1 & 2 & 2 & 3 \end{bmatrix}^t$$

The values provided in coefMatrix, intervalMatrix, and basisVector represent the following B-spline basis:

$$B_1(x) = \begin{cases} -1.0x + 0.0 & \text{if } -1 \le x < 0 \\ 0 & \text{if } 0 \le x \le 1 \end{cases}$$

$$B_2(x) = \begin{cases} 1.0x + 1.0 & \text{if } -1 \le x < 0 \\ -1.0x + 1.0 & \text{if } 0 \le x \le 1 \end{cases}$$

and

$$B_3(x) = \begin{cases} 0 & \text{if } -1 \le x < 0 \\ 1.0x + 0 & \text{if } 0 \le x \le 1 \end{cases}$$

REFERENCE

de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978, pp. 108-109

■ bSplineInterp

FUNCTION

[coefMatrix, intervalMatrix] = bSplineInterp(k, u, xVector, yVector)

PURPOSE

Generate the B-spline interpolation of order k (= 2, 3 or 4) from a set of n montonically increasing knots and n+k-2 appropriately chosen monotonically increasing interpolation points and their corresponding function values

INPUT

k (Integer Scalar): the order (=2, 3 or 4) of the desired interpolation B-spline

u (Real Vector): the n-dimensional vector containing the n monotonically increasing knots, i.e., u[i] < u[i+1] for i=1, ... , n-1

xVector: the (n+k-2)-dimensional interpolating point vector containing the n+k-2 interpolation points

yVector: (n+k-2)-dimensional vectors containing the n+k-2 y values corresponding to those contained in xVector

OUTPUT

coefMatrix (Real Matrix): the k by (n-1) matrix containing the coefficients of the polynomials of order k (or degree k-1) for the interpolation B-spline in the subintervals composed by two adjacent knots

intervalMatrix (Real Matrix): the 2 by (n-1) matrix containing the intervals, composed of two adjacent knots, corresponding to the columns of the output matrix coefMatrix for the interpolation B-spline

EXAMPLES

```
//  An example for:  bSplineInterp(k, u, xVector, yVector),
```

```
//  the B-spline interpolation formed from knot and
//  interpolated data
//  The order of the spline is k = 3:
k = 3;
//  The knot vector is:
u = {1, 2, 3, 4, 5.};
//  The vectors containing the interpolation data are:
xVector = { 1.5, 2.5, 3.5, 4.5, 4.75, 5};
yVector = {2,5,7,4,1,-1};

[coefMatrix, intervalMatrix] = bSplineInterp (k, u, xVector, yVector);

//  Result :
//  coefMatrix:  3 rows, 4 columns
//      -7339          3045         -987          229
//       8218         -2166          522          -86
//      -2216           380          -68            8
//  intervalMatrix:  2 rows, 4 columns
//          1             2            3            4
//          2             3            4            5
```

ALGORITHM AND COMMENTS

The interpolation B-spline defined on n knots with n+k-2 interpolation points $x_1, \ldots, x_{n+k-2}$, and their respective values $y_1, \ldots, y_{n+k-2}$ is the spline given by

$$C(x) = \sum_{j}^{n+k-2} \alpha_j B_j(X)$$

with properly chosen α_j which satisfy

$$C(x_i) = y_i$$

for i=1, ... , n+k-2 where $B_1(x), \ldots, B_{n+k-2}(x)$ are the B-spline basis of order k.

It is easy to see that the problem of constructing interpolation B-splines is equivalent to the solution of systems of linear equations

$$A\alpha = Y$$

where

$$A_{ij} = B_j(x_i) \qquad \text{for } 1 \le i,\ j \le n+k-2$$

$$\alpha = (\alpha_1, \ldots \alpha_{n+k-2})^t$$

$$Y = (y_1, \ldots y_{n+k-2})^t$$

Moreover, it is well known (see, for example, the reference) that the system has a unique solution, and A is a totally positive band matrix with lower and upper bandwidth both equal to k, if and only if $A_{ii} \neq 0$ for i=1, ... , n+k-2. In such cases, the Gaussian elimination can be used to compute the solution vector safely without pivoting. Due to the banded structure of matrix A, we store only the (possible) nonzero elements of A in a 2k+1 by n matrix, say A', and perform the Gaussian elimination without pivoting on A' to obtain the solution of the equations.

Comments :

1) The necessary and sufficient condition for the unique existence of interpolating B-splines is that $B_i(x_i) \neq 0$, for all $1 \leq i \leq n+k-2$. If this is not true, function bSplineInterp will return an error message and the computation of the B-spline will not be performed.

2) The output matrices coefMatrix and intervalMatrix are used to store the coefficients of the resultant interpolating B-spline in its piecewise polynomial representation. For example, suppose k =2, u[1] = -1, u[2] = 0 and u[3] = 1 are input values of the order for the B-spline and knots. With three appropriately chosen interpolation points and their corresponding values given by the vectors X and Y, respectively, the function bSplineInterp produces

$$\text{coefMatrix} = \begin{bmatrix} 3.1 & 3.1 \\ -1.0 & 2.4 \end{bmatrix}$$

$$\text{coefInterval} = \begin{bmatrix} -1.0 & 0 \\ 0 & 1.0 \end{bmatrix}$$

Then the computed interpolating B-spline is

$$C(x) = \begin{cases} -1.0x + 3.1 & \text{if } -1 \leq x < 0 \\ 2.4x + 3.1 & \text{if } 0 \leq x \leq 1 \end{cases}$$

REFERENCE

de Boor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978, pp. 199-202

■ comCubicSpline

FUNCTION

coefMatrix = comCubicSpline(xVector, yVector, k1Derivative, knDerivative)

PURPOSE

Generate the (coefficients of the) interpolating complete cubic spline for a given set of n monotonically increasing knots with known first order derivatives at the two end knots.

INPUT

xVector (Real Vector): the n-dimensional vector that contains the n monotonically increasing knots, $x_1 < x_2 < ... < x_n$

yVector (Real Vector): the n-dimensional vector that contains the y values at the knots, $y_1, y_2, ..., y_n$

k1Derivative (Real Scalar): the value of the derivative at the knot t_1

knDerivative (Real Scalar): the value of the derivative at the knot t_n

OUTPUT

coefMatrix (Real Matrix): the 4 by n-1 matrix containing the coefficients of the piecewise cubic polynomial $P_k(x)$ in the interval $[x_k, x_{k+1}]$, i.e.,

$$P_k(x) = C_{4,k}x^3 + C_{3,k}x^2 + C_{2,k}x + C_{1,k}$$

for all x in $[x_k, x_{k+1}]$

EXAMPLES

```
//  An example for: comCubicSpline(xVector, yVector,
//  k1Derivative, knDerivative), the complete cubic
//  spline fitting knots and end-point derivative data

//  The knot data is:
xVector = { 1, 2, 3, 4, 5 };
yVector = { 2, 5, 10, 17, 26 };
//  The derivative value at the 1st knot is k1Derivative = 3:
k1Derivative = 3;
//  The derivative value at the nth knot is knDerivative = -2:
knDerivative = -2;

coefMatrix = comCubicSpline(xVector, yVector, k1Derivative, knDeriva-
tive);

//  Results:
// coefMatrix:  4 rows, 4 columns
//   (1st Row)  -2.89285714285714   11.3928571428571
//                            -76.3571428571429    641.357142857143

//    (2nd Row) 7.73214285714286  -13.6964285714286
//                      74.0535714285714   -464.232142857143

//   (3rd Row) -3.78571428571429   6.92857142857143
//                     -22.3214285714286   112.25

//   (4th Row) 0.946428571428571 -0.839285714285714
//                       2.41071428571429  -8.80357142857143
```

ALGORITHM AND COMMENTS

The cubic spline is determined by solving the n-dimensional tridiagonal linear system equations, Aw = b, with

$$A_{i,i} = 2(h_{i-1} + h_i)$$

$$A_{i,i-1} = h_{i-1}$$

$$A_{i,i+1} = h_i \qquad \text{for } i = 1, ..., n-2$$

$$A_{1,1} = 2h_1$$

$$A_{n,n} = 2h_{n-1}$$

and

$$b_i = \frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \qquad \text{for } i = 2, ..., n-1$$

$$b = \frac{y_2 - y_1}{h_1} - s_1$$

$$b = s_n - \frac{y_n - y_{n-1}}{h_{n-1}}$$

$$w = (y''_1, ..., y''_n)^t$$

where y''_i denotes the second order derivative at knot x_i. The resultant cubic polynomial in the interval $[x_i, x_{i+1}]$, for i=1,...,n-1, is

$$P_i(x) = C_{4,i}x^3 + C_{3,i}x^2 + C_{2,i}x + C_{1,i}$$

where

$$C_{4,i} = \frac{y''_{i+1} - y''_i}{6h_i}$$

$$C_{3,i} = \frac{y''_i}{2}$$

$$C_{2,i} = \frac{y_{i+1} - y_i - C_{3,i}h_i^2 - C_{4,i}h_i^3}{h_i}$$

$$C_{1,i} = y_i$$

REFERENCE

DeBoor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978, pp. 53-55

■ evalBSplineInterp

FUNCTION

interpVector = evalBSplineInterp(k, u, xVector, yVector, xCoordVector)

PURPOSE

Evaluate the B-spline of order k (=2, 3 or 4) for a set of m points, where the B-spline is defined on a set of n monotonically increasing knots and n+k-2 appropriately chosen monotonically increasing interpolation points and their corresponding function values

INPUT

k (Integer Scalar): the order (=2, 3 or 4) of the B-spline

u (Real Vector): the n-dimensional vector containing monotonically increasing knots, i.e., u[i] < u[i+1] for i=1, ... , n-1

xVector (Real Vector): the (n+k-2)-dimensional vector containing the n+k-2 interpolation points

yVector (Real Vector): the (n+k-2)-dimensional vector containing the n+k-2 interpolation points and their corresponding function values

xCoordVector (Real Vector): the m-dimensional vector containing the m points whose function values are to be approximated from the B-spline interpolation

OUTPUT

interpVector (Real Vector): the m-dimensional vector containing the resultant B-spline evaluation at points given by xCoordVector. That is, the ith component of interpVector $w_i = C(v_i)$, where v is xCoordVector and

C(x) denotes the desired (interpolation) B-spline

EXAMPLES

```
//  An example for: evalBSplineInterp(k, u, xVector, yVector,
//   xCoordVector), the B-spline evaluation function

//  The order of the spline is k = 3:
k = 3;
//  The knot vector is:
u = {1, 3, 5, 10};
//  The interpolation data is:
xVector ={ 2, 4, 6, 8, 9};
yVector ={4, 8, 12, 16, 18};
//  The function evaluation vector is:
xCoordVector = { 1, 2, 3, 4, 5, 5.5};

interpVector = evalBSplineInterp(k, u, xVector, yVector,
xCoordVector);

//  Results:
//  interpVector:  6 rows
//              2
//              4
//              6
//              8
//             10
//             11
```

ALGORITHM AND COMMENTS

The function evalBSplineInterp calls the function bSplineInterp to establish the interpolation B-spline before evaluating the values at xCoordVector[i].

REFERENCE

de Boor, C.: *A Practical Guide to Splines,* Springer-Verlag, New York, 1978, pp. 108 , 199-202

■ evalComCubicSpline

FUNCTION

interpVector = evalComCubicSpline(xVector, yVector, k1Derivative, knDerivative, xCoordVector)

PURPOSE

Approximate the function values for a set of m points using the interpolating complete cubic spline generated from a sequence of n monotonically increasing knots and their corresponding values

INPUT

xVector (Real Vector): the n-dimensional vector containing the n monotonically increasing knots, $x_1 < x_2 < ... < x_n$

yVector (Real Vector): the n-dimensional vector containing the values $y_1, y_2, ..., y_n$ corresponding to the knot values

k1Derivative (Real Scalar): the value of the derivative at the knot t_1

knDerivative (Real Scalar): the value of the derivative at the knot t_n

xCoordVector (Real Vector): the m-vector containing the m points at which the function values are to be approximated

OUTPUT

interpVector (Real Vector): the m-dimensional vector containing the resultant function values at the points prescribed by the components of xCoordVector

EXAMPLES

```
//  An example for: evalComCubicSpline(xVector, yVector,
//  k1Derivative, knDerivative, xCoordVector), the evaluation
//  of the complete cubic spline interpolant

//  The interpolation data is:
xVector = { 1, 2, 3, 4, 5 };
yVector = { 2, 5, 10, 17, 26 };
//  The derivative value at the 1st knot is k1Derivative = 2:
k1Derivative = 2;
//  The derivative value at the nth knot is knDerivative = 10:
knDerivative = 10;
//  The function evaluation vector is:
xCoordVector = { 1, 1.7, 2.5, 3.5, 4.5, 5 };

interpVector = evalComCubicSpline(xVector, yVector,
k1Derivative,
knDerivative, xCoordVector);
//  Results:
//  interpVector:  6 rows
//                    2
//                    3.89
//                    7.25
//                   13.25
//                   21.25
//                   26
```

ALGORITHM AND COMMENTS

See the algorithm for the comCubicSpline function.

REFERENCE

DeBoor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978, pp. 53-55

■ evalHermInterp

FUNCTION

interpVector = evalHermInterp(xVector, yVector, xCoordVector)

PURPOSE

Approximate the function values for a set of m points using the nth-order Hermite interpolation polynomial from a given set of n data points

INPUT

xVector (Real Vector: the n-dimensional vector containing the x coordinates of the n data points being used to construct the Hermite interpolation polynomial

yVector (Real Vector): the n-dimensional vector containing the y coordinates of the n data points being used to construct the Hermite interpolation polynomial. Note that the ith data point is simply (xVector[i], yVector[i])

xCoordVector (Real Vector): the m-dimensional vector containing the m points whose function values are to be approximated

OUTPUT

interpVector (Real Vector): the m-dimensional vector containing the resultant function values at the points prescribed by the components of xCoordVector

EXAMPLES

```
// An example for: evalHermInterp(xVector, yVector,
// xCoordVector), the evaluation of the Hermite polynomial
// interpolant

//  The interpolation data is:
xVector = {1, 2, 3, 4, 5.};
yVector = {2, 5, 10, 17, 26.};
//  The function evaluation vector is:
xCoordVector = { 1.5, 2.5, 3.5, 4.5, 4.75, 5};

interpVector = evalHermInterp(xVector, yVector, xCoordVector);

//  Results:
//  interpVector:  6 rows
//                      3.25
//                      7.25
//                     13.25
```

```
//                    21.25
//                    23.5625
//                    26
```

ALGORITHM AND COMMENTS

The computation of the Hermite interpolation polynomial is based on the use of Newton's formula with divided differences. The evaluation of the function value at each point is accomplished in a manner similar to the efficient Horner's scheme.

REFERENCE

Stoer, J. and Bulirsch, R., *Introduction to Numerical Analysis*, Springer-Verlag, 1980, pp. 52-55

■ evalLagrInterp

FUNCTION

interpVector = evalLagrInterp(xVector, yVector, xCoordVector)

PURPOSE

Approximate the function values for a set of m points using the nth-order Lagrange interpolation polynomial from a given set of n data points

INPUT

xVector (Real Vector): the n-dimensional vector that contains the x coordinates of the n data points used to construct the Lagrange interpolation polynomial

yVector (Real Vector): the n-dimensional vector that contains the y coordinates of the n data points used to construct the Lagrange interpolation polynomial

xCoordVector (Real Vector): the m-dimensional vector containing the m points where the function values are to be approximated

OUTPUT

interpVector (Real Vector): the m-dimensional vector containing the resultant function values at the points prescribed by the components of xCoordVector

EXAMPLES

```
// An example for: evalLagrInterp(xVector, yVector,
// xCoordVector), the evaluation of the Lagrange polynomial
// interpolant

//  The interpolation data is:
xVector = {1, 2, 3, 4, 5.};
yVector = {2, 5, 10, 17, 26.};
//  The function evaluation vector is:
```

```
xCoordVector = { 1.5, 2.5, 3.5, 4.5, 4.75, 5};

interpVector = evalLagrInterp(xVector, yVector, xCoordVector);

//  Results:
//  interpVector:  6 rows
//                      3.25
//                      7.25
//                     13.25
//                     21.25
//                     23.5625
//                     26
```

ALGORITHM AND COMMENTS

The computation of the Lagrange interpolation polynomial is based on the use of Newton's formula with divided differences (see the reference below). The evaluation of the function value at each point is accomplished in a similar manner to the efficient Horner's scheme.

REFERENCE

Stoer, J. and Bulirsch, R., *Introduction to Numerical Analysis*, Springer-Verlag, 1980, pp. 43-47

■ evalNatCubicSpline

FUNCTION

interpVector = evalNatCubicSpline(xVector, yVector, xCoordVector)

PURPOSE

Approximate the function values for a set of m points using the interpolating natural cubic spline from a sequence of n monotonically increasing knots and their corresponding values

INPUT

xVector (Real Vector): the n-dimensional vector containing the n monotonically increasing knots, $x_1 < x_2 < ... < x_n$

yVector (Real Vector): the n-dimensional vector containing the values $y_1, y_2, ..., y_n$ corresponding to the knot values

xCoordVector (Real Vector): the m-vector containing the m points at which the function values are to be approximated

OUTPUT

interpVector (Real Vector): the m-dimensional vector containing the resultant function values at the points prescribed by the components of xCoordVector

EXAMPLES

```
//  An example for: evalNatCubicSpline(xVector, yVector,
//  xCoordVector), the evaluation of the natural cubic
//  spline interpolant

//  The interpolation data is:
xVector = {1, 2, 3, 4, 5.};
yVector = {2, 5, 10, 17, 26.};
//  The function evaluation vector is:
xCoordVector = { 1.5, 2.5, 3.5, 4.5, 4.75, 5};

interpVector = evalNatCubicSpline(xVector, yVector, xCoordVector);

//  Result :
//  interpVector:  6 rows
//         3.33928571428571
//         7.23214285714286
//         13.2321428571429
//         21.3392857142857
//         23.6495535714286
//         26
```

ALGORITHM AND COMMENTS

See natcubicspline.

REFERENCE

DeBoor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978, pp. 53-55

■ evalPiecePolyInterp

FUNCTION

interpVector = evalPiecePolyInterp(k, u, yVector, xCoordVector)

PURPOSE

Evaluate the piecewise polynomial of degree k (≥1) for a set of m points, where the piecewise polynomial is defined on a set of n monotonically increasing knots and their corresponding function values, where n=pk+1 for some positive integer p

INPUT

k (Integer Scalar): the degree (≥1) of the interpolating piecewise polynomial

u (Real Vector): the n-dimensional vector containing monotonically increasing knots, i.e., u[i] < u[i+1] for i=1, ... , n-1

yVector (Real Vector): the n-dimensional vector containing the function values of the given n knots u[1], ... , u[n]

xCoordVector (Real Vector): the m-dimensional vector containing the m points whose functions values are to be approximated from the interpolating piecewise polynomial

OUTPUT

interpVector (Real Vector): the m-dimensional vector containing the resultant piecewise polynomial evaluation at points given by the input vector xCoordVector. That is, interpVector[i] = P(xCoordVector[i]), where P(x) denotes the interpolating piecewise polynomial.

EXAMPLES

```
//  An example for: evalPiecePolyInterp(k, u, yVector,
// xCoordVector), the evaluation of the piecewise polynomial //inter-
polant

//  The degree of the polynomial is k = 2:
k = 2;
//  The knots are located in:
u = {1, 2, 3, 4, 5.};
//  The values at the knots are:
yVector = {2, 5, 10, 17, 26.};
//  The function evaluation vector is:
xCoordVector = { 1.5, 2.5, 3.5, 4.5, 4.75, 5};

interpVector = evalPiecePolyInterp(k, u, yVector,
xCoordVector);

//  Results:
//  interpVector:  6 rows
//                        3.25
//                        7.25
//                       13.25
//                       21.25
//                       23.5625
//                       26
```

ALGORITHM AND COMMENTS

The function evalPiecePolyInterp calls the function piecePolyInterp to establish the interpolating piecewise polynomial before evaluating the values at xCoordVector[i].

REFERENCE

Prenter, P.M. : *Spline and Variation Methods*, John Wiley and Sons, New York, 1975, p. 48

■ evalRatInterp

FUNCTION

interpVector = evalRatInterp(xVector, yVector, xCoordVector)

PURPOSE

Approximate the function values for a set of m points using a nth-order interpolation rational function from a given set of n data points

INPUT

xVector (Real Vector): the n-dimensional vector containing the x coordinates of the n data points used to construct the rational interpolation function

yVector (Real Vector): the n-dimensional vector containing the y coordinates of the n data points used to construct the rational interpolation function. Note that the ith data point is simply (xVector[i] , yVector[i])

xCoordVector (Real Vector): the m-dimensional vector containing the m points whose function values are to be approximated

OUTPUT

interpVector (Real Vector): the m-dimensional vector containing the resultant function values at the points prescribed by the components of xCoordVector

EXAMPLES

```
//  An example for: evalRatInterp(xVector, yVector,
// xCoordVector), the evaluation of the rational function
//  interpolant

//  The interpolation data is:
xVector = {1, 2, 3, 4, 5.};
yVector = {2, 5, 10, 17, 26.};
//  The function evaluation vector is:
xCoordVector = { 1.5, 2.5, 3.5, 4.5, 4.75, 5};

interpVector = evalRatInterp(xVector, yVector, xCoordVector);

//  Results:
//  interpVector:  6 rows
//                     3.25
//                     7.25
//                    13.25
//                    21.25
//                    23.5625
//                    26
```

ALGORITHM AND COMMENTS

The computation of the diagonal rational interpolation function is based on the use of the continued fraction formula with inverse differences.

REFERENCE

Stoer, J. and Bulirsch, R., I*ntroduction to Numerical Analysis*, Springer-Verlag, 1980, pp. 63-65

■ formLSFit

FUNCTION

[A, bVector] = formLSFit(basis, n, xVector, yVector, weight);

PURPOSE

Generate the matrix A and vector b for the general least-squares problem i.e., to

$$\text{minimize } \|Ac - b\|$$

such that

$$(Ac-b)^t(Ac-b) = \sum_{i=1}^{m} w_i \left[\sum_{j=1}^{n} c_j \beta_j(x_i) - y_i \right]^2$$

where m , n ≥ 1, ||•|| denotes the Euclidean norm in R^m, $x_1, \ldots, x_m$ are the given data points, $y_1, \ldots, y_m$ are the measurements, $w_1, \ldots, w_m$ are the positive weights, $ß_1, \ldots, ß_n$ are the basis functions and $c_1, \ldots, c_n$ are the unknown parameters

INPUT

basis (function): the basis functions (form: basis(i,x), i = function index, x = real vector)

n (Integer Scalar): the number of basis functions

xVector (Real Vector): the m-dimensional vector containing the data points $x_1, \ldots, x_m$

yVector (Real Vector): the m-dimensional vector containing the measurements $y_1, \ldots, y_m$

weight (Real Vector): the m-dimensional vector containing all the positive weights $w_1, \ldots, w_m$

OUTPUT

A (Real Matrix): the desired m by n matrix for the general linear least-squares approximation problem

bVector (Real Vector): the desired m-dimensional vector for the general linear least-squares approximation problem

EXAMPLES

```
//  An example for: formLSFit(basis, n, xVector,
//  yVector, weight),
//  the general linear least squares fit function

//  The number of basis functions is n = 3:
n = 3;
//  The data points are:
xVector = {.2, .4, .6, .8, 1.0, 1.2};
//  The measurements are:
yVector = { 3.507, 3.387, 4.267, 5.147, 6.027, 6.907};
//  The positive weights are:
weight = {1, 1, 1, 1, 1, 1};

function basis (ind, x)
  s = (x-.2)/.2;
  select ind from
   case 1:
    return (1);
   case 2:
    return (1 -2*s/5.);
   case 3:
    return (1 - 3*s/2 + 3*s*s/10);
  end select;
end function;

[A, bVector] = formLSFit(basis, n, xVector, yVector, weight);

//  Results :
//  A:  6 rows, 3 columns
//    1              1              1
//    1            0.6           -0.2
//    1            0.2           -0.8
//    1           -0.2           -0.8
//    1           -0.6           -0.2
//    1             -1              1
//  bVector:  6 rows
//                  3.507
//                  3.387
//                  4.267
//                  5.147
//                  6.027
//                  6.907
```

ALGORITHM AND COMMENTS

The matrix A and b are defined as follows :

$$a_{i,j} = \sqrt{w_i}\beta_j(x_i)$$

$$b_i = \sqrt{w_i}y_i$$

for $1 \le i \le m$, $1 \le j \le n$.

Comments:

1) The matrix A and b are used as inputs of SVD followed by SVDFit for statistical function fitting.

2) The matrix A and b are also used as inputs for the functions fullRankLS and defRankLS.

■ formPolyFit

FUNCTION

[A, bVector] = formPolyFit(n, xVector, yVector, weight)

PURPOSE

Generate the matrix A and vector b for the least-squares nth-degree polynomial approximation problem, i.e., to

$$\text{minimize } \|Ac - b\|$$

such that

$$(Ac-b)^t(Ac-b) = \sum_{i=1}^{m} w_i \left[\sum_{j=1}^{n+1} c_j x_i^{j-1} - y_i \right]^2$$

where $m \ge 1$, $n \ge 0$, $\|\bullet\|$ denotes the Euclidean norm in R^m, $x_1, ..., x_m$ are the given data points, $y_1, ..., y_m$ are the measurements, $w_1, ..., w_m$ are the positive weights and $c_1, ..., c_{n+1}$ are the unknown parameters

INPUT

n (Integer Scalar): the degree of the approximating polynomial

xVector (Real Vector): the m-dimensional vector containing the data points $x_1, ..., x_m$

yVector (Real Vector): the m-dimensional vector containing the measurements $y_1, ..., y_m$

weight (Real Vector): the m-dimensional vector containing all the positive weights $w_1, ..., w_m$

OUTPUT

A (Real Matrix): the desired m by n+1 matrix for the least-squares polynomial approximation problem

bVector (Real Vector): the desired m-dimensional vector for the least-squares polynomial approximation problem

EXAMPLES

```
//  An example for: formPolyFit(n, xVector, yVector, weight),
//  the least-squares nth-degree polynomial approximation

//  The degree of the approximating polynomial is n = 3:
n = 3;
//  The data points are:
xVector ={ 2, 4., 8., 9.};
//  The measurements are:
yVector ={5, 17.,65., 82.};
//  The positive weights are:
weight = {1,1,1,1};

[A, bVector] = formPolyFit(n, xVector, yVector, weight);

//  Results:
//  A:  4 rows, 4 columns
//        1            2           4             8
//        1            4          16            64
//        1            8          64           512
//        1            9          81           729
//
//  bVector:  4 rows
//                      5
//                     17
//                     65
//                     82
```

ALGORITHM AND COMMENTS

The matrix A and b are defined as follows :

$$a_{i,j} = \sqrt{w_i}\, x_i^{j-1}$$

$$b_i = \sqrt{w_i}\, y_1$$

for $1 \le i \le m$, $1 \le j \le n+1$.

Comments:

1) The matrix A and b are used as inputs of SVD followed by SVDFit for statistical function fitting.

2) The matrix A and b are also used as inputs for the functions fullRankLS and defRankLS.

■ genFit

FUNCTION

[nonlinCoefficients, nonlinGradient, nonlinResidual] = genFit(nonlinFct, xVector, yVector, weight, initCoefficients, maxIterations)

PURPOSE

Solve the general nonlinear least-squares problem, i.e., to

$$\text{minimize} \sum_{i=1}^{m} w_i \left[F(A, x_i) - y_i \right]^2$$

using the Marquardt-Levenberg method. Here w_i is the ith positive weight, x_i is the ith data point with corresponding measurement y_i, for i=1, ... , n and $F(A,x_i)$ is a prescribed nonlinear function where A is an (unknown) n-dimensional vector ($n \leq m$), which is defined on all or a subset of the real numbers containing $x_1, \ldots, x_m$

INPUT

nonlinFct (function): the m-valued function which computes $F(A,x_i)$ simultaneously for i =1, ... , m.

xVector (Real Vector): the m-dimensional vector containing the data points $x_1, \ldots, x_m$

yVector (Real Vector): the m-dimensional vector containing the measurement values $y_1, \ldots, y_m$

weight (Real Vector): the m-dimensional vector containing the m positive weights $w_1, \ldots, w_m$

initCoefficients (Real Vector): the n-dimensional vector containing the initial guess for the solution of the given nonlinear least-squares problem

maxIterations (Integer Scalar): the number of iterations for establishing a check point to confirm the continuation of Marquardt-Levenberg method

OUTPUT

nonlinCoefficients (Real Vector): the n-dimensional vector containing the computed solution of the given nonlinear least-squares problem.

nonlinGradient (Real Vector): the n-dimensional vector containing the gradient vector of the minimizing functional at the solution vector A*.

nonlinResidual (Real Scalar): the sum of the squares of the least-squares residuals

EXAMPLES

```
//  Example of using genFit(nonlinFct, xVector, yVector,
//  weight,initCoefficients, maxIterations)

//  The data points are:
xVector = {1, 2, 3, 4, 5, 6};
```

```
//  The measurements are:
yVector = {3, 6, 12, 18,27,38};
//  The weights are:
weight = {1, 1, 1, 1, 1, 1};
//  The initial guess for the coefficients is:
initCoefficients = {0, 1, -1};
//  The iteration check value is:
maxIterations = 10;

function nonlinFct(a, x)
   for i = 1 to 6 do
      y[i] = a[1]*(x[i])^2+exp(a[2]*x[i])-a[3];
   end for;
   return y;
end function;

[nonlinCoefficients, nonlinGradient, nonlinResidual] =
genFit(nonlinFct, xVector, yVector, weight, initCoefficients,
      maxIterations);

//  Results :
//  nonlinCoefficients:  3 rows
//        0.924927462416745
//        0.216974847978868
//       -1.01859609578672
//
//  nonlinGradient:  3 rows
//       -1.42082191227199e-10
//       -2.63475690375457e-11
//        1.69478980444071e-12
//
//  nonlinResidual:          0.700184868422663
```

ALGORITHM AND COMMENTS

The function

$$\sum_{i=1}^{m} w_i [F(A, x_i) - y_i]^2$$

which is being minimized can be represented by the form of

$$S(A) = \sum_{i=1}^{m} [f_i(A)]^2 = f^t(A) f(A)$$

where $f(A)=(f_1(A),...,f_m(A))^t$ is the m-dimensional vector function with $f_i(A)$ an n-dimensional functional such that

$$f_i(A) = \sqrt{w_i}\,[F(A, x_i) - y_i]$$

Note that a (local) minimum A* of the functional S(A) must be a saddle point, i.e., the gradient of S(A) , $g(A)=2J^t(A)f(A)$, is the zero vector at A* where J(A) denotes the Jacobian matrix with

$$[J(A)]_{ij} = \frac{\partial}{\partial A_j} f_i(A)$$

(For illustration convenience, from now on, we shall use f and J to represent f(A) and J(A), respectively, when the context is clear).

The Marquardt-Levenberg method is an iterative algorithm which combines the the Steepest Descent method for minimizing S(A) and the Cauchy-Newton method for finding a zero gradient, g(A), of S(A). With generic modifications, the Marquardt-Levenberg method overcomes the slow convergence of Steepest Descent and the scaling and singularity problem with J^tJ often encountered in Cauchy-Newton methods. For a more detailed theoretical discussion of the basic algorithm, the reference listed below. A summary of the algorithm steps (with modifications for the implementation of this function genFit) is given as follows.

Suppose that an initial guess A_0 for the solution is given.

Step 1: (Initialization)

Set $\mu = 0.0001$.

Compute f at A_0, and $S_0 = f^Tf$.

Step 2:

Compute the Jacobian matrix J at A_0.

Step 3:

Construct the (symmetric) matrix

$$B = J^tJ + \mu D$$

and the vector

$$e = -J^tf$$

where D is the diagonal matrix with

$$D_{ii} = (J^TJ)_{ii} + 1.$$

Step 4:

If B is not positive definite

set μ to be 10μ, then go to Step 3.

Otherwise, go to Step 5.

Step 5: Solve $B(\Delta A) = e$.

Set $A_1 = A_0 + \Delta A$.

Step 6: If $A_1 = A_0$ (numerically),

then go to Step 9.

Otherwise, go to Step 7.

Step 7: Compute f at A_1, and $S_1 = f^T f$.

Step 8: If $S_1 \geq S_0$,

set μ to be 10μ, then go to Step 3.

Otherwise,

set S_0 to be S_1 , A_0 to be A_1, and μ to be 0.4μ,

then go to Step 2.

Step 9: Set $A^*=A_0$, and $S^*=S_0$.

Terminate.

Comments:

1) The Jacobian matrix J(A) is evaluated numerically within the function genFit using the central difference formula, i.e.,

$$[J(A)]_{ij} = \frac{f_i(A + h_j\varepsilon_j) - f_i(A - h_j\varepsilon_j)}{2h_j}$$

where ε_j is the jth unit vector,

$$\varepsilon_j = \left(A_j + \frac{1}{\mu_3}\right)\frac{1}{\mu_3}$$

and μ is the machine precision.

2) To avoid wasting computational effort due to the slow convergence of some improperly chosen initial guess, a default lower bound, 100, is set for miter in the function to confirm continuing the process of the Marquardt-Levenberg procedure. In other words, the input miter can overwrite the default value only if it is larger than 100.

3) Due to the nonlinearity of the problem and unavoidable roundoff errors using real arithmetic (vs. interval arithmetic) on a digital computer, the method may converge incidently to some point which is not a saddle point. Thus a final check on the gradient vector at the solution is highly recommended. It is also important to be aware that success in using this function sometimes depends on how close the initial guess is to the solution. So, it is always worth taking effort and time to obtain a "good" initial guess for the solution from any available resources before using the function.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*, Adam Hilger Ltd., Bristol, 1979, pp. 174-177

■ hermInterp

FUNCTION

coefVector = hermInterp(xVector, yVector)

PURPOSE

Generate the (coefficients of the) Hermite interpolation polynomial from a given set of n data points

INPUT

xVector (Real Vector): the n-dimensional vector that contains the x coordinates of the n data points used to construct the Hermite interpolation polynomial

yVector (Real Vector): the n-dimensional vector that contains the y coordinates of the n data points used to construct the Hermite interpolation polynomial. Note that the ith data point is simply (xVector[i], yVector[i])

OUTPUT

coefVector (Real Vector): the n-dimensional vector containing the coefficients of the resultant Hermite interpolation polynomial $P_n(x)$ of order n (i.e., degree n-1) such that:

$$P_n(x) = C_n x^{n-1} + C_{n-1} x^{n-2} + ... + C_1$$

EXAMPLES

```
//  An example for: hermInterp(xVector, yVector), the
//  Hermite polynomial interpolant coefficients

//  The data points are:
xVector = {1, 2, 3, 4, 5.};
yVector = {2, 5, 10, 17, 26.};

coefVector = hermInterp(xVector, yVector);

//  Results :
//  coefVector:  5 rows
//                        1
//                        0
//                        1
//                        0
//                        0
```

ALGORITHM AND COMMENTS

The computation of the coefficients for the Hermite interpolation polynomial is based on the use of Newton's formula with divided differences.

Comments :

1)The duplication of the components in the input vector X indicates the values of successively different

orders of derivatives. For example, if $X=(1, 1, 1, 2, 3, 3)^t$ and $Y=(1, 2, 3, 4, 5, 6)$ are the input vectors, then the resultant Hermite interpolation polynomial, denoted by $H_6(x)$, must be the sixth order (i.e., 5th degree) polynomial such that

$$H_6(1) = 1 \qquad H'_6(1) = 2 \qquad H''_6(1) = 3$$

$$H_6(2) = 4$$

$$H_6(3) = 5 \qquad H'_6(3) = 6$$

2) To use the function correctly, the duplication of the components in Vector X must be grouped together; for example, $X=(1, 2, 1, 1)^t$ is not appropriate input for this function, but $X=(1, 1, 1, 2)^t$ or $(2, 1, 1, 1)^t$ are both acceptable by the function as long as the Y vector contains the correct corresponding values.

REFERENCE

Stoer, J. and Bulirsch, R., *Introduction to Numerical Analysis*, Springer-Verlag, 1980, pp. 52-55

■ lagrInterp

FUNCTION

coefVector = lagrInterp(xVector, yVector)

PURPOSE

Generate the (coefficients of the) Lagrange interpolation polynomial from a given set of n data points

INPUT

xVector (Real Vector): the n-dimensional vector that contains the x coordinates of the n data points used to construct the Lagrange interpolation polynomial

yVector (Real Vector): the n-dimensional vector that contains the y coordinates of the n data points used to construct the Lagrange interpolation polynomial. Note that the ith data point is simply (xVector[i], yVector[i])

OUTPUT

coefVector (Real Vector): the n-dimensional vector containing the coefficients of the resultant Langrange interpolation polynomial $P_n(x)$ of order n (i.e., of degree n-1) such that:

$$P_n(x) = C_n x^{n-1} + C_{n-1} x^{n-2} + ... + C_1$$

EXAMPLES

```
//  An example for: lagrInterp(xVector, yVector), the
//  Lagrange polynomial interpolation coefficients

//  The data points are:
```

```
xVector = {1, 2, 3, 4, 5.};
yVector = {2, 5, 10, 17, 26.};

coefVector = lagrInterp(xVector, yVector);

//  Results:
//  coefVector:  5 rows
//                        1
//                        0
//                        1
//                        0
//                        0
```

ALGORITHM AND COMMENTS

The computation of the coefficients for the Lagrange interpolation polynomial is based on the efficient solution of the Vandermonde system. The algorithm uses the "master" polynomial

$$P_{n+1}(x) = (x - x_1)(x - x_2)...(x - x_n)$$

with synthetic division on each factor ($x-x_i$) to form the inverse of a Vandermonde matrix column by column. Such an algorithm requires only $\mathbf{O}(n^2)$ arithmetic operations to construct the inverse and solution.

Comments :

Due to the ill-conditioning of Vandermonde matrices, the computed coefficients may contain large errors when the number of points (or the order of the interpolation polynomial) gets large (say > 5, for example).

REFERENCE

Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T., *Numerical Recipes in C,* Cambridge University Press, 1988, pp. 52-54

■ lineFit

FUNCTION

[lineFitCoefficients,lineFitResidual] = lineFit(xVector, yVector)

PURPOSE

Compute the least-squares line approximation (i.e., $y = A_2x+A_1$) for a set of n (≥ 2) points $x_1, ..., x_n$ and their corresponding estimated function values $y_1, ..., y_n$

INPUT

xVector (Real Vector): a set of n (≥ 2) data points $x_1, ..., x_n$

yVector (Real Vector): the n estimated function values $y_1, ..., y_n$

OUTPUT

lineFitCoefficients (Real Vector): the 2-dimensional vector containing the coefficients of the resultant least-squares line, i.e., $y=A_2x+A_1$

lineFitResidual (Real Scalar): the sum of the squares of the least-square residuals

EXAMPLES

```
//  An example for: lineFit(xVector, yVector),
//  the least-squares linear approximation function

//  The data points are:
xVector = {1, 2, 3};
          //  The estimated function values are:
yVector = { 5.5, 4.0, 3.2};

[lineFitCoefficients, lineFitResidual] = lineFit(xVector, yVector);

//  Results:
//  lineFitCoefficients:  2 rows
//         6.53333333333333
//        -1.15
//
//  lineFitResidual:         0.0816666666666667
```

ALGORITHM AND COMMENTS

The coefficients of the least-squares line (i.e., $y=A_2x+A_1$) for a set of n data pairs $(x_1, y_1), \ldots, (x_n,y_n)$ are obtained from the following formulas:

$$A_1 = \frac{\left(\sum_{i=1}^{n} x_i^2\right)\left(\sum_{i=1}^{n} y_i\right) - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} x_i y_i\right)}{n\left(\sum_{i=1}^{n} x_i^2\right) - \left(\sum_{i=1}^{n} x_i\right)^2}$$

and

$$A_2 = \frac{n\left(\sum_{i=1}^{n} x_i y_i\right) - \left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{n\left(\sum_{i=1}^{n} x_i^2\right) - \left(\sum_{i=1}^{n} x_i\right)^2}$$

REFERENCE

Spiegel, M.R., *Schaum's Outline Series : Theory and Problems of Statistics*, 2nd. ed., McGraw-Hill, New York, 1988, pp. 266-267

■ natCubicSpline

FUNCTION

coefMatrix = natCubicSpline(xVector, yVector)

PURPOSE

Generate the (coefficients of the) interpolating natural cubic spline for a given set of n monotonically increasing knots with vanishing second order derivatives at the two end knots

INPUT

xVector (Real Vector): the n-dimensional vector that contains the n monotonically increasing knots, $x_1 < x_2 < ... < x_n$

yVector (Real Vector): the n-dimensional vector that contains the y values at the knots, $y_1, y_2, ..., y_n$

OUTPUT

coefMatrix (Real Matrix): the 4 by n-1 matrix containing the coefficients of the piecewise cubic polynomial $P_k(x)$ in the interval $[x_k, x_{k+1}]$, i.e.,

$$P_k(x) = C_{4,k}x^3 + C_{3,k}x^2 + C_{2,k}x + C_{1,k}$$

for all x in $[x_k, x_{k+1}]$

EXAMPLES

```
//  An example for: natCubicSpline(xVector, yVector), the
//  natural cubic spline interpolation coefficients

//  The data points are:
xVector = {1, 2, 3, 5};
```

```
yVector = {2, 5, 10, 26};

coefMatrix = natCubicSpline(xVector, yVector);

//  Results:
//  coefMatrix:  4 rows, 3 columns
// -1                    1.78260869565217     8.82608695652174
//  3.78260869565217    -0.391304347826087   -7.43478260869565
// -1.17391304347826     0.91304347826087     3.26086956521739
//  0.391304347826087    0.0434782608695652  -0.217391304347826 \
```

ALGORITHM AND COMMENTS

The cubic spline is determined by solving the (n-2)-dimensional tridiagonal linear system of equations, Aw = b, with

$$A_{i,i} = 2(h_i + h_{i+1})$$

$$A_{i+1,1} = A_{i,i+1} = h_{i+1} \qquad \text{for } i = 1, ..., n-2$$

$$b_i = \frac{y_{i+2} - y_{i+1}}{h_{i+1}} - \frac{y_{i+1} - y_i}{h_i} \qquad \text{for } i = 2, ..., n-1$$

$$w = (y''_1, ..., y''_n)^t$$

where y''_i denotes the second order derivative at knot x_i. The resultant cubic polynomial in the interval $[x_i, x_{i+1}]$, for i=1, ... , n-1, is

$$P_i(x) = C_{4,i}x^3 + C_{3,i}x^2 + C_{2,i}x + C_{1,i}$$

where

$$C_{4,i} = \frac{y''_{i+1} - y''_i}{6h_i}$$

$$C_{3,i} = \frac{y''_i}{2}$$

$$C_{2,i} = \frac{y_{i+1} - y_i - C_{3,i}h_i^2 - C_{4,i}h_i^3}{h_i}$$

$$C_{1,i} = y_i$$

REFERENCE

DeBoor, C., *A Practical Guide to Splines*, Springer-Verlag, 1978, pp. 53-55

■ piecePolyBasis

FUNCTION

[coefMatrix, intervalMatrix, basisVector] = piecePolyBasis(k, u)

PURPOSE

Generate the cardinal basis for the continuous piecewise polynomial of degree k (≥1) from a set of n monotonically increasing knots with n = pk+1 for some positive integer p

INPUT

k (Integer Scalar): the degree (≥1) of the desired cardinal basis

u (Real Vector): the n-dimensional vector containing the monotonically increasing knots, i.e., u[i] < u[i+1] for i=1, ... , n-1

OUTPUT

coefMatrix (Real Matrix): the (k+1) by n-1+(n-1)/k matrix containing the coefficients of the piecewise polynomials of degree k for the nonzero supports of the cardinal basis

intervalMatrix (Real Matrix): the 2 by n-1+(n-1)/k matrix containing the intervals of the piecewise polynomial of degree k for the nonzero supports of the cardinal basis corresponding to the columns of the output matrix coefMatrix

basisVector (Integer Vector): the integer vector of dimension n-1+(n-1)/k containing the indices of the cardinal basis corresponding to the columns of coefMatrix

EXAMPLES

```
//  An example for: piecePolyBasis(k, u), a piecewise
//  polynomial basis

//  The degree of the basis is k = 2:
k = 2;
//  The knot vector is:
u = {1, 2, 3, 4, 5.};

[coefMatrix, intervalMatrix, basisVector] = piecePolyBasis(k, u);
//  Results:
//  coefMatrix:  3 rows, 6 columns
//     3        -3       1        10        -15       6
//    -2.5       4      -1.5      -4.5        8      -3.5
//     0.5      -1       0.5       0.5       -1       0.5
//
//  intervalMatrix:  6 rows
```

```
// 1
// 2
// 3
// 3
// 4
// 5
//
//  basisVector:  2 rows, 6 columns
//     1         1         1         3         3         3
//     3         3         3         5         5         5
```

ALGORITHM AND COMMENTS

For a set of n knots, u[1] < ... < u[n], the cardinal basis is a set containing n piecewise polynomials of degree k, each having the nonzero support on an interval containing no more than 2k+1 knots. In addition, the ith cardinal basis function $C_i(x)$ has to satisfy the condition:

$$C_i(u[j]) = \delta_{ij}$$

where δ_{ij} is the Kronecker delta which equals 1 if i=j and equals 0 whenever i≠j.

A complete, explicit expression of the cardinal basis for a continuous piecewise polynomial of degree k with n=pk+1 knots, u[i], i=1, ... , n, can be summarized as follows:

$$C_1(x) = \begin{cases} \prod_{i=2}^{k+1} \frac{x-u[i]}{u[1]-u[i]} & \text{if } x \in [u[1], u[k+1]] \\ 0 & \text{otherwise} \end{cases}$$

$$C_n(x) = \begin{cases} \prod_{i=n-k}^{n-1} \frac{x-u[i]}{u[n]-u[i]} & \text{if } x \in [u[n-k], u[n]] \\ 0 & \text{otherwise} \end{cases}$$

For m= pk+j ≠ 1, n; we have

Case 1: j=1.

$$C_m(x) = \begin{cases} \prod_{i=1-k}^{0} \frac{x - u[pk+i]}{u[m] - u[pk+i]} & \text{if } x \in [u[m-k], u[m]] \\ \prod_{i=2}^{k+1} \frac{x - u[pk+i]}{u[m] - u[pk+i]} & \text{if } x \in [u[m], u[m+k]] \\ 0 & \text{otherwise} \end{cases}$$

Case 2: j=2, ... , k.

$$C_m(x) = \begin{cases} \prod_{i=1, i\neq j}^{k+1} \frac{x - u[pk+i]}{u[m] - u[pk+i]} & \text{if } x \in [u[pk+1], u[pk+k+1]] \\ 0 & \text{otherwise} \end{cases}$$

Comments :

The output of function piecePolyBasis consists of two matrices, coefMatrix, intervalMatrix, and an integer vector basisVector. They specify only the nonzero supports of the cardinal basis. In other words, any subinterval composed by the two adjacent knots from the original knot sequence, u[1], ... , u[n], having zero function values over it will not be shown in coefMatrix, intervalMatrix and basisVector.

For example, suppose that the input arguments are k=1, u[1] = -1, u[2] = 0 and u[3] = 1, then after calling the function piecePolyBasis(k,u), we will obtain the following outputs:

$$\text{coefMatrix} = \begin{bmatrix} 0 & 1.0 & 1.0 & 0 \\ -10 & 1.0 & -1.0 & 1.0 \end{bmatrix}$$

$$\text{coefInterval} = \begin{bmatrix} -1.0 & -1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 1.0 \end{bmatrix}$$

and

$$\text{basisVector} = \begin{bmatrix} 1 & 2 & 2 & 3 \end{bmatrix}^t$$

The values given in coefMatrix, intervalMatrix, and basisVector represent the following cardinal basis

$$C_1(x) = \begin{cases} -1.0x + 0.0 & \text{if } -1 \le x < 0 \\ 0 & \text{if } 0 \le x \le 1 \end{cases}$$

$$C_2(x) = \begin{cases} 1.0x + 1.0 & \text{if } -1 \le x < 0 \\ -1.0x + 1.0 & \text{if } 0 \le x \le 1 \end{cases}$$

and

$$C_3(x) = \begin{cases} 0 & \text{if } -1 \le x < 0 \\ 1.0x + 0.0 & \text{if } 0 \le x \le 1 \end{cases}$$

REFERENCE

Prenter, P.M. : *Spline and Variation Methods*, John Wiley and Sons, New York, 1975, p. 48

■ piecePolyInterp

FUNCTION

[coefMatrix, intervalMatrix] = piecePolyInterp(k, xVector, yVector)

PURPOSE

Generate the continuous piecewise polynomial interpolation of degree k (≥1) from a set of n monotonically increasing knots and their corresponding function values, where n=pk+1 for some positive integer p

INPUT

k (Integer Scalar): the degree (≥1) of the desired piecewise polynomial interpolation

xVector (Real Vector): the n-dimensional vector containing the monotonically increasing knots, i.e., u[i] < u[i+1] for i=1, ... , n-1

yVector (Real Vector): the n-dimensional vector containing the function values at the given n knots u[1], ... , u[n]

OUTPUT

coefMatrix (Real Matrix): the (k+1) by (n-1)/k matrix containing the coefficients of the interpolating piecewise polynomial of degree k in the subintervals being specified by the output matrix intervalMatrix

intervalMatrix (Real Matrix): the 2 by (n-1)/k matrix containing the intervals of the interpolating piecewise polynomial of degree k corresponding to the columns of the output matrix coefMatrix

EXAMPLE

```
//  An example for: piecePolyInterp(k, xVector, yVector), the
//  piecewise polynomial interpolant

//  The degree of the interpolant is:
k = 2;
//  The interpolation data is:
xVector = {1, 2, 3, 4, 5.};
```

```
yVector = {2, 5, 10, 17, 26.};

[coefMatrix, intervalMatrix] = piecePolyInterp (k, xVector, yVector);

//  Results:
//  coefMatrix:  3 rows, 2 columns
//                       1            1
//                       0            0
//                       1            1
//
//   intervalMatrix:  2 rows, 2 columns
//                       1            3
//                       3            5
```

ALGORITHM AND COMMENTS

The interpolation piecewise polynomial of degree k defined on n knots, u[1],... , u[n], and their values y_1, ... , y_n are given by

$$P(x) = \sum_{j=1}^{n} \alpha_j C_j(x)$$

such that

$$P(u[i]) = y_i$$

for i=1, ... , n, where $C_1(x)$, ... , $C_n(x)$ are the cardinal basis functions.

Since $C_i(u[j]) = \delta_{ij}$ (the Kronecker delta), it is easy to see that $a_i=y_1$ for i=1, … n.

Comments :

The output matrices coefMatrix and intervalMatrix are used to store the coefficients of the resultant interpolating piecewise polynomial of degree k in the following manner. Suppose k =1, u[1] = -1, u[2] = 0, u[3] = 1, y1= 4.1, y2= 3.1 and y3=5.5 are the input degrees of the piecewise polynomial, knots and the function values at knots. After calling the function piecePolyInterp, we have

$$\text{coefMatrix} = \begin{bmatrix} 3.1 & 3.1 \\ -1.0 & 2.4 \end{bmatrix}$$

$$\text{intervalMatrix} = \begin{bmatrix} -1.0 & 0 \\ 0 & 1.0 \end{bmatrix}$$

Then the computed interpolating piecewise polynomial is

$$P(x) = \begin{cases} -1.0x + 3.1 & \text{if } -1 \le x < 0 \\ 2.4x + 3.1 & \text{if } 0 \le x \le 1 \end{cases}$$

REFERENCE

Prenter, P.M. : *Spline and Variation Methods*, John Wiley and Sons, New York, 1975, p. 48

■ ratInterp

FUNCTION

[numcoefVector, dencoefVector] = ratInterp(xVector, yVector)

PURPOSE

Generate the (coefficients of the) diagonal interpolation rational function, $P_{nk}(x)/Q_{dk}(x)$ from a given set of n data points, where $P_{nk}(x)$ and $Q_{dk}(x)$ are polynomials of order nk and dk respectively

INPUT

xVector (Real Vector): the n-dimensional vector that contains the x coordinates of the n data points used to construct the diagonal interpolation rational function

yVector (Real Vector): the n-dimensional vector that contains the y coordinates of the n data points used to construct the diagonal interpolation rational function. Note that the ith data point is simply (xVector[i], yVector[i]).

OUTPUT

numcoefVector (Real Vector): the nk-dimensional vector containing the coefficients of the numerator polynomial $P_{nk}(x)$ of order nk (i.e., degree nk-1):

$$P_{nk}(x) = C_{nk}x^{nk-1} + C_{nk-1}x^{nk-2} + ... + C_1$$

for the resultant rational function, where nk=[n/2] + 1 with [···] representing the integer part of a real number

dencoefVector (Real Vector): the dk-dimensional vector containing the coefficients of the denominator polynomial $Q_{dk}(x)$ of order dk (i.e., degree dk-1):

$$Q_{dk}(x) = D_{dk}x^{dk-1} + D_{dk-1}x^{dk-2} + ... + D_1$$

for the resultant rational function, where dk=n + 1 - nk and nk is the order of the numerator polynomial $P_{nk}(x)$

EXAMPLES

```
//  An example for: ratInterp(xVector, yVector), the diagonal
//  rational function interpolant
```

```
//  The interpolation data is:
xVector = {1, 2, 3, 4};
yVector = {0, 0.333, 1, 1.8};

[numcoefVector, dencoefVector] = ratInterp(xVector, yVector);

//  Results:
//  dencoefVector:  2 rows
//        0.986547085201794
//        1.00149476831091
//
//  numcoefVector:  3 rows
//        1.00448430493274
//       -2.00448430493274
//        1
```

ALGORITHM AND COMMENTS

The computation of the interpolation diagonal rational function is based on the use of the continued fraction formula with inverse differences.

REFERENCE

Stoer, J. and Bulirsch, R., *Introduction to Numerical Analysis*, Springer-Verlag, 1980, pp. 63-65

■ SVDFit

FUNCTION

[SVDCoefficients, SVDResidual] = SVDFit(U, V, S, bVector, tolerance)

PURPOSE

To compute the solution for a general linear least-squares problem, i.e., to minimize ||Bx - Y||using the singular value decomposition, where B is a m by n matrix with $m \geq n$, Y is an m-dimensional vector, and ||•|| is the Euclidean norm on R^n

INPUT

U (Real Matrix): the orthogonal matrix U obtained from the singular value decomposition $B = USV^t$

V (Real Matrix): the orthogonal matrix V obtained from the singular value decomposition $B = USV^t$

S (Real Vector): the n-dimensional vector containing all the singular values (arranged in descending order) of the n by n diagonal matrix S* in the singular value decomposition $B = USV^t$

bVector (Real Vector): the m-dimensional vector used to form the least-squares problem

tolerance (Real Scalar): the nonnegative tolerance using to determine when singular values are equal to zero

OUTPUT

SVDCoefficients (Real Vector): the n-dimensional vector containing the solution of the given least-squares problem

SVDResidual (Real Scalar): the sum of the squares of the residuals for the solution vector x of the least-squares problem

EXAMPLES

```
//  An example for: SVDFit(U, V, S, bVector, tolerance), the general
//  linear least-squares fit using the singular value decomposition

//  The singular values from the SVD are:
S ={899.283494598676; 11.4313592915513;
    1.22580300919018; 0.133319630433653};
//  The least-squares vector is:
bVector = {5;17; 65; 82};
//  The singular value tolerance is:
tolerance = 1e-11;
//  The orthogonal matrix U from the SVD is:
U = {-0.0093819,  0.3134102,   -0.8062042,  -0.5017179;
     -0.0728034,  0.7947699,   -0.0630924,   0.5992160;
     -0.5738214,  0.3870653,    0.4880459,  -0.5317148;
     -0.8156841, -0.3468362,   -0.3284287,   0.3263414};
//  The orthogonal matrix V from the SVD is:
V = {-0.0016365,  0.1004613,   -0.5789506,  -0.8091483;
     -0.0136127,  0.3307478,   -0.7474851,   0.5759224;
     -0.1156447,  0.9315085,    0.3246147,  -0.1163768;
     -0.9931960, -0.1131608,   -0.0265982,   0.0069902};


[SVDCoefficients, SVDResidual] = SVDFit(U, V, S, bVector,
tolerance);

//  Results:
//  SVDCoefficients:  4 rows
//        0.999961829193046
//        2.31698435697502e-05
//        0.999995747979319
//        2.17082752904899e-07
//
//  SVDResidual:       1.64870020759814e-09
```

ALGORITHM AND COMMENTS

The solution for minimizing ||Bx - Y|| via the singular value decomposition of B is given by:

$$x = VS^{+}U^{t}b$$

where

$$S^{+} = \begin{cases} \frac{1}{S_{ii}} & \text{if } S_{ii} > \text{tol} \\ 0 & \text{otherwise} \end{cases}$$

with $B=USV^{t}$ the singular value decomposition.

Comments:

To solve the least-squares problem , i.e., to minimize ||BX-Y|| via the singular value decomposition, the function SVD() should be called to obtain the desired input matrices U,V and the vector S before using the function SVDFit.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation,* Adam Hilger Ltd., Bristol, 1979, pp. 32-35

CHAPTER 21
GRAPHICAL FUNCTIONS

■ addPlot

FUNCTION

addPlot(G, P)

PURPOSE

Display a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

P (Plot): the entered 2- or 3-dimensional plot

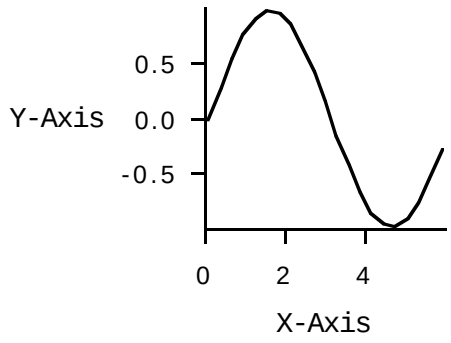
OUTPUT

None (Display of the input plot P on the desired graph G in a Graph Editor window)

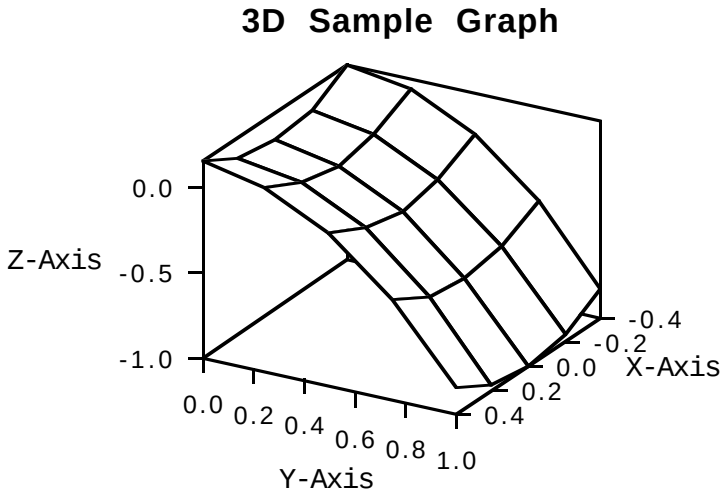
EXAMPLES

```
// Examples for: addplot(G, P)

// Perform addPlot( G2, P2) where G2 is the
// 2-dimensional  graph generated by new2DGraph and P2 is
// the 2-dimensional plot generated by new2DDataPlot for the
// data set  {(xi, yi)}  with yi = sin(xi) and xi = (i-1)*π/10

G2 = new2DGraph("2D Sample Graph");
x = {  0;   0.299; 0.598;  0.898; 1.197; 1.496; 1.795;
     2.094; 2.394; 2.693;  2.992; 3.291; 3.590; 3.890;
     4.189; 4.488; 4.787;  5.086; 5.386; 5.685; 5.984};
y = {  0;   0.295; 0.563;  0.782; 0.931;  0.997; 0.975;
     0.866; 0.680; 0.434;  0.149;-0.149; -0.434;-0.680;
    -0.866;-0.975;-0.997; -0.931;-0.782; -0.563;-0.295};
P2 = new2DDataPlot("  ", x, y);
addPlot(G2, P2);

// Result: (Display of the resultant plot P2 in graph G2)
```

2D Sample Graph




```
// Perform addPlot( G3, P3) where G3 is the
// 3-dimensional  graph generated by new3DGraph and P3 is
// the 3-dimensional plot generated by new3DDataPlot for the
// data set {(xi, yj, zij)} with zij = xi^2 - yj^2

G3 = new3DGraph("3D Sample Graph");

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {    0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ", x, y, z);
addPlot(G3, P3);

// Result: (Display of the resultant plot P3 in graph G3)
```



SEE ALSO

new2DDataPlot, new3DDataPlot, new2DGraph, new3DGraph

ALGORITHM AND COMMENTS

If graph G and plot P have different dimensions, then an error message will be returned.

■ fitToWindow

FUNCTION

fitToWindow (G)

PURPOSE

Fit a graph, i.e., center and scale the graph, inside the Graph Editor window

INPUT

G (Graph): the entered 2- or 3-dimensional graph

OUTPUT

None (The graph G is redrawn to fit inside the Graph Editor window)

EXAMPLES

```
// Example for: fitToWindow(G)
```

```
// Perform fitToWindow(G2) where G2 is the 2-dimensional
// graph generated by the function new2DGraph

G2 = new2DGraph("fitToWindowGraph");
x = {  0;   0.299; 0.598;  0.898; 1.197; 1.496; 1.795;
     2.094; 2.394; 2.693;  2.992; 3.291; 3.590; 3.890;
     4.189; 4.488; 4.787;  5.086; 5.386; 5.685; 5.984};
y = {  0;   0.295; 0.563;  0.782; 0.931;  0.997; 0.975;
     0.866; 0.680; 0.434;  0.149;-0.149; -0.434;-0.680;
    -0.866;-0.975;-0.997; -0.931;-0.782; -0.563;-0.295};
P2 = new2DDataPlot("  ", x, y);
addPlot(G2, P2);
fitToWindow(G2);

// Result: NOT DISPLAYED

// Perform fitToWindow(G3) where G3 is the 3-dimensional
// graph generated by the  function  new3DGraph

G3 = new3DGraph("fitToWindow3DGraph");
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ", x, y, z);
addPlot(G3, P3);
fitToWindow(G3);

//  Result:  NOT DISPLAYED
```

SEE ALSO

new2DDataPlot, new3DDataPlot, new2DGraph, new3DGraph, addPlot

■ focusCamera

FUNCTION

focusCamera (G, p)

PURPOSE

Change the perspectivity to view a 3-dimensional graph in the perspective projection mode

INPUT

G (Graph): the entered 3-dimensional graph

p (Real Scalar): the perspectivity of the graph with p satisfying 0.01 < p < 0.99

OUTPUT

None (The graph G is redrawn to the new perspectivity)

EXAMPLE

```
// An example for: focusCamera(G, p)

// Perform focusCamera(G3,p) where G3 is the 3-dimensional
// graph generated  by the function new3DGraph with
// perspective projection type and p = 0.5

G3 = new3DGraph("focusCameraGraph");
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ", x, y, z);
addPlot(G3, P3);
setProjectionType(G3,<perspective>);
p3= 0.5;
f3 = focusCamera(G3, p);

// Result :
// f3:   NONE DISPLAYED
```

SEE ALSO

setProjectionType, new3DGraph, new3DDataPlot, addPlot,

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph or the current projection type is not perspective, then an error message will be returned.

■ getAxisFlag

FUNCTION

flag = getAxisFlag (G, axisIndex, attribute)

PURPOSE

Query the attribute of an axis of the graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

axisIndex (Integer Scalar): the index of the axis whose limits are queried where axisIndex = 0 for x-axis, axisIndex = 1 for y-axis and axisIndex = 2 for z-axis

attribute (Integer Scalar): the attribute of axis axisIndex of graph G (contained in the HiQ-Script language constants — see the Algorithm and Comments section below)

OUTPUT

flag (Integer Scalar): the (boolean) value of the queried attribute

EXAMPLES

```
// Examples for: getAxisFlag(G, axisIndex, attribute)

// Perform getAxisFlag(G2, axisIndex2, attribute2) where
// G2 is the 2-dimensional graph generated by the
// function new2DGraph, and axisIndex2 = 1, attribute2 = 2

G2 = new2DGraph("getAxisFlagGraph");
x = {  0;   0.299; 0.598;  0.898; 1.197; 1.496; 1.795;
     2.094; 2.394; 2.693;  2.992; 3.291; 3.590; 3.890;
     4.189; 4.488; 4.787;  5.086; 5.386; 5.685; 5.984};
y = {  0;   0.295; 0.563;  0.782; 0.931;  0.997; 0.975;
     0.866; 0.680; 0.434;  0.149;-0.149; -0.434;-0.680;
    -0.866;-0.975;-0.997; -0.931;-0.782; -0.563;-0.295};
P2 = new2DDataPlot("  ", x, y);
addPlot(G2, P2);
axisIndex2 = 1;
attribute2 = 2;
flag2 = getAxisFlag(G2, axisIndex2, attribute2);

// Result:
// flag2:  1

// Perform getAxisFlag(G3, axisIndex3, attribute3) where
// G3 is the 3-dimensional graph generated by the
// function new3DGraph, and axisIndex3 = 2, attribute3 = 2

G3 = new3DGraph("getAxisFlag3DGraph");
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
```

```
P3 = new3DDataPlot("   ", x, y, z);
addPlot(G3, P3);
axisIndex3 = 2;
attribute3 = 2;
flag3 = getAxisFlag(G3, axisIndex3, attribute3);

// Result:
// flag3:    1
```

SEE ALSO

new2DDataPlot, new3DDataPlot, new2DGraph, new3DGraph, addPlot, setAxisFlag

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ-Script Language Constants (in the HiQ folder); they are: <x_axis>, <y_axis>, <z_axis>.

The following Language Constants are available for getAxisFlag():

`<vertical_title>` = 32

= <true> means axis title is displayed vertically instead of horizontally

`<vertical_labels>` = 64

= <true> means labels with tick marks/grid lines are displayed vertically

`<reverse_placement>` = 128

= <true> for 2D means: a) x-axis is drawn at the top of the graph instead of the bottom and b) y-axis is drawn on the right side of the graph instead of the left side

= <true> for 3D means: a) x-axis is drawn at the top left instead of the bottom right; b) y-axis is drawn at the top right instead of the bottom left; and c) z-axis is drawn on the right side instead of the left side.

<false> is the default setting for newly created graphs.

■ getAxisLimits

FUNCTION

[lower, upper] = getAxisLimits (G, axisIndex)

PURPOSE

Query the limits of an axis of a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

axisIndex (Integer Scalar): the index of the axis whose limits are queried where axisIndex = 0 for x-axis, axisIndex = 1 for y-axis and axisIndex = 2 for z-axis

OUTPUT

lower, upper (Real Scalar): the lower limit and the upper limit of the axis

EXAMPLES

```
// Examples for: getAxisLimits (G, axisIndex)

// Perform getAxisLimits (G2, axisIndex2) where
// G2 is the 2-dimensional graph generated by the
// function new2DGraph, and axisIndex2 = 0

G2 = new2DGraph("getAxisLimitsGraph");
x = {  0;   0.299; 0.598;  0.898; 1.197; 1.496; 1.795;
     2.094; 2.394; 2.693;  2.992; 3.291; 3.590; 3.890;
     4.189; 4.488; 4.787;  5.086; 5.386; 5.685; 5.984};
y = {  0;   0.295; 0.563;  0.782; 0.931;  0.997; 0.975;
     0.866; 0.680; 0.434;  0.149;-0.149; -0.434;-0.680;
    -0.866;-0.975;-0.997; -0.931;-0.782; -0.563;-0.295};
P2 = new2DDataPlot("  ", x, y);
addPlot(G2, P2);
axisIndex2 = 0;
[lower2, upper2] = getAxisLimits (G2, axisIndex2);

// Results:
// lower2:   -1
// upper2:    1

// Perform getAxisLimits (G3, axisIndex3) where
// G3is the 3-dimensional graph generated by the
// function  new3DGraph, and axisIndex3 = 2

G3 = new3DGraph("getAxisLimits3DGraph");
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ", x, y, z);
addPlot(G3, P3);
axisIndex3 = 2;
[lower3, upper3] = getAxisLimits (G3, axisIndex3);
// Results:
// lower3:   -1
// upper3:    1
```

SEE ALSO

new2DGraph, new2DDataPlot, new3DGraph, new3DDataPlot, addPlot, setAxisLimits

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ-Script Language Constants (in the HiQ folder); they are: <x_axis>, <y_axis>, <z_axis>.

If graph G is a 2-dimensional graph and axisIndex = 2, then an error message will be returned.

■ getAxisMinorTicks

FUNCTION

n = getAxisMinorTicks(G, axisIndex)

PURPOSE

Query the number of minor ticks (between major ticks) of a linearly scaled axis of a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

axisIndex (Integer Scalar) : the index of linearly scaled axis where axisIndex = 0 for x-axis, axisIndex = 1 for y-axis and axisIndex = 2 for z-axis

OUTPUT

n (Integer Scalar): the number of minor ticks between major ticks

EXAMPLES

```
// Examples for: getAxisMinorTicks(G, axisIndex)

// Perform getAxisMinorTicks(G2, axisIndex2) where
// G2 is the 2-dimensional graph generated by the
// function new2DGraph, and axisIndex2 = 0

G2 = new2DGraph("getAxisMinorTicksGraph");
x = {  0;   0.299; 0.598;  0.898; 1.197; 1.496; 1.795;
     2.094; 2.394; 2.693;  2.992; 3.291; 3.590; 3.890;
     4.189; 4.488; 4.787;  5.086; 5.386; 5.685; 5.984};
y = {  0;   0.295; 0.563;  0.782; 0.931;  0.997; 0.975;
     0.866; 0.680; 0.434;  0.149;-0.149; -0.434;-0.680;
    -0.866;-0.975;-0.997; -0.931;-0.782; -0.563;-0.295};
P2 = new2DDataPlot("  ", x, y);
addPlot(G2, P2);
axisIndex2 = 0;
n2 = getAxisMinorTicks(G2, axisIndex2);

// Result:
// n2:    1

// Perform getAxisMinorTicks(G3,Iaxis3) where
```

```
// G3 is the 3-dimensional graph generated by the
// function  new3DGraph, and Iaxis3 = 2

G3 = new3DGraph("getAxisMinorTicks3DGraph");
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ", x, y, z);
addPlot(G3, P3);
Iaxis3 = 2;
n3 = getAxisMinorTicks(G3, Iaxis3);

// Result:    n3:    1
```

SEE ALSO

new2DGraph, new2DDataPlot, new3DGraph, new3DDataPlot, addPlot, setAxisMinorTicks

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ-Script Language Constants (in the HiQ folder); they are: <x_axis>, <y_axis>, <z_axis>.

If graph G is a 2-dimensional graph and axisIndex = 2, then an error message will be returned.

■ getAxisScale

FUNCTION

n = getAxisScale(G, Iaxis)

PURPOSE

Query the scaling mode of an axis of a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

Iaxis (Integer Scalar): the index of the axis whose scaling mode is queried where Iaxis = 0 for x-axis, Iaxis = 1 for y-axis and Iaxis = 2 for z-axis

OUTPUT

n (Integer Scalar): the scaling mode of the graph axis where n = <linear_scale> = 0 for linear scaling and n = <log_scale> = 1 for logarithmic scaling, where <linear_scale> and <log_scale> are
HiQ-Script Language Constants

EXAMPLES

```
// Examples for: getAxisScale(G, Iaxis)

// Perform getAxisScale(G2,Iaxis2) where G2 is
// the 2-dimensional graph generated by the function
// new2DGraph, and Iaxis2 = 0

G2 = new2DGraph("getAxisScaleGraph");
x = {  0;   0.299; 0.598;  0.898; 1.197; 1.496; 1.795;
     2.094; 2.394; 2.693;  2.992; 3.291; 3.590; 3.890;
     4.189; 4.488; 4.787;  5.086; 5.386; 5.685; 5.984};
y = {  0;   0.295; 0.563;  0.782; 0.931;  0.997; 0.975;
     0.866; 0.680; 0.434;  0.149;-0.149; -0.434;-0.680;
    -0.866;-0.975;-0.997; -0.931;-0.782; -0.563;-0.295};
P2 = new2DDataPlot("  ", x, y);
addPlot(G2, P2);
Iaxis2 = 0;
n2 = getAxisScale(G2,Iaxis2);

// Result:
// n2: 0

// Perform getAxisScale(G3,Iaxis3) where G3 is
// the 3-dimensional graph generated by the function
// new3DGraph, and Iaxis3 = 1

G3 = new3DGraph("getAxisScale3DGraph");
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ", x, y, z);
addPlot(G3, P3);
Iaxis3 = 1;
n3 = getAxisScale(G3,Iaxis3);

// Result:
// n3:  0
```

SEE ALSO

new2DGraph, new2DDataPlot, new3DGraph, new3DDataPlot, addPlot, setAxisScale

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ-Script Language Constants (in the HiQ folder); they are: <x_axis>, <y_axis>, <z_axis>.

If graph G is a 2-dimensional graph and Iaxis = 2, then an error message will be returned.

■ getAxisTitle

FUNCTION

title = getAxisTitle (G, Iaxis)

PURPOSE

Query the title of an axis of a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

Iaxis (Integer Scalar): the index of the axis whose title is queried where Iaxis = 0 for x-axis, Iaxis = 1 for y-axis and Iaxis = 2 for z-axis

OUTPUT

title (String): the title of the queried axis

EXAMPLES

```
// Examples for: getAxisTitle (G, Iaxis)

// Perform getAxisTitle(G2,Iaxis2) where G2 is
// the 2-dimensional graph generated by the function
// new2DGraph, and Iaxis2 = 0

G2 = new2DGraph("getAxisTitleGraph");
x = {  0;   0.299; 0.598;  0.898; 1.197; 1.496; 1.795;
     2.094; 2.394; 2.693;  2.992; 3.291; 3.590; 3.890;
     4.189; 4.488; 4.787;  5.086; 5.386; 5.685; 5.984};
y = {  0;   0.295; 0.563;  0.782; 0.931;  0.997; 0.975;
     0.866; 0.680; 0.434;  0.149;-0.149; -0.434;-0.680;
    -0.866;-0.975;-0.997; -0.931;-0.782; -0.563;-0.295};
P2 = new2DDataPlot("  ", x, y);
addPlot(G2, P2);
Iaxis2 = 0;
title2 = getAxisTitle(G2,Iaxis2);

// Result:
// title2:  X-Axis

// Perform getAxisTitle(G3,Iaxis3) where G3 is
// the 3-dimensional graph generated by the function
// new3DGraph, and Iaxis3 = 2

G3 = new3DGraph("getAxisTitle3DGraph");
```

```
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ", x, y, z);
addPlot(G3, P3);
Iaxis3 = 2;
title3 = getAxisTitle(G3,Iaxis3);

// Result:
// title3:  Z-Axis
```

SEE ALSO

new2DGraph, new2DDataPlot, new3DGraph, new3DDataPlot, addPlot, setAxisTitle

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ-Script Language Constants (in the HiQ folder); they are: <x_axis>, <y_axis>, <z_axis>.

If graph G is a 2-dimensional graph and Iaxis = 2, then an error message will be returned.

■ getGraphDimension

FUNCTION

n = getGraphDimension (G)

PURPOSE

Query the dimension of a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

OUTPUT

n (Integer Scalar): the dimension of the input graph G where n = 2 for a 2-dimensional graph or 3 for a 3-dimensional graph

EXAMPLES

```
// Examples for: getGraphDimension(G)

// Perform getGraphDimension(G2) where G2 is
// the 2-dimensional graph generated by the function
```

```
// new2DGraph

G2 = new2DGraph("getGraphDimensionGraph");
n2 = getGraphDimension(G2);

// Result:
// n2:  2

// Perform getGraphDimension(G3) where G3 is
// the 3-dimensional graph generated by the function
// new3DGraph

G3 = new3DGraph("getGraphDimension3DGraph");
n3 = getGraphDimension(G3);

// Result:
// n3:  3
```

■ getGraphFlag

FUNCTION

flag = getGraphFlag (G, attribute)

PURPOSE

Query an attribute of a graph.

INPUT

G (Graph): the entered 2- or 3-dimensional graph

attribute (Integer Scalar): the attribute to be queried where attribute = 1 for hiding graph title, attribute = 2 for hiding axes and their annotation, attribute = 4 for hiding grids, and attribute = 8 for hiding axes annotation (title, labels and ticks)

OUTPUT

flag (Integer Scalar): the (boolean) value of the queried attribute

EXAMPLES

```
// Examples for: getGraphFlag(G,attribute)

// Perform getGraphFlag(G2,attribute2) for attribute2 = 1
// where G2 is the 2-dimensional graph generated by
// the function new2DGraph

G2 = new2DGraph("getGraphFlagGraph");
attribute2 = 2;
flag2 = getGraphFlag (G2, attribute2);
```

```
// Result:
// flag2:    0

// Perform getGraphFlag(G3,attribute3) for attribute3 = 2
// where G3 is the 3-dimensional graph generated by
// the function new3DGraph

G3 = new3DGraph("getGraphFlag3DGraph");
attribute3 = 2;
flag3 = getGraphFlag (G3, attribute3);

// Result:
// flag3:   0
```

SEE ALSO

setGraphFlag

ALGORITHM AND COMMENTS

There are HiQ-Script Language Constants that are equivalent to the numerical values for the attribute parameter; they are:

<hidden_title> = 1 <hidden_axes> = 2 <hidden_grids>=4
<hidden_annotation> = 8

The default attributes for each axis (when a graph is created) are:
hiding graph title:0<false> hiding axes:0<false>
hiding grids:1<true> hiding axis annotation:0<false>

■ getGraphPlotBackfaceMode

FUNCTION

n = getGraphPlotBackfaceMode (G, m)

PURPOSE

Query the backface mode of a 3-dimensional plot in a graph

INPUT

G (Graph): the entered 3-dimensional plot

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

n (Integer Scalar): the backface mode of the m-th plot in graph G where n = 0 for backfaces to be removed and n = 1 for backfaces to be drawn

EXAMPLE

```
// An example for: getGraphPlotBackfaceMode(G,m)

// Perform getGraphPlotBackfaceMode(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotBackfaceMode3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotBackfaceMode(G3, m3);

// Result:
// n3:  1
```

SEE ALSO

setGraphPlotBackfaceMode

ALGORITHM AND COMMENTS

The default backface mode is 1.

If plot P is not a faceted 3-dimensional plot, then an error message will be returned.

■ getGraphPlotContourPlane

FUNCTION

n = getGraphPlotContourPlane (G,m)

PURPOSE

Query the coordinate plane along which contours of a 3-dimensional plot in a graph are to be made

INPUT

G (Graph): the entered 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

OUTPUT

n (Integer Scalar): the coordinate plane to make contours of the m-th plot in graph G where n = <xy_plane> = 0 for xy-plane, n = <yz_plane> = 1 for yz-plane and n = <zx_plane> = 2 for zx-plane, where the <…> indicate HiQ-Script Language Constants

EXAMPLE

```
// An example for: getGraphPlotContourPlane (G,m)

// Perform getGraphPlotContourPlane(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotContourPlaneGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotContourPlane (G3,m3);
// Result:     n3:  0
```

SEE ALSO

setProjectionType

ALGORITHM AND COMMENTS

If plot P is not a faceted 3-dimensional plot, then an error message will be returned.

■ getGraphPlotCoordSystem

FUNCTION

n = getGraphPlotCoordSystem (G,m)

PURPOSE

Query the coordinate system of a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

n (Integer Scalar): the coordinate system of the input plot where
n = <cartesian> = 0 for Cartesian coordinate system, n = <polar> = 1 for polar coordinate system (if P is a 2-dimensional plot) or (<spherical>) spherical coordinate system (if P is a 3-dimensional plot) and m = <cylindrical> = 2 for a cylindrical coordinate system (if P is a 3-dimensional plot); the <…> indicate HiQ-Script Language Constants

EXAMPLES

```
// Examples for: getGraphPlotCoordSystem(G, m)

// Perform getGraphPlotCoordSystem(G2,m2) for m2 = 2
// where G2 isthe 2-dimensional graph containing 3
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets x, y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getGraphPlotCoorSystemGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
m2 = 2;
n2 = getGraphPlotCoordSystem(G2,m2);

// Result:
// n2:  0

// Perform getGraphPlotCoordSystem(G3,m3) for m3 = 0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,new3DDataPlot
// and addPlot using the data sets x, y, z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotCoordSystem3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
```

```
n3 = getGraphPlotCoordSystem(G3,m3);

// Result:
// n3:  0
```

SEE ALSO

setGraphPlotCoordSystem

■ getGraphPlotDimension

FUNCTION

n = getGraphPlotDimension (G, m)

PURPOSE

Query the dimension of a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

n (Integer Scalar): the dimension of the m-th plot in graph G where n = 2 for a 2-dimensional plot or n = 3 for a 3-dimensional plot

EXAMPLES

```
// Examples for: getGraphPlotDimension(G,m)

// Perform getGraphPlotDimension(G2,m2) for m2 = 2
// where G2 isthe 2-dimensional graph containing 3
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets x, y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getGraphPlotDimensionGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
m2 = 2;
n2 = getGraphPlotDimension(G2,m2);
```

```
// Result:
// n2:  2

// Perform getGraphPlotDimension(G3,m3) for m3 = 0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph, new3DDataPlot
// and addPlot using the data sets x, y, z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotDimension3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotDimension(G3,m3);

// Result:
// n3:  3
```

■ getGraphPlotDisplayFormat

FUNCTION

n = getGraphPlotDisplayFormat (G, m)

PURPOSE

Query the display format of a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

n (Integer Scalar): the display format of the m-th plot in graph G where n = 0 for a curve plot (if P is a 2-dimensional plot) or a surface plot (if P is a 3-dimensional plot) ; n = 1 for a point plot; and n = 2 for a connected plot (if P is a 2-dimensional plot) or a contour plot (if P is a 3-dimensional plot)

EXAMPLES

```
// Examples for: getGraphPlotDisplayFormat(G,m)

// Perform getGraphPlotDisplayFormat(G2,m2) for m2 =2
```

```
// where G2 isthe 2-dimensional graph containing 3
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets x, y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getGraphPlotDisplayFormatGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);  // plot 0
addPlot(G2, P22); //  plot 1
addPlot(G2, P23); //  plot 2
m2 = 2;
n2 = getGraphPlotDisplayFormat(G2,m2);

// Result:
// n2:  0

// Perform getGraphPlotDisplayFormat(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,new3DDataPlot
// and addPlot using the data sets x, y, z listed below
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotDisplayFormat3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotDisplayFormat(G3,m3);

// Result:
// n3:  0
```

SEE ALSO

setGraphPlotDisplayFormat

ALGORITHM AND COMMENTS

The default display format is n = 0. This is a curve plot in 2D and a surface plot in 3D. Each of the values of n are also given as HiQ Language Constants; for instance, <curve> = 0; <surface> = 0; <point> = 1; <connected> = 2; and <contour> = 2.

■ getGraphPlotEdgeMode

FUNCTION

n = getGraphPlotEdgeMode (G, m)

PURPOSE

Query the edge mode of a 3-dimensional plot in a graph

INPUT

G (Graph): the entered 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

n (Integer Scalar): the edge mode of the m-th plot in graph G where n = 0 for edges to be removed and n = 1 for edges to be drawn

EXAMPLES

```
// An example for: getGraphPlotEdgeMode(G,m)

// Perform getGraphPlotEdgeMode(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotEdgeMode");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotEdgeMode(G3,m3);

// Result:
// n3:  1
```

SEE ALSO

setGraphPlotDisplayFormat, setGraphShading, setGraphPlotEdgeMode

ALGORITHM AND COMMENTS

The edge mode is only meaningful for shaded and lighted 3D surface plots, but the value is preserved for wireframe and line graphs. The default setting is n = edgemode = 1.

If plot P is not a faceted 3-dimensional plot, then an error message will be returned.

■ getGraphPlotFillColor

FUNCTION

n = getGraphPlotFillColor (G, m)

PURPOSE

Query the color to fill facets of a 3-dimensional plot in a graph

INPUT

G (Graph): the entered 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

n (Integer Scalar): the fill color of the m-th plot in graph G where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

EXAMPLE

```
// An example for: getGraphPlotFillColor(G,m)

// Perform getGraphPlotFillColor(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotFillColorGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotFillColor(G3,m3);
```

```
// Result:
// n3:  1
```

SEE ALSO

setGraphPlotFillColor, setGraphPlotDisplayFormat

ALGORITHM AND COMMENTS

The fill color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

If plot P is not a faceted 3-dimensional plot, then an error message will be returned.

■ getGraphPlotLineColor

FUNCTION

n = getGraphPlotLineColor (G,m)

PURPOSE

Query the color to draw edges of a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

n (Integer Scalar): the line color of the m-th plot in graph G where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

EXAMPLES

```
// Examples for: getGraphPlotLineColor(G,m)

// Perform getGraphPlotLineColor(G2,m2) for m2 = 1
// where G2 isthe 2-dimensional graph containing 2
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets x, y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getGraphPlotLineColorGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
```

```
addPlot(G2, P21);  // plot 0
addPlot(G2, P22);  // plot 1
m2 = 1;
n2 = getGraphPlotLineColor(G2,m2);

// Result:
// n2:  0

// Perform getGraphPlotLineColor(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotLineColor");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotLineColor(G3,m3);

// Result:
// n3:  0
```

SEE ALSO

setGraphPlotLineColor

ALGORITHM AND COMMENTS

The line color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

■ getGraphPlotLineWidth

FUNCTION

n = getGraphPlotLineWidth (G, m)

PURPOSE

Query the line width to draw edges of a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

OUTPUT

n (Integer Scalar): the line width of the m-th plot in graph G

EXAMPLES

```
// Examples for: getGraphPlotLineWidth(G,m)

// Perform getGraphPlotLineWidth(G2,m2) for m2 = 1
// where G2 isthe 2-dimensional graph containing 2
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets x, y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getGraphPlotLineWidthGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
addPlot(G2, P21);
addPlot(G2, P22);
m2 = 1;
n2 = getGraphPlotLineWidth(G2,m2);

// Result:
// n2:  1

// Perform getGraphPlotLineWidth(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotLineWidth3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotLineWidth(G3,m3);

// Result:
// n3:  1
```

ALGORITHM AND COMMENTS

The user interface for the Graph Editor allows you to set line widths between 1 and 4; line widths greater than 4 can be set using one of the functions setGraphPlotLineWidth() or setPlotLineWidth().

■ getGraphPlotMarkerColor

FUNCTION

n = getGraphPlotMarkerColor (G, m)

PURPOSE

Query the color to draw vertices of a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

n (Integer Scalar): the marker color of the m-th plot in graph G where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow

EXAMPLES

```
// Examples for: getGraphPlotMarkerColor(G,m)

// Perform getGraphPlotMarkerColor(EXGraph2,m2) for m2 = 1
// where EXGraph2 isthe 2-dimensional graph containing 2
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets X, Y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getGraphPlotMarkerColorGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
addPlot(G2, P21);
addPlot(G2, P22);
m2 = 1;
n2 = getGraphPlotMarkerColor(G2,m2);

// Result:
// n2:  0

// Perform getGraphPlotMarkerColor(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing one
```

```
// plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotMarkerColor3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotMarkerColor(G3,m3);

// Result:
// n3:  0
```

ALGORITHM AND COMMENTS

The marker color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

■ getGraphPlotMarkerStyle

FUNCTION

n = getGraphPlotMarkerStyle (G, m)

PURPOSE

Query the marker style to draw vertices of a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

OUTPUT

n (Integer Scalar): the marker style of the m-th plot in graph G where n = 0 for circular-shaped marker, n = 1 for square-shaped marker, n = 2 for diamond-shaped marker, n = 3 for triangular-shaped marker, n = 4 for cross-shaped marker and n = 5 for x-shaped marker (See the Algorithm and Comments section)

EXAMPLES

```
// Examples for: getGraphPlotMarkerStyle(G,m)
```

```
// Perform getGraphPlotMarkerStyle(G2,m2) for m2 = 1
// where G2 isthe 2-dimensional graph containing 2
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets x, y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getGraphPlotMarkerStyleGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
addPlot(G2, P21);
addPlot(G2, P22);
m2 = 1;
n2 = getGraphPlotMarkerStyle(EXGraph2,m2);

// Result:
// n2:  0

// Perform getGraphPlotMarkerStyle(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotMarkerStyle3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotMarkerStyle(G3,m3);

// Result:
// n3:  0
```

ALGORITHM AND COMMENTS

The following HiQ-Script Language Constants can be used for the returned values: <circular>, <square>, <diamond>, <triangular>, <cross>, and <x_shape>,

■ getGraphPlotProjectedContour

FUNCTION

n = getGraphPlotProjectedContour (G, m)

PURPOSE

Query the contour placement of a 3-dimensional plot in a graph to project on the contour coordinate plane or overlay on the plot itself

INPUT

G (Graph): the entered 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

n (Integer Scalar): the contour placement of the m-th plot in graph G where n = 0 for overlaid contours and n = 1 for projected contours

EXAMPLES

```
// An example for: getGraphPlotProjectedContour(G,m)

// Perform getGraphPlotProjectedContour(G3,m3) for
// m3 =0 where G3 isthe 3-dimensional graph containing
// one  plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotProjectedContourGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotProjectedContour(G3,m3);

// Result:
// n3:  0
```

SEE ALSO

setGraphPlotProjectedContour

ALGORITHM AND COMMENTS

The HiQ-Script Language Constants that are equivalent to the output values of n are: <overlaid> = 0 and <projected> = 1.

If plot P is not a faceted 3-dimensional plot, then an error message will be returned.

■ getGraphPlotTitle

FUNCTION

title = getGraphPlotTitle (G, m)

PURPOSE

Query the title of a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

OUTPUT

title (String): the title of the m-th plot in graph G

EXAMPLES

```
// Examples for: getGraphPlotTitle(G,m)

// Perform getGraphPlotTitle(G2,m2) for m2 = 1
// where G2 isthe 2-dimensional graph containing 3
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets x, y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getGraphPlotTitleGraph");
P21 = new2DDataPlot("2D Plot 1 ",x,y);  // plot 0
P22 = new2DDataPlot("2D Plot 2 ",y,x);  // plot 1
P23 = new2DDataPlot("2D Plot 3 ",x,x);  // plot 2
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
m2 = 1;
title2 = getGraphPlotTitle(G2,m2);

// Result:
// title2:  2D Plot 2
```

```
// Perform getGraphPlotTitle(G3,m3) for m3 = 0
// where G3 isthe 3-dimensional graph containing one
// plot generated by the functions new3DGraph,new3DDataPlot
// and addPlot using the data sets x, y, z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotTitle3DGraph");
P3 = new3DDataPlot("3D Plot 1",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotTitle(G3,m3);

// Result:
// n3:  3D Plot 1
```

■ getGraphPlotTitleColor

FUNCTION

n = getGraphPlotTitleColor (G, m)

PURPOSE

Query the color to draw title of a plot in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

OUTPUT

n (Integer Scalar): the title color of the m-th plot in graph G where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow

EXAMPLES

```
// Examples for: getGraphPlotTitleColor(G,m)

// Perform getGraphPlotTitleColor(G2,m2) for m2 = 1
// where G2 isthe 2-dimensional graph containing 2
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets x, y
// listed below
```

```
x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getGraphPlotTitleColorGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
addPlot(G2, P21);
addPlot(G2, P22);
m2 = 1;
n2 = getGraphPlotTitleColor(G2,m2);

// Result:
// n2:  0  BLACK

// Perform getGraphPlotTitleColor(G3,m3) for m3 =0
// where G3 is the 3-dimensional graph containing one
// plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getGraphPlotTitleColor3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = getGraphPlotTitleColor(G3,m3);

// Result:
// n3:  0  BLACK
```

ALGORITHM AND COMMENTS

The title color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

■ getGraphShading

FUNCTION

n = getGraphShading (G)

PURPOSE

Query the shading mode of a 3-dimensional graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

OUTPUT

n (Integer Scalar): the shading mode of the input graph G where n = 0 for wireframe, n = 1 for hidden-line, n= 2 for shading in height and n = 3 for shading with light sources

EXAMPLE

```
// An example for: getGraphShading(G)

// Perform getGraphShading(G3) where G3
// is the 3-dimensional graph  generated by the functions
// new3DGraph

G3 = new3DGraph("getGraphShadingGraph");
n3 = getGraphShading(G3);
// Result:    n3:  1
```

SEE ALSO

setGraphShading, wireFrameGraph, hiddenLineGraph, heightShading, lightSourceShading

ALGORITHM AND COMMENTS

There are HiQ Language Constants which are equivalent to the returned values of n; they are: <wire> = 0, <line> = 1, <height> = 2, and <light> = 3.

The default shading for 3D graphs is hidden-line (1).

If graph G is not a 3-dimensional graph, then an error message will be returned.

■ getGraphTitle

FUNCTION

title = getGraphTitle (G)

PURPOSE

Query the title of a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

OUTPUT

title (String): the title of the input graph G

EXAMPLES

```
// Examples for: getGraphTitle(G)

// Perform getGraphShading(G2) where G2
// is the 2-dimensional graph generated by the function
// new3DGraph

G2 = new3DGraph("getGraphTitleGraph");
title2 = getGraphTitle(G2);
// Result:    title2:  getGraphTitleGraph

// Perform getGraphShading(G3) where G3
// is the 3-dimensional graph generated by the function
// new3DGraph
G3 = new3DGraph("getGraphTitle3DGraph");
title3 = getGraphTitle(G3);
// Result:    title3:  getGraphTitle3DGraph
```

SEE ALSO

setGraphTitle

■ getNumberOfPlots

FUNCTION

n = getNumberOfPlots (G)

PURPOSE

Query the number of plots included in a graph

INPUT

G (Graph): the entered 2- or 3-dimensional graph

OUTPUT

n (Integer Scalar): the number of plots included in the input graph G

EXAMPLES

```
// Examples for: getNumberOfPlots(G)

// Perform getNumberOfPlots(G2) where G2 is
// the 2-dimensional graph containing 3 plots generated
// by the  functions new2DGraph, new2DDataPlot and
// addPlot using the data sets X, Y listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
```

```
G2 = new2DGraph("getNumberOfPlotsGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
n2 = getNumberOfPlots(G2);

// Result:
// n2:  3

// Perform getNumberOfPlots(G3) where G3 is
//the 3-dimensional graph containing single plot generated
// by the  functions new2DGraph, new2DDataPlot and
// addPlot using the data sets x, y, z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getNumberOfPlots3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
n3 = getNumberOfPlots(G3);

// Result:
// n3:  1
```

■ getPlot

FUNCTION

Q = getPlot (G, m)

PURPOSE

Query the plot symbol of a plot in a graph. This function copies the plot in graph G to plot Q. Plot Q contains the data from the source plot, with the attributes of the copied plot

INPUT

G (Graph): the entered 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G

OUTPUT

Q (Plot): the plot symbol for the m-th plot in G

EXAMPLES

```
// Examples for: getPlot(G,m)

// Perform getPlot(G2,m2) for m2 = 2 where G2 is
// the 2-dimensional graph containing 3 plots generated
// by the  functions new2DGraph, new2DDataPlot and
// addPlot using the data sets X, Y listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("getPlotGraph");
P21 = new2DDataPlot("P.21",x,y);
P22 = new2DDataPlot("P.22",y,x);
P23 = new2DDataPlot("P.23",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
m2 = 2;
Q2 = getPlot(G2,m2);

// Result:
// Q2:  Plot

// Perform getPlot(G3,m3) for m3 = 0 where G3 is
// the 3-dimensional graph containing single plot generated
// by the functions new2DGraph, new2DDataPlot and
// addPlot using the data sets x, y, z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("getPlot3DGraph");
P3 = new3DDataPlot("P.3 ",x,y,z);
addPlot(G3, P3);
m3 = 0;
Q3 = getPlot(G3,m3);

// Result:
// Q3:  Plot
```

■ getPlotCoordSystem

FUNCTION

n = getPlotCoordSystem (P)

PURPOSE

Query the coordinate system of a plot

INPUT

P (Plot): a 2- or 3-dimensional plot

OUTPUT

n (Integer Scalar): the coordinate system of the input plot where n = 0 for Cartesian coordinate system, n =1 for polar coordinate system (if P is a 2-dimensional plot) or spherical coordinate system (if P is a 3-dimensional plot) and m = 2 for cylindrical coordinate system (if P is a 3-dimensional plot)

EXAMPLES

```
//   Examples for: getPlotCoordSystem(P)
//

// Perform getPlotCoordSystem(EXPlot2) where EXPlot2
//  is the 2-dimensional plot generated by the function
// new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
EXPlot2 = new2DDataPlot(" ",x,y);
EXgetPCoordS2 = getPlotCoordSystem(EXPlot2);

// Result:
//    EXgetPCoordS2:  0

// Perform getPlotCoordSystem(EXPlot3) where EXPlot3
//  is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0     0;
        0,   -0.05};
EXPlot3 = new3DDataPlot(" ",x,y,z);
EXgetPCoordS3 = getPlotCoordSystem(EXPlot3);

//  Result:
//    EXgetPCoordS3:  0
```

■ getPlotDisplayFormat

FUNCTION

n= getPlotDisplayFormat (P)

PURPOSE

Query the display format of a plot

INPUT

P (Plot): the entered 2- or 3-dimensional plot

OUTPUT

n (Integer Scalar): the display format of the input plot P where n = 0 for a curve plot (if P is a 2-dimensional plot) or a surface plot (if P is a 3-dimensional plot) ; n = 1 for a point plot; and n = 2 for a connected plot (if P is a 2-dimensional plot) or a contour plot (if P is a 3-dimensional plot)

EXAMPLES

```
// Examples for: getPlotDisplayFormat(P)

// Perform getPlotDisplayFormat(P2) where P2
// is the 2-dimensional plot generated by the function
// new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot("getPlotDisplayFormatGraph",x,y);
n2 = getPlotDisplayFormat(P2);

// Result:
// n2:  0

// Perform getPlotDisplayFormat(P3) where P3
// is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0, 0;
        0, -0.05};
P3 = new3DDataPlot("getPlotDisplayFormat3DGraph",x,y,z);
n3 = getPlotDisplayFormat(P3);

// Result:
// n3:  0
```

SEE ALSO

setPlotDisplayFormat

ALGORITHM AND COMMENTS

Consult the HiQ Language Constants file for the constants equivalent to the output values of n.

■ getPlotFillColor

FUNCTION

n = getPlotFillColor (P)

PURPOSE

Query the color to fill facets of a 3-dimensional plot

INPUT

P (Plot): the entered 3-dimensional plot

OUTPUT

n (Integer Scalar): the fill color of the input plot P where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow

EXAMPLE

```
// An example for: getPlotFillColor(P)

// Perform getPlotFillColor(P3) where P3
// is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0, 0;
        0, -0.05};
P3 = new3DDataPlot(" ",x,y,z);
n3 = getPlotFillColor(P3);
// Result:
// n3:  1    WHITE
```

ALGORITHM AND COMMENTS

The fill color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

If plot P is not a faceted 3-dimensional plot, then an error message will be returned.

■ getPlotLineColor

FUNCTION

n = getPlotLineColor (P)

PURPOSE

Query the color to draw edges of a plot

INPUT

P (Plot): the entered 2- or 3-dimensional plot

OUTPUT

n (Integer Scalar): the line color of the input plot P where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow

EXAMPLES

```
// Examples for: getPlotLineColor(P)

// Perform getPlotLineColor(P2) where P2
// is the 2-dimensional plot generated by the function
// new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot(" ",x,y);
n2 = getPlotLineColor(P2);

// Result:
// n2:  0  BLACK

// Perform getPlotLineColor(P3) where P3
// is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0, 0;
        0, -0.05};
P3 = new3DDataPlot(" ",x,y,z);
n3 = getPlotLineColor(P3);

// Result:
// n3:  0  BLACK
```

ALGORITHM AND COMMENTS

The line color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are:

<black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

■ getPlotLineWidth

FUNCTION

n = getPlotLineWidth (P)

PURPOSE

Query the line width to draw edges of a plot

INPUT

P (Plot): the entered 2- or 3-dimensional plot

OUTPUT

n (Integer Scalar): the line width of the plot; the possible values are: n = 1, 2, 3, and 4, where 4 is the thickest value available from the Graph Editor interface

EXAMPLES

```
// Examples for: getPlotLineWidth(P)

// Perform getPlotLineWidth(P2) where P2
// is the 2-dimensional plot generated by the function
// new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot(" ",x,y);
n2 = getPlotLineWidth(P2);

// Result:
// n2:  1

// Perform getPlotLineWidth(P3) where P3
// is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0, 0;
        0, -0.05};
P3 = new3DDataPlot(" ",x,y,z);
n3 = getPlotLineWidth(P3);

// Result:
// n3:  1
```

SEE ALSO

setPlotLineWidth

ALGORITHM AND COMMENTS

The user interface for the Graph Editor allows you to set line widths between 1 and 4; line widths greater than 4 can be set using one of the functions setGraphPlotLineWidth() or setPlotLineWidth().

■ getPlotMarkerColor

FUNCTION

n = getPlotMarkerColor (P)

PURPOSE

Query the color to draw vertices of a plot

INPUT

P (Plot): the entered 2- or 3-dimensional plot

OUTPUT

n (Integer Scalar): the marker color of the input plot P where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

EXAMPLES

```
// Examples for: getPlotMarkerColor(P)

// Perform getPlotMarkerColor(P2) where P2
// is the 2-dimensional plot generated by the function
// new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot(" ",x,y);
n2 = getPlotMarkerColor(P2);

// Result:
// n2:  0

// Perform getPlotMarkerColor(P3) where P3
// is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z
```

```
x = { 0; -0.2};
y = { 0; 0.25};
z = {   0, 0;
        0, -0.05};
P3 = new3DDataPlot(" ",x,y,z);
n3 = getPlotMarkerColor(P3);

// Result:
// n3:  0
```

ALGORITHM AND COMMENTS

The marker color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

■ getPlotMarkerStyle

FUNCTION

n = getPlotMarkerStyle (P)

PURPOSE

Query the marker style to draw vertices of a plot

INPUT

P (Plot): the entered 2- or 3-dimensional plot

OUTPUT

n (Integer Scalar): the marker style of the input plot P where n = 0 for circular-shaped marker, n = 1 for square-shaped marker, n = 2 for diamond-shaped marker, n = 3 for triangular-shaped marker, n = 4 for cross-shaped marker and n = 5 for x-shaped marker (See the Algorithm and Comments section)

EXAMPLES

```
// Examples for: getPlotMarkerStyler(P)

// Perform getPlotMarkerStyle(P2) where P2
// is the 2-dimensional plot generated by the function
// new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot(" ",x,y);
n2 = getPlotMarkerStyle(P2);

// Result:
// n2:  0
```

```
// Perform getPlotMarkerStyle(P3) where P3
// is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0, 0;
        0, -0.05};
P3 = new3DDataPlot(" ",x,y,z);
n3 = getPlotMarkerStyle(P3);

// Result:
// n3:  0
```

ALGORITHM AND COMMENTS

The following HiQ-Script Language Constants can be used for the returned values: <circular>, <square>, <diamond>, <triangular>, <cross>, and <x_shape>,

■ getPlotTitle

FUNCTION

title = getPlotTitle (P)

PURPOSE

Query the title of a plot

INPUT

P (Plot): the entered 2- or 3-dimensional plot

OUTPUT

title (String): the title of the plot

EXAMPLES

```
// Examples for: getPlotTitle(P)

// Perform getPlotTitle(P2) where P2
// is the 2-dimensional plot generated by the function
// new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot("getPlotTitleGraph",x,y);
title2 = getPlotTitle(P2);
```

```
// Result:
// title2:  getPlotTitleGraph

// Perform getPlotTitle(P3) where P3
// is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0, 0;
        0, -0.05};
P3 = new3DDataPlot("getPlotTitle3DGraph",x,y,z);
n3 = getPlotTitle(P3);

// Result:   n3:  getPlotTitle3DGraph
```

SEE ALSO

setPlotTitle

ALGORITHM AND COMMENTS

Note: Plot titles are displayed for 3D plots only if the optional graph legend is requested.

■ getPlotTitleColor

FUNCTION

n = getPlotTitleColor (P)

PURPOSE

Query the color to draw title of a plot

INPUT

P (Plot): the entered 2- or 3-dimensional plot

OUTPUT

n (Integer Scalar): the title color of the input plot P where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

EXAMPLES

```
// Examples for: getPlotTitleColor(P)

// Perform getPlotTitleColor(P2) where P2
// is the 2-dimensional plot generated by the function
// new2DDataPlot using the following data sets x,y
```

```
x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot("getPlotTitleColorGraph",x,y);
n2 = getPlotTitleColor(P2);

// Result:   n2:  0  BLACK

// Perform getPlotTitleColor(P3) where P3
// is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0, 0;
        0, -0.05};
P3 = new3DDataPlot("getPlotTitleColor3DGraph",x,y,z);
n3 = getPlotTitleColor(P3);

// Result:
// n3:  0  BLACK
```

SEE ALSO

setPlotTitleColor

ALGORITHM AND COMMENTS

The title color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

■ getProjectionType

FUNCTION

n = getProjectionType (G)

PURPOSE

Query the projection type of a 3-dimensional graph

INPUT

G (Graph): the input graph

OUTPUT

n (Integer Scalar): the projection type of the input graph G where n = 0 for orthographic projection and n = 1 for perspective projection

EXAMPLES

```
 //   An example for: getProjectionType(G)

// Perform getProjectionType(G3) where G3
//  is the 3-dimensional grapg generated by the function
// new3DGraph
G3 = new3DGraph("getProjectionTypeGraph");
n3 = getProjectionType(G3);
// Result:     n3:    0
```

ALGORITHM AND COMMENTS

The output values for n are also available as HiQ Language Constants; they are: <orthographic> = 0 and <projection> = 1.

If graph G is not a faceted 3-dimensional graph, then an error message will be returned.

■ heightShading

FUNCTION

heightShading (G)

PURPOSE

Change a 3-dimensional graph with color as a linear function of the z value

INPUT

G (Graph): the entered 3-dimensional graph

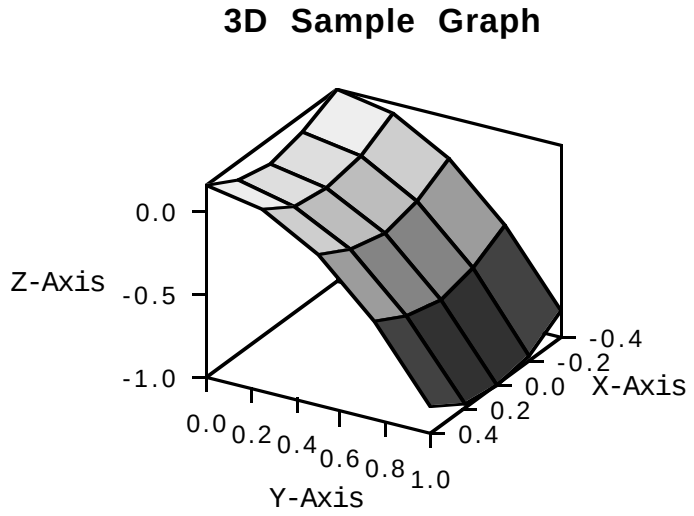
OUTPUT

(The graph G is redrawn with its color as a function of the z value)

EXAMPLES

```
// An example for: heightShading(G)

// Perform heightShading (G) where G3 is the
// 3-dimensional graph containing a plot P3
// generated by new3DGraph and P3 is
// the 3-dimensional plot generated by new3DDataPlot using
// the data sets x,y,z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
```

```
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("heightShadingGraph");
P3 = new3DDataPlot("  ",x,y,z);
addPlot(G3,P3);
heightShading(G3);

// Result: (Display of the resultant G3)
```



SEE ALSO

setGraphShading, hiddenLineGraph, lightSourceShading, wireFrameGraph

ALGORITHM AND COMMENTS

The function setGraphShading(•) can also be used to specify height shading for a graph.

If graph G is not a 3-dimensional graph, then an error message will be returned.

■ hiddenLineGraph

FUNCTION

hiddenLineGraph(G)

PURPOSE

Render a 3-dimensional graph such that it hides all lines not visible to the viewer (hidden line removal)

INPUT

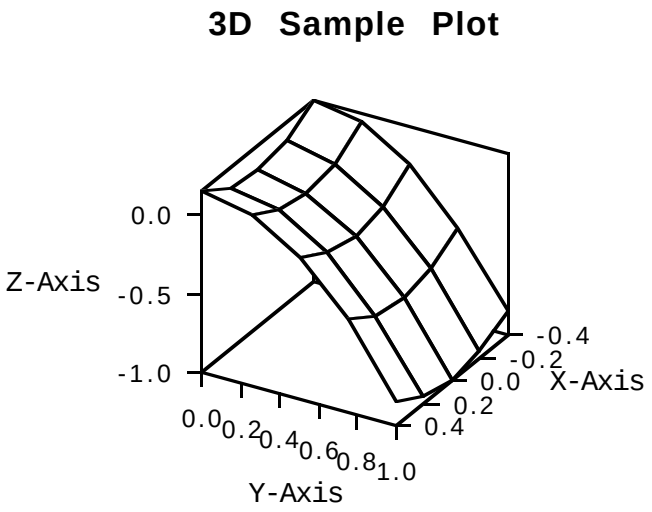
G (Graph): the entered 3-dimensional graph

OUTPUT

(The graph is redrawn to hide all lines not visible to the viewer)

EXAMPLES

```
// Perform hiddenLineGraph(G3) where G3 is the
// 3-dimensional graph containing a plot P3
// generated by new3DGraph and P3 is
// the 3-dimensional plot generated by new3DDataPlot
// using the data sets x, y, z listed below
x = {  0.449; 0.898; 1.346; 1.795; 2.244};
y = { -0.449;-0.898;-1.346;  -1.795; -2.244};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("hiddenLineGraph");
P3 = new3DDataPlot("  ",x,y,z);
addPlot(G3,P3);
hiddenLineGraph(G3);
// Result: (Display of the resultant G3)
```



SEE ALSO

setGraphShading, heightshading, lightSourceShading, wireFrameGraph

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph, then an error message will be returned.

The function setGraphShading(•) can also be used to specify a hiddenline Graph format.

■ lightSourceShading

FUNCTION

lightSourceShading (G)

PURPOSE

Change the shading mode of a three dimensional graph to a light source shading mode

INPUT

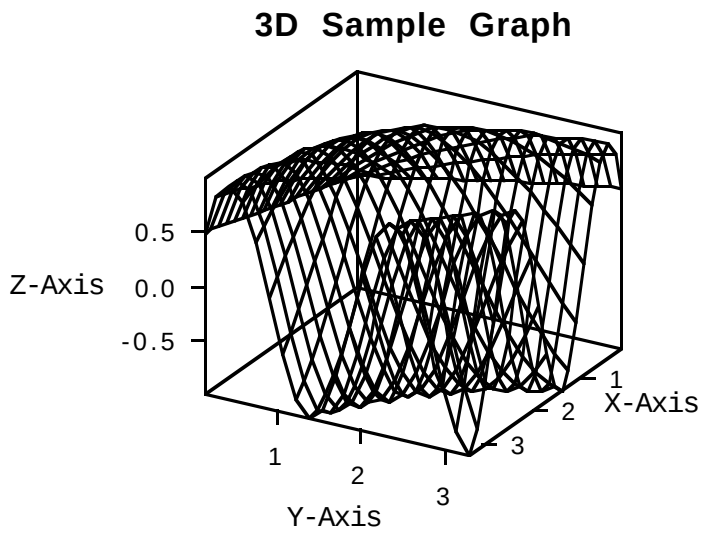
G (Graph): the entered 3-dimensional graph

OUTPUT

(The graph G is redrawn with the new light source shading mode)

EXAMPLES

```
// An example for: lightSourceShading(G)

// Perform lightSourceShading(G3) where G3 is
// the 3-dimensional graph generated by the function
// new3DGraph and containing a surface generated by the
// following formulae

for i = 1 to 21 do
  x[i] = i*<pi>/20;
  y[i] = x[i];
end for;
for i = 1 to 21 do
  for j = 1 to 21 do
    z[i,j] = sin(x[i]*y[j]);
  end for;
end for;
P3 = new3DDataPlot(" ",x,y,z);
G3 = new3DGraph("3D Sample Graph");
lightSourceShading(G3);
// The following statement is for displaying the
// resultant plot P3 in  G3

addPlot(G3,P3);

// Result:   (Display of resultant plot P3 in graph G3)
```



SEE ALSO

heightShading, hiddenLineGraph, lightSourceShading, setGraphShading

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph, then an error message will be returned.

The function setGraphShading(•) can also be used to specify the light source shading for a Graph. Several additional functions affect the appearance of lighted graphs: setLightDirection, setLightState, setLightType, and setLightIntensity.

■ linePlot

FUNCTION

linePlot(P)

PURPOSE

Render a plot with a line but without marking the data points

INPUT

P (Plot): the entered 2- or 3-dimensional plot

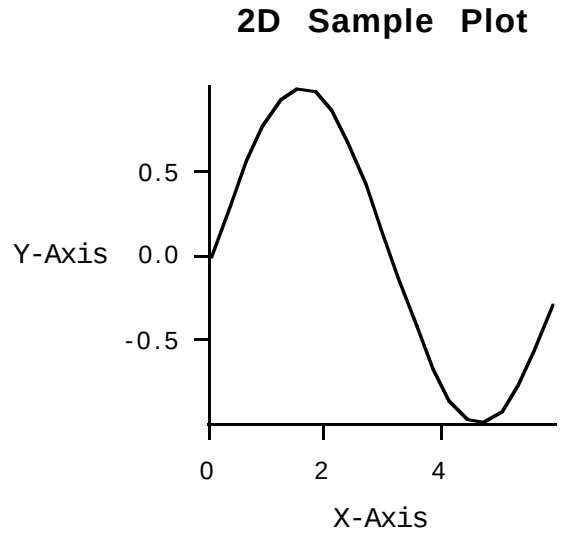
OUTPUT

(The Plot attribute is changed to line Plot. For a 3D Plot, this is equivalent to specifying a surface plot) There is no visible effect until the plot P is displayed in a graph with the function addPlot()

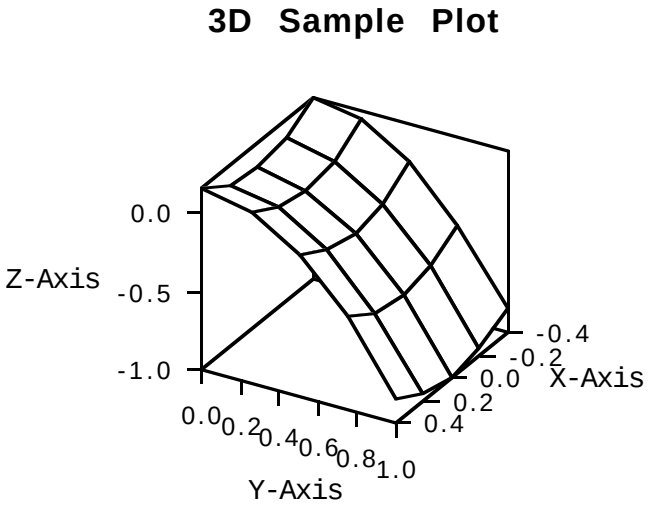
EXAMPLES

```
// Examples for: linePlot(P)

// Perform linePlot(P2) where is the 2-dimensional plot
// generated by new2DDataPlot using the data sets x,y,z
// list below

x = {  0;   0.299; 0.598;  0.898; 1.197; 1.496; 1.795;
     2.094; 2.394; 2.693;  2.992; 3.291; 3.590; 3.890;
     4.189; 4.488; 4.787;  5.086; 5.386; 5.685; 5.984};
y = {  0;   0.295; 0.563;  0.782; 0.931;  0.997; 0.975;
     0.866; 0.680; 0.434;  0.149;-0.149; -0.434;-0.680;
    -0.866;-0.975;-0.997; -0.931;-0.782; -0.563;-0.295};
P2 = new2DDataPlot("  ",x,y);
G2 = new2DGraph("2D Sample Plot");
linePlot(P2);
```

```
// The following statement is for displaying the result
addPlot(G2 , P2);

// Result: (Display of the resultant plot P2 in graph G2)
```




```
// Perform linePlot(P3) where is the 3-dimensional plot
// generated by new3DDataPlot using the data sets x,y,z
// list below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot(" ",x,y,z);
G3 = new3DGraph("3D Sample Plot");
linePlot(P3);

// The following statement is for displaying the result
addPlot(G3, P3);

// Result: (Display of the resultant plot P3 in graphG3)
```



SEE ALSO

pointLinePlot, pointPlot, and surfacePlot

ALGORITHM AND COMMENTS

The equivalent effect can be achieved using the function setPlotDisplayFormat

■ new2DDataPlot

FUNCTION

P = new2DDataPlot(T, X, Y)

PURPOSE

Create a 2-dimensional plot from numerical data

INPUT

T (String): title of the 2-dimensional plot

X (Integer or Real Vector): n-dimensional vector specifying the domain (i.e., x-axis values) of the plot

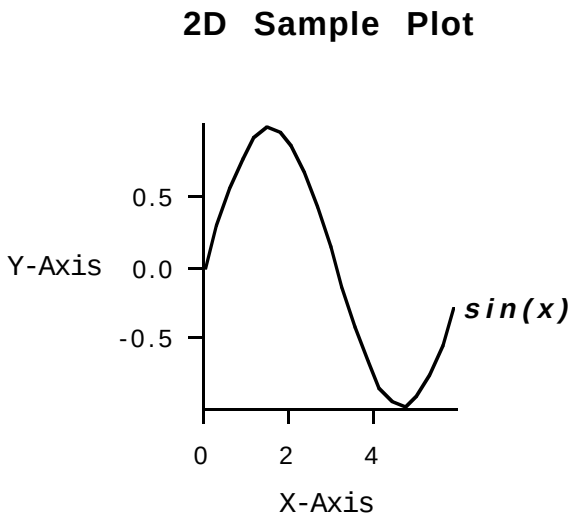
Y (Integer or Real Vector): n-dimensional vector specifying the range (i.e., y-axis values) of the plot

OUTPUT

P (Plot): the plot generated from input vectors X and Y

EXAMPLES

```
// An example for: new2DDataPlot(T,X,Y)
// Perform new2DDataPlot(Title,x,y) where Title is the
// string "2D Sample Plot", and  x,y are the data sets listed below
x = {  0;   0.299; 0.598;  0.898; 1.197; 1.496; 1.795;
     2.094; 2.394; 2.693;  2.992; 3.291; 3.590; 3.890;
     4.189; 4.488; 4.787;  5.086; 5.386; 5.685; 5.984};
y = {  0;   0.295; 0.563;  0.782; 0.931;  0.997; 0.975;
     0.866; 0.680; 0.434;  0.149;-0.149; -0.434;-0.680;
    -0.866;-0.975;-0.997; -0.931;-0.782; -0.563;-0.295};
Title = "2D Sample Plot";
P2 = new2DDataPlot("  ",x,y);
// The following statements are for displaying the result
G2 = new2DGraph("2D Sample Plot");
addPlot(G2, P2);
// Result: (Display of the resultant plot P2 in graph G2)
```



ALGORITHM AND COMMENTS

After creating a 2D Plot, you must add it to a 2D or 3D Graph. Before adding the Plot to a Graph, the following functions can be used to change its format: setPlotDisplayFormat, setPlotLineColor, setPlotLineWidth, setPlotMarkerColor, setPlotMarkerStyle, setPlotTitle, setPlotTitleColor, pointPlot, linePlot, and pointLinePlot.

■ new2DGraph

FUNCTION

G = new2DGraph(T)

PURPOSE

Initialize a new 2-dimensional graph

INPUT

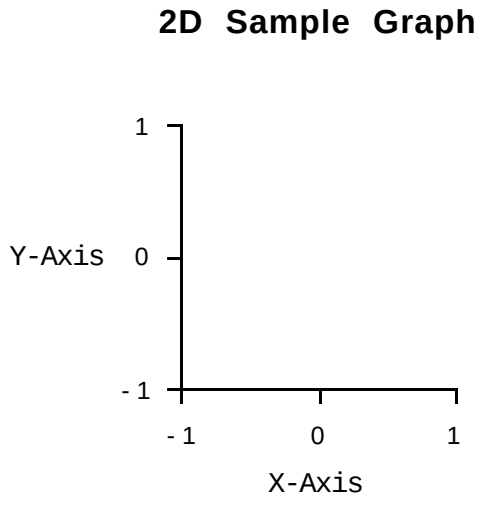
T (String): title of the new 2-dimensional graph

OUTPUT

G (Graph): the resultant new graph containing x-axis, y-axis with default lower limit, upper limit, scaling and the specified title in the window

EXAMPLES

```
// An example for: new2DGraph(T)
// Perform new2DGraph(Title) where Title is the
// string "2D Sample Graph"
Title = "2D Sample Graph";
G2 = new2DGraph(Title);
// Result: (Display of the resultant graph G2)
```



ALGORITHM AND COMMENTS

After creating a 2D Graph, you must use addPlot to display plots in the graph

■ new3DDataPlot

FUNCTION

P = new3DDataPlot(T, X, Y, Z)

PURPOSE

Create a 3-dimensional plot from numerical data

INPUT

T (String): title of the 3-dimensional plot

X (Integer or Real Vector or Matrix): n-dimensional vector specifying the first coordinate of the domain (i.e., x-axis values) of the plot

Y (Integer or Real Vector or Matrix): n-dimensional vector specifying the second coordinate of the domain (i.e., y-axis values) of the plot

Z (Integer or Real Vector or Matrix): n-dimensional vector specifying the range (i.e., z-axis values) of the plot

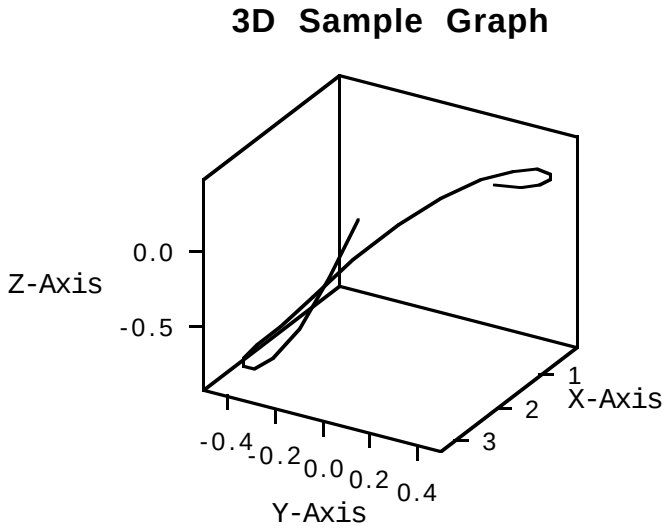
(Note that if X, Y and Z are all vectors, the plot is a 3-dimensional curve. If X and Y are vectors and Z is a matrix, the plot is a surface. If all three are matrices, the plot is a full surface such as a sphere)

OUTPUT

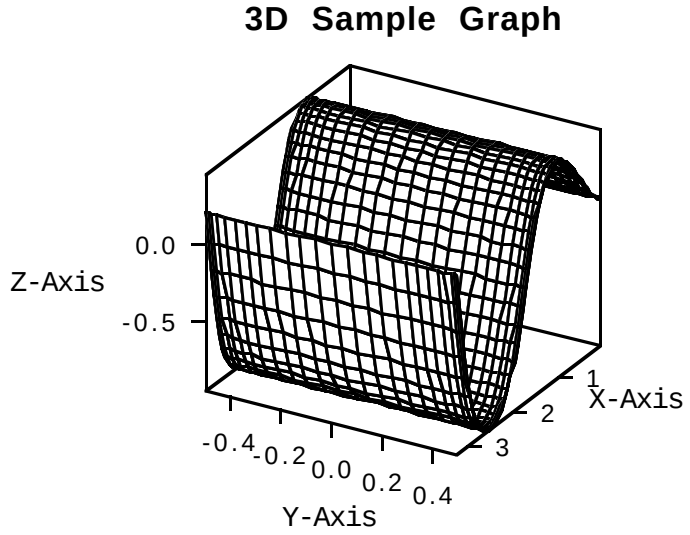
P (Plot): the generated plot from X, Y and Z

EXAMPLES

```
// Examples for: new3DDataPlot(T,X,Y,Z)
// Perform new3DDataPlot(Title31,x,y,z)  for a curve plot where
// Title31 is the string "Curve", and  x,y and z are the data
// set generated by the following formulae

for i = 1 to 21 do
  x[i] = i*<pi>/20;
  y[i] = sin(x[i])*cos(x[i]);
  z[i] = sin(x[i]*y[i]);
end for;
Title31 = "Curve";
P31 = new3DDataPlot(Title31,x,y,z);
// The following statements are for displaying the result
G3 = new3DGraph("3D Sample Graph");
addPlot(G3, P31);

// Result: (Display of the resultant plot P31 in graph G3)
```

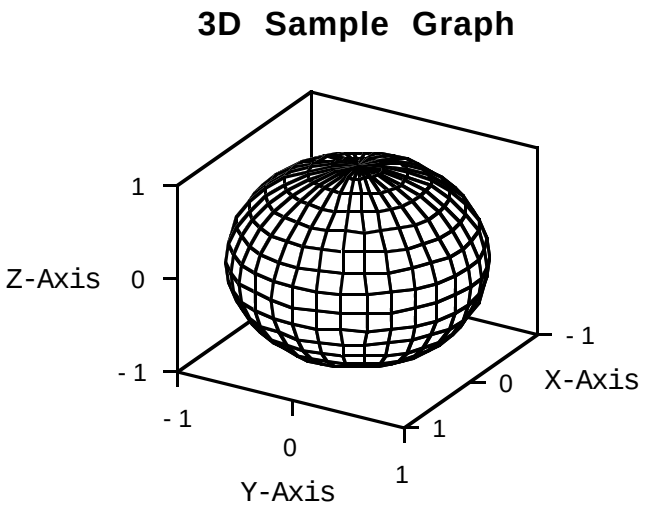



```
// Perform new3DDataPlot(Title32,x,y,z) for a surface
// plot where Title32 is the string "Surface", and  x,y  and z
// are the data set generated by the following formulae

n = 51;
for i = 1 to n do
  x[i] = i*<pi>/(n-1);
  y[i] = sin(x[i])*cos(x[i]);
end for;

for i = 1 to n do
  for j = 1 to n do
    z[i,j] = sin(x[i]*y[i]);
  end for;
end for;

Title32 = "Surface";
P32 = new3DDataPlot(Title32,x,y,z);

// The following statements are for displaying the result

G3 = new3DGraph("3D Sample Graph");
addPlot(G3, P32);

// Result: (Display of the resultant plot P32 in graph G3)
```




```
// Perform new3DDataPlot(Title33,as,bs,cs) for a full
// surface plot where Title33 is the string "Full Surface",
// and  as, bs, and cs are the data set generated by the
// following formulae

pt1 = 16;
pt2 = 32;

for i = 0 to pt1 do
   u = <pi> * i / pt1;
   for j = 0 to pt2 do
        v = 2 * <pi> * j / pt2;
        as[i+1,j+1] = sin(u) * cos(v);
         bs[i+1,j+1] = sin(u) * sin(v);
         cs[i+1,j+1] = cos(u);
   end for;
end for;

Title33 = "Full Surface";
P33 = new3DDataPlot(Title33,as,bs,cs);

// The following statements are for displaying the result

G3 = new3DGraph("3D Sample Graph");
addPlot(G3, P33);
```

```
// Result: (Display of the resultant plot P33 in graph G3)
```



ALGORITHM AND COMMENTS

After creating a 3D Data Plot, you must use the function addPlot(•) to add it to a 3D Graph. Before adding it, you can use the following functions to change the Plot's attributes:
setPlotDisplayFormat, setPlotLineColor, setPlotLineWidth, setPlotMarkerColor, setPlotMarkerStyle, setPlotTitle, setPlotTitleColor, pointPlot, surfacePlot.

■ new3DGraph

FUNCTION

G = new3DGraph(T)

PURPOSE

Initialize a new 3-dimensional graph

INPUT

T (String): title of the new 3-dimensional graph

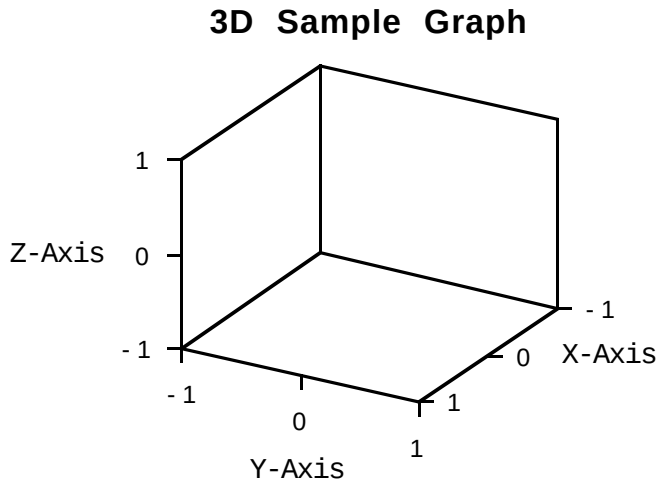
OUTPUT

G (Graph): the resultant new graph, containing x-axis, y-axis, z-axis with default lower limit, upper limit,

scaling and the specified title in the Graph Editor window

EXAMPLE

```
// An example for: new3DGraph(T)

// Perform new3DGraph(Title) where Title is the
// string "3D Sample Graph"

Title = "3D Sample Graph";
G3 = new3DGraph(Title);

// Result: (Display of the resultant graph G3)
```



■ pointLinePlot

FUNCTION

pointLinePlot(P)

PURPOSE

Render a plot with both data points and connecting lines

INPUT

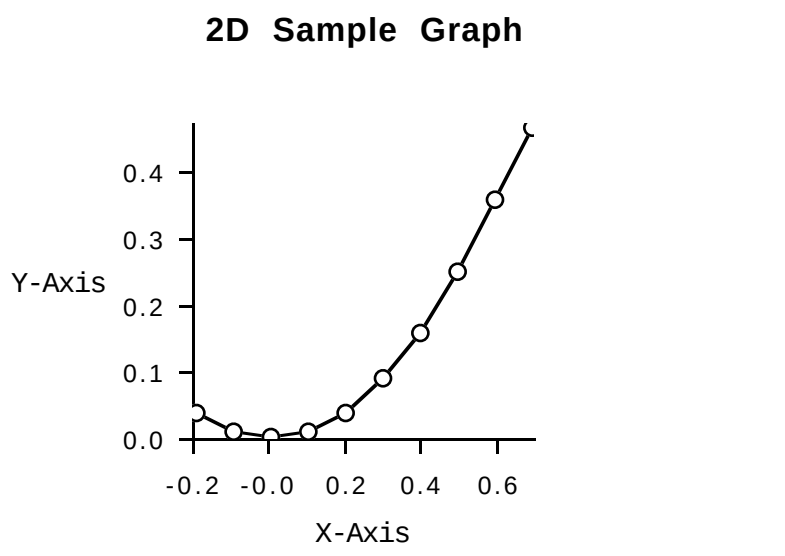
P (Plot): a 2-dimensional plot

OUTPUT

(The attributes of a plot are changed. There is no visible effect until the plot P is displayed on a graph with addPlot)

EXAMPLES

```
// Examples for: pointLinePlot(P)

// Perform pointLinePlot(P2) where P2 is
// the 2-dimensionalplot generated by the function
// new2DDataPlot using the data sets x,y listed below

x = { -0.2; -0.1; 0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7};
y = {0.04; 0.01; 0; 0.01; 0.04; 0.09;
     0.16; 0.25; 0.36; 0.47};
P2 = new2DDataPlot("  ",x,y);
pointLinePlot(P2);
//The following statements are for displaying the result

G2 = new2DGraph("pointLinePlotGraph");
addPlot(G2,P2);

// Result: (Display of the resultant plot P2 in graph G2)
```



SEE ALSO

linePlot, pointPlot

ALGORITHM AND COMMENTS

The equivalent effect can be achieved by using the function setPlotDisplayFormat

■ pointPlot

FUNCTION

pointPlot(P)

PURPOSE

Render a plot with data points displayed only

INPUT

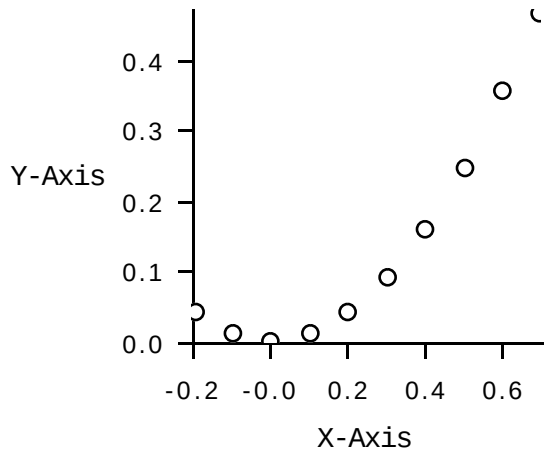
P (Plot): a 2- or 3-dimensional plot

OUTPUT

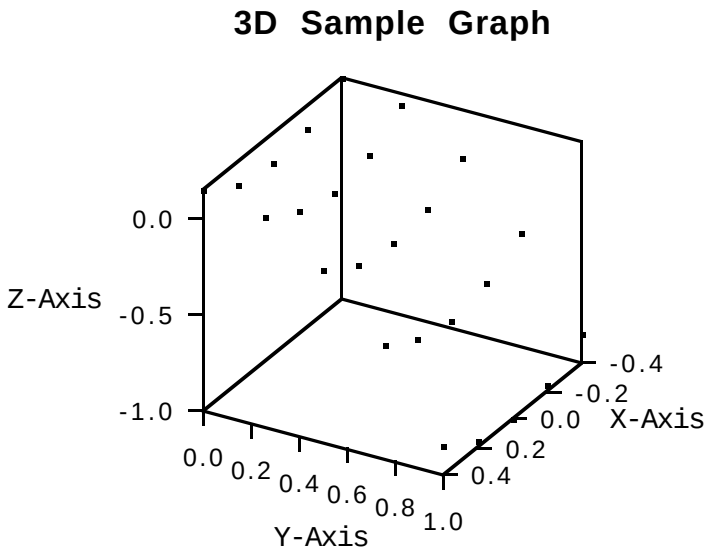
(An attribute of the plot is changed. There is no visible effect until that plot P is displayed on a graph with addPlot)

EXAMPLES

```
// Examples for: pointPlot(P)

// Perform pointPlot(P2) where P2 is
// the 2-dimensional plot generated by the function
// new2DDataPlot using the data sets x,y listed below

x = { -0.2; -0.1; 0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7};
y = {0.04; 0.01; 0; 0.01; 0.04; 0.09;
     0.16; 0.25; 0.36; 0.47};
P2 = new2DDataPlot("  ",x,y);
pointPlot(P2);

//The following statements are for displaying the result

G2 = new2DGraph("2D Sample Graph");
addPlot(G2,P2);

// Result: (Display of the resultant plot P2 in graph G2)
```




```
// Perform pointPlot(P3) where P3 is
// the 3-dimensional plot generated by the function
// new3DDataPlot using the data sets x,y,z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("  ",x,y,z);
pointPlot(P3);

// The following statements are for displaying the result

G3 = new3DGraph("3D Sample Graph");
addPlot(G3,P3);

// Result: (Display of the resultant plot P3 in graph G3)
```



SEE ALSO

linePlot, pointLinePlot, surfacePlot

ALGORITHM AND COMMENTS

The equivalent effect can be achieved by using the function setPlotDisplayFormat

■ removePlot

FUNCTION

removePlot (G, m)

PURPOSE

Remove a plot from a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

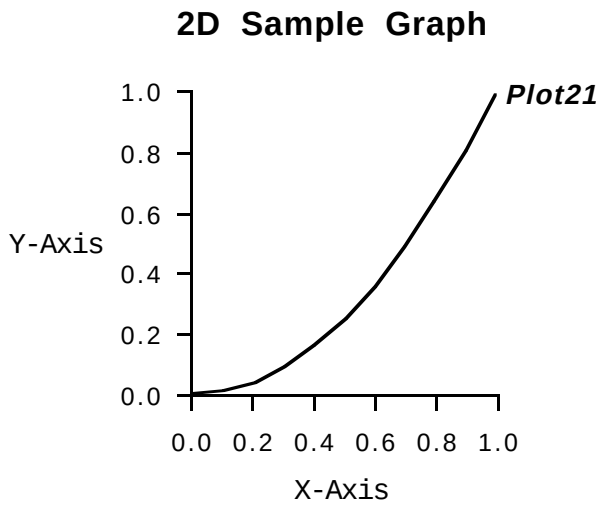
m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

OUTPUT

(The m-th plot in graph G is removed)

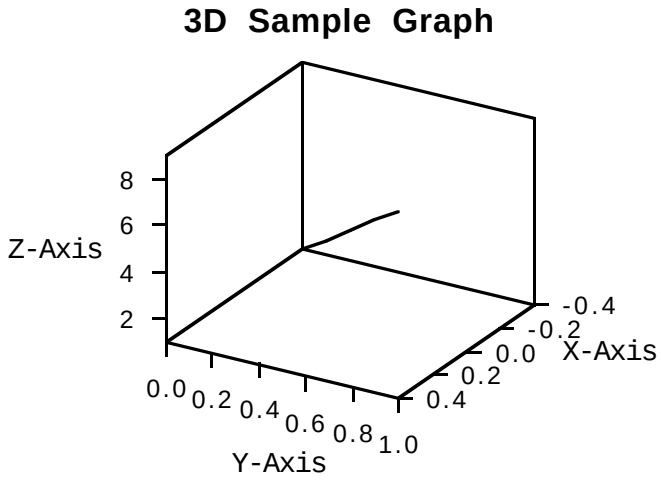
EXAMPLES

```
// Examples for: removePlot(G,m)

// Perform removePlot(G2,m2) for m2 = 1
// where G2 isthe 2-dimensional graph containing 2
// plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets X,Y listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("2D Sample Graph ");
P21 = new2DDataPlot("Plot21 ",x,y);
P22 = new2DDataPlot("Plot22 ",y,x);
addPlot(G2, P21);
addPlot(G2, P22);
m2 = 1;
removePlot(G2,m2);

// Result: (Display of the resultant G2)
```




```
// Perform removePlot(G3,m3) for m3 =0
// where G3 isthe 3-dimensional graph containing 2
// plots generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets X, Y, Z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
```

```
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
w = {1; 3; 5; 7; 9};
G3 = new3DGraph("3D Sample Graph ");
P31 = new3DDataPlot(" ",x,y,z);
P32 = new3DDataPlot(" ",x,y,w);
addPlot(G3, P31);
addPlot(G3, P32);
m3 = 0;
removePlot(G3,m3);

// Result:
// (Display of the resultant G3. Note that surface plot
// corresponding to the first plot is not seen in the graph)
```



ALGORITHM AND COMMENTS

Plots are numbered *from zero* in the order that they are added to a graph. If a plot is added to an empty graph, it will be plot 0. Subsequent plots that are added will be numbered plot 1, plot 2, and so on. Removing a plot causes the remaining plots to be renumbered from zero but maintains the same ordering. Therefor, repeatedly removing plot 0 will remove all plots from a graph.

■ rotateCamera

FUNCTION

rotateCamera (G, theta, phi)

PURPOSE

Change the direction from which a 3-dimensional graph is viewed

INPUT

G (Graph): a 3-dimensional graph

theta (Real Scalar): the azimuth of the camera in degrees where $0 \leq theta \leq 360$

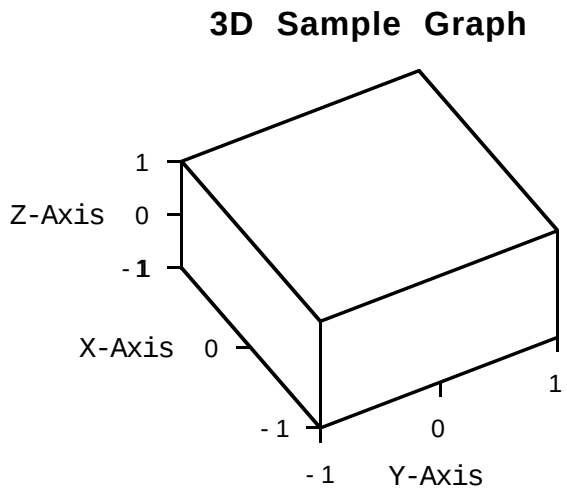
phi (Real Scalar): the ascension of the camera in degrees where $-90 \leq phi \leq 90$

OUTPUT

(The graph G is redrawn to the new azimuth and ascension)

EXAMPLE

```
// An example for: rotateCamera(G,theta,phi)
// Perform rotateCamera(G3,theta,phi) where
// G3 is the 3-dimensional graph generated by the
// function new3DGraph and theta = 30 and phi = -60
G3 = new3DGraph("rotateCameraGraph");
theta = 30;
phi = -60;
rotateCamera(G3, theta, phi);
// Result: (Display of the resultant graph G3)
```



ALGORITHM AND COMMENTS

The default azimuth for a newly-created graph is 30 degrees. The default ascension is 45 degrees. The functions viewFromFront(), viewFromSide(), and viewfromTop() also change the direction from which a 3D graph is viewed.

If graph G is not a 3-dimensional graph, then an error message will be returned.

■ set2DClipRange

FUNCTION

set2DClipRange (G, xmin, xmax, ymin, ymax)

PURPOSE

Change the axis ranges of a 2-dimensional graph

INPUT

G (Graph): a 2-dimensional graph

xmin, xmax (Real Scalar): the lower and the upper limits of the x-axis

ymin, ymax (Real Scalar): the lower and the upper limits of the y-axis

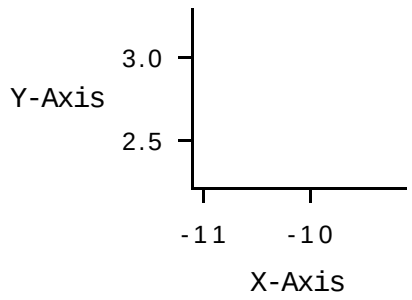
OUTPUT

(The graph G is redrawn to the new axis ranges)

EXAMPLES

```
// Perform set2DClipRange(G2,xmin,xmax,ymin,ymax)
// where G2 is the 2-dimensional graph generated
// by the function new2DGraph  and xmin = -11.1, xmax = -9.1,
// ymin = 2.2, ymax= 3.3

G2 = new2DGraph("set2DClipRangeGraph");
xmin = -11.1;
xmax = -9.1;
ymin = 2.2;
ymax= 3.3;
setAutoClipping(G2,0);
set2DClipRange(G2,xmin,xmax,ymin,ymax);
```

```
// Result:  (Display of the resultant graph G2)
```



SEE ALSO

setAutoClip, setAxisScale, setAxisLimits

ALGORITHM AND COMMENTS

An error message will be returned by the function, if one of the following cases is encountered:

1) Graph G is not a 2-dimensional graph,

2) auto clip state is current on,

3) xmin > xmax or ymin > ymax,

4) scaling mode of any axis is logarithmic and its minimum is non-positive.

■ set3DClipRange

FUNCTION

set3DClipRange (G, xmin, xmax, ymin, ymax,zmin,zmax)

PURPOSE

Change the axis ranges of a 3-dimensional graph

INPUT

G (Graph): a 3-dimensional graph

xmin, xmax (Real Scalar): the lower and the upper limits of the x-axis

ymin, ymax (Real Scalar): the lower and the upper limits of the y-axis

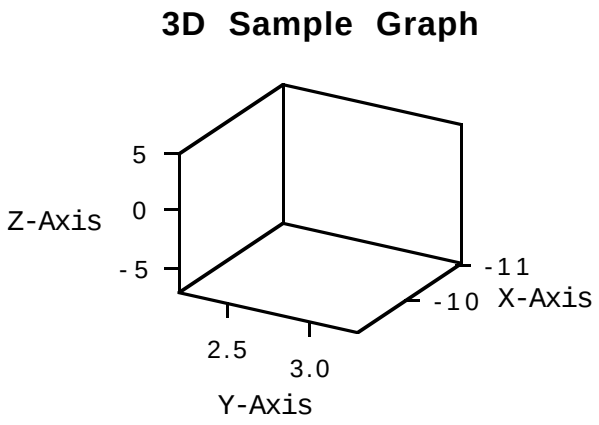
zmin, zmax (Real Scalar): the lower and the upper limits of the z-axis

OUTPUT

(The graph is redrawn to the new axis ranges)

EXAMPLES

```
// An example for: set3DClipRange(G,xmin,xmax,ymin,ymax,zmin,zmax)

// Perform set3DClipRange(G3,xmin,xmax,ymin,ymax,
// zmin,zmax) where G3 is the 3-dimensional graph
//  generated by the function new3DGraph and xmin = -11.1,
// xmax = -9.1, ymin = 2.2, ymax= 3.3, zmin = -7, zmax = 5

G3 = new3DGraph("set3DClipRangeGraph");
xmin = -11.1;
xmax = -9.1;
ymin = 2.2;
ymax= 3.3;
zmin = -7;
zmax= 5;
setAutoClipping(G3,0);
set3DClipRange(G3,xmin,xmax,ymin,ymax,zmin,zmax);

// Result:  (Display of the resultant graph G3)
```



SEE ALSO

setAutoClip, setAxisScale,SetAxisLimits

ALGORITHM AND COMMENTS

An error message will be returned by the function, if one of the following cases is encountered

1) Graph G is not a 3-dimensional graph,

2) auto clip state is current on,

3) xmin > xmax or ymin > ymax or zmin > zmax,

4) scaling mode of any axis is logarithmic and its minimum is non-positive.

■ setAutoClipping

FUNCTION

setAutoClipping (G, clipstate)

PURPOSE

Change the clipping state of a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

clipstate (Integer Scalar): the clipping state where clipstate = 0 for fixing the ranges of axes, i.e., all the plot added to the graph will be clipped to the fixed graph limits; and clipstate = 1 for making the graph resize automatically whenever a new plot is added to the graph such that all the plots are fitted inside the graph limits

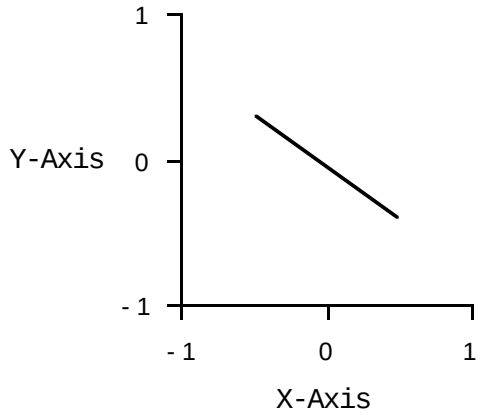
OUTPUT

(The graph G is redrawn to the new clipping state)

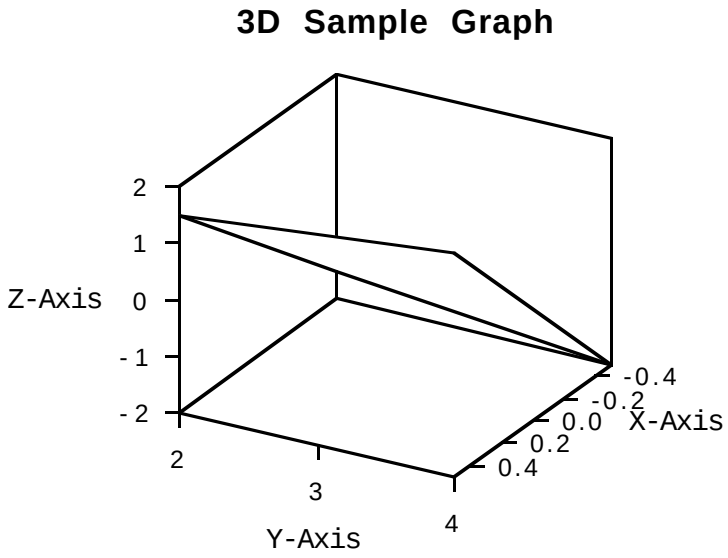
EXAMPLES

```
// Examples for: setAutoClipping(G,clipstate)

// Perform setAutoClipping(G2,EXclip2) where
// G2 is the 2-dimensional graph generated by the
// function new2DGraph and EXclip2 = 0

G2 = new2DGraph("2D Sample Graph");
EXclip2 = 0;
setAutoClipping(G2,EXclip2);

// The following statements are for displaying the
// effect of using setAutoClipping

x = {-0.5; 0.5};
y = { 0.3; -0.4};
P2 = new2DDataPlot(" ",x,y);
addplot(G2,P2);
// Result: (Display of the resultant effect in graph G2)
```

2D Sample Graph




```
// Perform setAutoClipping(G3,EXclip3) where
// G3 is the 3-dimensional graph generated by the
// function new2DGraph and EXclip3 = 1

G3 = new3DGraph("3D Sample Graph");
EXclip3 = 1;
setAutoClipping(G3,EXclip3);

// The following statements are for displaying the
// effect of using setAutoClipping

x = {-0.5; 0.5};
y = { 2; 4};
z = {-1.5, -2; 1.5, 2};
P3 = new3DDataPlot(" ",x,y,z);
addplot(G3,P3);

// Result: (Display of the resultant effect in graph G3)
```



SEE ALSO

setAxisScale, setAxisLimits, set2DClipRange, set3DClipRange

■ setAutoScale

FUNCTION

setAutoScale (G, Iscale)

PURPOSE

Change the scaling state of a 3-dimensional graph

INPUT

G (Graph): a 3-dimensional graph

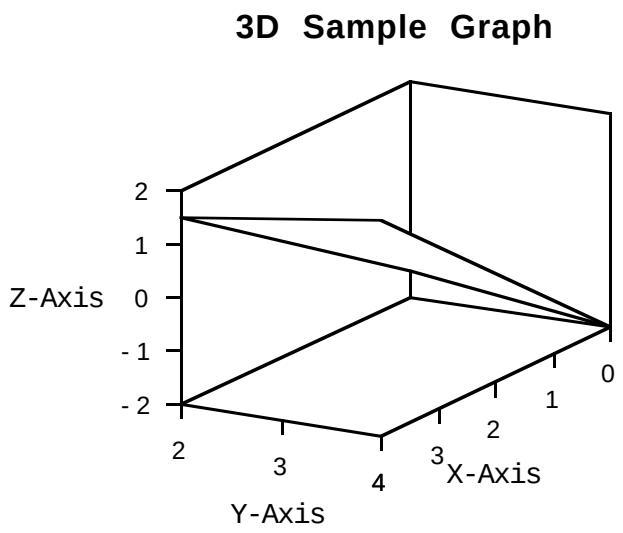
Iscale (Integer Scalar): the scaling state where Iscale = 0 for turning off the auto-scale state, i.e., all the three axes are drawn proportionally (to the same scale) to each other; and Iscale = 1 for turning on the auto-scale state, i.e., each axis is scaled differently so that the bounding box of the graph appears to be a cube

OUTPUT

(The graph is redrawn to the new scaling state)

EXAMPLES

```
// Examples for: setAutoScale(G,Iscale)

// Perform setAutoScale(G3,scale3) where
// G3 is the 3-dimensional graph generated by the
// function new3DGraph and clip3 = 0:

G3 = new3DGraph("setAutoScaleGraph");
scale3 = 0;
setAutoScale(G3,scale3);

// The following statements are for displaying the
// effect of using setAutoScale

x = {0; 4};
y = { 2; 4};
z = {-1.5, -2; 1.5, 2};
P3 = new3DDataPlot(" ",x,y,z);
addplot(G3,P3);

// Result: (Display of the resultant effect in graph G3)
```



ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph, then an error message will be returned.

■ setAxisFlag

FUNCTION

flag = setAxisFlag (G, Iaxis, attribute, enabled)

PURPOSE

Change the attribute of an axis of the graph

INPUT

G (Graph): a 2- or 3-dimensional graph

Iaxis (Integer Scalar): the index of the axis whose limits are queried where
Iaxis = 0 for x-axis, Iaxis = 1 for y-axis and Iaxis = 2 for z-axis (See the Algorithm and Comments section)

attribute (Integer Scalar): the attribute of Iaxis of graph G to be changed

enabled (Integer Scalar): the (boolean) flag to disable or enable the selected attribute (See the Algorithm and Comments section)

OUTPUT

(The graph is redrawn with the new attributes for the selected axis)

EXAMPLES

```
// Examples for: setAxisFlag(G,Iaxis,attribute,enabled)

// Perform setAxisFlag(G2,Iaxis2,attribute2,enabled2)
// where G2 is the 2-dimensional graph generated by
// the function new2DGraph, and Iaxis2 = 1, attribute2 = 2,
// enabled2 = 0

G2 = new2DGraph("setAxisFlagGraph");
Iaxis2 = 1;
attribute2 = 2;
enabled2 = 0;
setAxisFlag(G2,Iaxis2,attribute2,enabled2);

//     The following statement is for displaying the result

flag2 = getAxisFlag(G2,Iaxis2,attrib2);
// Result:
// flag2:  0

// Perform setAxisFlag(G3,Iaxis3,attribute3,enabled3)
// where G3 is the 3-dimensional graph generated by
// the function new3DGraph, and Iaxis3 = 1, attribute3 = 2,
// enabled3 = 0

G3 = new3DGraph("getAxisFlag3DGraph");
```

```
Iaxis3 = 2;
attribute3 = 2;
enabled3 = 0;
setAxisFlag(G3,Iaxis3,attrib3,enabled3);

//     The following statement is for displaying the result

flag3 = getAxisFlag(G3,Iaxis3,attrib3);

// Result:
// flag3:    0
```

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ Language Constants; they are: <x_axis>, <y_axis>, and <z_axis>.

The setAxisFlag() attribute options are given in this section of the function getAxisFlag().

■ setAxisLimits

FUNCTION

setAxisLimits (G, Iaxis, lower, upper)

PURPOSE

Change the limits of an axis of a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

Iaxis (Integer Scalar): the index of the axis whose limits are queried where Iaxis = 0 for x-axis, Iaxis = 1 for y-axis and Iaxis = 2 for z-axis (See the Algorithm and Comments section)

lower, upper (Real Scalar): the lower limit and the upper limit of the axis

OUTPUT

(The graph is redrawn to the new limits in desired the axis)

EXAMPLES

```
// Examples for: setAxisLimits(G,Iaxis,lower,upper)

// Perform setAxisLimits (G2,Iaxis2,l2,u2) where
// G2 is the 2-dimensional graph generated by the
// function new2DGraph,Iaxis2 = 0, l2=-3.14, u2= 4.15

G2 = new2DGraph("setAxisLimitsGraph");
Iaxis2 = 0;
```

```
l2 = -3;
u2 = 9;
setAutoClipping(G2,0);
setAxisLimits(G2,Iaxis2,l2,u2);

// The following statement is for displaying the results

[lower2,upper2] = getAxisLimits(G2,Iaxis2);

// Results:
//     lower2:     -3
//     upper2:      9

// Perform setAxisLimits(G3,Iaxis3, l3, u3) where
// G3is the 3-dimensional graph generated by the
// function  new3DGraph, and Iaxis3 = 2, l3 =-4, u3 = 5

G3 = new3DGraph("setAxisLimits3DGraph");
Iaxis3 = 2;
l3 = -4;
u3 = 5;
setAutoClipping(G3,0);
setAxisLimits(G3, Iaxis3, l3, u3);

// The following statement is for displaying the results

[lower3,upper3] = getAxisLimits(G3,Iaxis3);

// Results:
//     lower3:   -4
//     upper3:    5
```

SEE ALSO

setAutoClip, setAxisScale

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ Language Constants; they are: <x_axis>, <y_axis>, and <z_axis>.

An error message will be returned by the function, if one of the following cases is encountered

1) Graph G is a 2-dimensional graph and Iaxis = 2,

2) auto clip state is current on,

3) lower > upper,

4) scaling mode of the current axis is logarithmic and its lower limit is non-positive.

■ setAxisMinorTicks

FUNCTION

setAxisMinorTicks (G, Iaxis, n)

PURPOSE

Change the number of minor ticks (between the major ticks) for a linearly scaled axis of a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

Iaxis (Integer Scalar): the linearly scaled axis where Iaxis = 0 for x-axis, Iaxis = 1 for y-axis and Iaxis = 2 for z-axis

n (Integer Scalar): the number of minor ticks between the major ticks

OUTPUT

(The graph G is redrawn to the given number of minor ticks)

EXAMPLES

```
// Examples for: setAxisMinorTicks(G,Iaxis,n)

// Perform setAxisMinorTicks(G2,Iaxis2,n2) where
// G2 is the 2-dimensional graph generated by the
// function new2DGraph,Iaxis2 = 0 and n2 = 4

G2 = new2DGraph("setAxisMinorTicksGraph");
setGraphFlag(G2,<hidden_grids>,<false>);
Iaxis2 = 0;
n2 = 4;
setAxisMinorTicks(G2,Iaxis2,n2);

// The following statement is for displaying the
// resultant minor ticks in G2

n2 = getAxisMinorTicks(G2,Iaxis2);

// Result:   n2:      4

// Perform setAxisMinorTicks(G3,Iaxis3,n3) where
// G3 is the 3-dimensional graph generated by the
// function new3DGraph,Iaxis3 = 2 and n3 = 4

G3 = new3DGraph("  ");
setGraphFlag(G3,<hidden_grids>,<false>);
Iaxis3 = 2;
n3 = 12;
setAxisMinorTicks(G3,Iaxis3,n3);

// The following statement is for displaying the
```

```
// resultant minor ticks in G3

n3 = getAxisMinorTicks(G3,Iaxis3);

// Result:
// n3:     12
```

SEE ALSO

setGraphFlag

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ Language Constants; they are: <x_axis>, <y_axis>, and <z_axis>.

If graph G is a 2-dimensional graph and Iaxis = 2, then an error message will be returned.

The graph must have grid flag on to see the effect of minor ticks.

■ setAxisScale

FUNCTION

setAxisScale (G, Iaxis, n)

PURPOSE

Change the scaling mode of an axis of a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

Iaxis (Integer Scalar): the axis whose scaling mode is set where Iaxis = 0 for x-axis, Iaxis = 1 for y-axis and Iaxis = 2 for z-axis

n (Integer Scalar): the scaling mode of the axis where n = 0 for linear scaling, and n = 1 for logarithmic scaling

OUTPUT

(The graph is redrawn to the new scaling mode)

EXAMPLES

```
// Examples for: setAxisScale(G,Iaxis,n)

// Perform setAxisScale(G2,Iaxis2,n2) where
// G2 is the 2-dimensional graph generated by the
// function new2DGraph,Iaxis2 = 0 and n2 = 1

G2 = new2DGraph("setAxisScaleGraph");
```

```
Iaxis2 = 0;
n2 = 1;
setAxisScale(G2,Iaxis2,n2);

// The following statement is for displaying the
// resultant scaling mode in G2

n2 = getAxisScale(G2,Iaxis2);

// Result:
// n2:      1

// Perform setAxisScale(G3,Iaxis3,n3) where
// G3 is the 3-dimensional graph generated by the
// function new3DGraph,Iaxis3 = 2 and n3 = 0

G3 = new3DGraph("setAxisScale3DGraph");
Iaxis3 = 2;
n3 = 0;
setAxisScale(G3,Iaxis3,n3);

// The following statement is for displaying the
// resultant scaling mode in G3

n3 = getAxisScale(G3,Iaxis3);

// Result:
// n3:     0
```

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ Language Constants; they are: <x_axis>, <y_axis>, and <z_axis>. The scaling mode parameters are also available; they are: <linear_scale> and <log_scale>.

An error message will be returned by the function, if one of the following cases is encountered

1) Graph G is a 2-dimensional graph and Iaxis = 2,

2) auto clip state is currently off,

3) scaling mode of the current axis is logarithmic and its lower limit is non-positive.

■ setAxisTitle

FUNCTION

setAxisTitle (G, Iaxis, T)

PURPOSE

Change the title of an axis of a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

Iaxis (Integer Scalar): the axis where Iaxis = 0 for x-axis, Iaxis = 1 for y-axis and Iaxis = 2 for z-axis

T (String): the title of the axis to be changed

OUTPUT

(The graph is redrawn with the new axis title)

EXAMPLES

```
// Examples for: setAxisTitle(G,Iaxis,T)

// Perform setAxisTitle(G2,Iaxis2,T2) where
// G2 is the 2-dimensional graph generated by the
// function new2DGraph,Iaxis2 = 0,n2 = 1 and T2 is the
// string "New X-Axis"

G2 = new2DGraph("setAxisTitleGraph");
Iaxis2 = 0;
T2 = "New X-Axis";
setAxisTitle(G2,Iaxis2,T2);

// The following statement is for displaying the
// title of x-axis in resultant G2

title2 = getAxisTitle(G2,Iaxis2);

// Result:
// title2:      New X-Axis

// Perform setAxisTitle(G3,Iaxis3,T3) where
// G3 is the 3-dimensional graph generated by the
// function new3DGraph,Iaxis3 = 2 and T3 is the string
// "New Z-Axis"

G3 = new3DGraph("setAxisTitle3DGraph");
Iaxis3 = 2;
T3 = "New Z-Axis";
setAxisTitle(G3,Iaxis3,T3);

// The following statement is for displaying the
// title of x-axis in resultant G2

title3 = getAxisTitle(G3,Iaxis3);

/  Result:
```

```
// title3:      New Z-Axis
```

ALGORITHM AND COMMENTS

The axis index parameters are also available as HiQ Language Constants; they are: <x_axis>, <y_axis>, and <z_axis>.

If graph G is a 2-dimensional graph and Iaxis = 2, then an error message will be returned.

■ setGraphFlag

FUNCTION

setGraphFlag (G, attribute, state)

PURPOSE

Change the attribute of a graph.

INPUT

G (Graph): a 2- or 3-dimensional graph

attribute (Integer Scalar): the attribute to be queried where attribute = 1 for hiding the graph title, attribute = 2 for hiding axes and their annotation, attribute = 4 for hiding grids, and attribute = 8 for hiding axes annotation (title, labels and ticks)

state (Integer Scalar): new value of the attribute where state = 0 for turning on the attribute, state = 1 for turning off the attribute

OUTPUT

(The graph is redrawn with the new attribute)

EXAMPLES

```
// Examples for: setGraphFlag(G,attribute,state)

// Perform setGraphFlag(G2,attribute2,state2) for
// attribute2 = 1, state2 = 0 where G2 is the
// 2-dimensional graph generated by the function new2Graph

G2 = new2DGraph("setGraphFlagGraph");
attribute2 = 1;
state2 = 0;
setGraphFlag(G2,attribute2,state2);

// The following stement is for displaying the result

flag2 = getGraphFlag (G2, attribute2);
```

```
// Result:
//    flag2:0

// Perform setGraphFlag(G3,attribute3,state3) for
// attribute3 = 2, state3 = 1 where G3 is the
// 3-dimensional graph generated by the function new3Graph

G3 = new3DGraph("setGraphFlag3DGraph");
attribute3 = 2;
state3 = 1;
setGraphFlag(G3,attribute3,state3);
// The following stement is for displaying the result
flag3 = getGraphFlag (G3, attribute3);

// Result:
//        flag3 : 1
```

ALGORITHM AND COMMENTS

The attribute parameters are also available as HiQ Language Constants; they are: <hidden_title> = 1; <hidden_axes> = 2; <hidden_grids> = 4; and <hidden_annotation> = 8.

The default attributes for each axis (when a graph is created) are: hiding graph title: 0<false> hiding axes: 0<false> hiding grids: 1<true> hiding axis annotation: 0<false>

■ setGraphPlotBackfaceMode

FUNCTION

setGraphPlotBackfaceMode(G, m, n)

PURPOSE

Change the backface mode of a 3-dimensional plot in a graph

INPUT

G (Graph): a 3-dimensional plot

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

n (Integer Scalar): the backface mode of the m-th plot in graph G where n = 0 for backfaces to be removed and n = 1 for backfaces to be drawn

OUTPUT

(The m-th plot in graph G is changed to the new backface mode)

EXAMPLES

```
// An example for: setGraphPlotBackfaceMode(G,m,n)
```

```
// Perform setGraphPlotBackfaceMode(G3,m3,n3) for
// m3 =0,  n3 = 1 where G3 isthe 3-dimensional graph
// containing one plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotBackfaceModeGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = 1;
setGraphPlotBackfaceMode(G3,m3, n3);

// The following statement is for displaying the resultant
// back face mode in the resultant graph G3

Mode3 = getGraphPlotBackfaceMode(G3,m3);

// Result:
// Mode3:  1
```

ALGORITHM AND COMMENTS

If the m-th plot in graph G is not a faceted 3-dimensional plot, an error message will be returned.

■ setGraphPlotContourPlane

FUNCTION

setGraphPlotContourPlane (G, m, n)

PURPOSE

Change the coordinate plane along which contours of a 3-dimensional plot in a graph are to be made

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

n (Integer Scalar): the coordinate plane to make contours of the m-th plot in graph G where n = 3 for the xy-plane, n = 1 for the yz-plane and n = 2 for the zx-plane

OUTPUT

(The m-th plot in graph G is changed to the new coordinate plane)

EXAMPLE

```
// An example for: setGraphPlotContourPlane(G,m,n)

// Perform setGraphPlotContourPlane(G3,m3,n3) for
// m3 =0, n3 = 2 where G3 is the 3-dimensional graph
// containing a single plot generated by the functions
// new3DGraph,new3DDataPlot  and addPlot using the data sets
// X, Y, Z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotContourPlaneGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = 2;
setGraphPlotContourPlane (G3,m3,n3);

// The following statement is for displaying the
// resultant coordinate plane  in G3

Plane3 = getGraphPlotContourPlane (G3,m3);

// Result:
// Plane3:  2
```

ALGORITHM AND COMMENTS

The coordinate plane parameter n is also available as a HiQ Language Constant; the values are: <xy_plane>, <yz_plane>, and <xz_plane>.

This function is used along with setGraphPlotProjectedContour(). To project the contour onto a plane, you must call setGraphPlotProjectedContour().

If the m-th plot in graph G is not a faceted 3-dimensional plot, an error message will be returned.

■ setGraphPlotCoordSystem

FUNCTION

setGraphPlotCoordSystem (G, m, n)

PURPOSE

Change the coordinate system of a plot in a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

n (Integer Scalar): the coordinate system of the m-th plot in graph G where n = 0 for Cartesian coordinate system, n =1 for polar coordinate system (if P is a 2-dimensional plot) or spherical coordinate system (if P is a 3-dimensional plot) and m = 2 for cylindrical coordinate system (if P is a 3-dimensional plot)

OUTPUT

(The m-th plot of graph G is changed to the new coordinate system)

EXAMPLES

```
// Examples for: setGraphPlotCoordSystem(G,m,n)

// Perform setGraphPlotCoordSystem(G2,m2,n2) for
// m2 = 0, n2 = 2 where G2 is the 2-dimensional
// graph generated by thefunction new2DGraph and
// containing a plot P2

G2 = new2DGraph("setGraphPlotCoordSystemGraph");
x ={1,2,3};
P2 = new2DDataPlot(" ",x,x);
addplot(G2,P2);
m2 = 0;
n2 = 1;
setGraphPlotCoordSystem(G2,m2,n2);

//     The following statement is for display the result

System2 = getGraphPlotCoordSystem(G2,m2);

// Result:
//       System2:  1

// Perform setGraphPlotCoordSystem(G3,m3,n3) for
// m3 = 0, n3 = 2 where G3 is the 3-dimensional
// graph generated by thefunction new3DGraph and
```

```
// containing a plot P3

G3 = new3DGraph("setGraphPlotCoordSystem3DGraph");
x ={1,2,3};
P3 = new3DDataPlot(" ",x,x,x);
addplot(G3,P3);
m3 = 0;
n3 = 2;
setGraphPlotCoordSystem(G3,m3,n3);

//     The following statement is for display the result

System3 = getGraphPlotCoordSystem(G3,m3);

// Result:
//       System3:  2
```

ALGORITHM AND COMMENTS

The coordinate system parameter n is also available as a HiQ-Script Language Constant; the values are: <cartesian>, <polar>, <spherical> and <cylindrical>.

In order to change the coordinate system from cartesian to polar, spherical, or cylindrical, you must follow the sequence:

1) addPlot() (adds the plot to the graph)
2) setGraphPlotCoordSystem() (changes coordinate system)
3) Pc = getPlot(Graph, m) (to copy the plot)
4) removePlot(Graph, m) (destroy the original plot)
5) addPlot(Graph, Pc) (redraw with the non-cartesian coordinates)

■ setGraphPlotDisplayFormat

FUNCTION

setGraphPlotDisplayFormat (G, m, n)

PURPOSE

Change the display format of a plot in a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

n (Integer Scalar): the display format of the m-th plot in graph G where n has the following effect,

	G is two-dimensional	G is three-dimensional
n = 0	line plot	surface plot
n = 1	point plot	point plot
n = 2	point-line plot	contour plot

OUTPUT

(The m-th plot in graph G is changed to the new display format)

EXAMPLES

```
// Examples for: setGraphPlotDisplayFormat(G,m,n)

// Perform setGraphPlotDisplayFormat(G2,m2,n2) for
// m2 =2, n2= 1 where G2 isthe 2-dimensional graph
// containing 3 plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets X, Y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("setGraphPlotDisplayFormatGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
m2 = 2;
n2 = 1;
setGraphPlotDisplayFormat(G2,m2,n2);

// The following statement is for displaying the
// resultant display format in G2

Format2 = getGraphPlotDisplayFormat(G2,m2);

// Result:
// Format2:    1

// Perform setGraphPlotDisplayFormat(G3,m3) for m3 =0
// n3 = 2; where G3 isthe 3-dimensional graph containing
// a plot generated by functions new3DGraph,new3DDataPlot
// and addPlot using the data sets x, y, z listed below
```

```
x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotDisplayFormat3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = 2;
setGraphPlotDisplayFormat(G3,m3,n3);

// The following statement is for displaying the
// resultant display format in G3

Format3 = getGraphPlotDisplayFormat(G3,m3);
// Result:    Format3:     2
```

SEE ALSO

setPlotDisplayFormat

ALGORITHM AND COMMENTS

The display format parameter n is also available as a HiQ Language Constant; the values are: <curve>, <surface>, <point>, <connected>, <contour>.

■ setGraphPlotEdgeMode

FUNCTION

setGraphPlotEdgeMode (G, m, n)

PURPOSE

Change the edge mode of a 3-dimensional plot in a graph

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

n (Integer Scalar): the edge mode of the m-th plot in graph G where n = 0 for edges to be removed and n = 1 for edges to be drawn

OUTPUT

(The m-th plot in graph G is changed to the new edge mode)

EXAMPLE

```
// An example for: setGraphPlotEdgeMode(G,m,n)

// Perform setGraphPlotEdgeMode(G3,m3,n3) for m3 =0
// n3 = 1 where G3 isthe 3-dimensional graph containing
// a plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets X, Y, Z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotEdgeModeGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = 1;
setGraphPlotEdgeMode(G3,m3,n3);

// The following statement is for displaying the result

Mode3 = getGraphPlotEdgeMode(G3,m3);

// Result:    Mode3:      1
```

ALGORITHM AND COMMENTS

If the m-th plot in graph G is not a faceted 3-dimensional plot, an error message will be returned.

■ setGraphPlotFillColor

FUNCTION

setGraphPlotFillColor (G, m, n)

PURPOSE

Change the color to fill facets of a 3-dimensional plot in a graph

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

n (Integer Scalar): the fill color of the m-th plot in graph G where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

OUTPUT

(The m-th plot of graph G is changed with the new fill color)

EXAMPLE

```
// An example for: setGraphPlotFillColor(G,m,n)

// Perform setGraphPlotFillColor(G3,m3,n3) for m3 =0
// n3 = 5 where G3 isthe 3-dimensional graph containing
// a plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotFillColorGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = 5;
setGraphPlotFillColor(G3,m3,n3);

// The following statement is for displaying the result

FillColor3 = getGraphPlotFillColor(G3,m3);

// Result:
// FillColor3:   5
```

SEE ALSO

setGraphPlotShading, setGraphPlotDisplayFormat

ALGORITHM AND COMMENTS

The fill color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

If the m-th plot in graph G is not a faceted 3-dimensional plot, an error message will be returned.

■ setGraphPlotLineColor

FUNCTION

setGraphPlotLineColor (G, m, n)

PURPOSE

Change the color used to draw edges of a plot in a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

n (Integer Scalar): the line color of the m-th plot in graph G where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow

OUTPUT

(The m-th plot of graph G is changed to the new line color)

EXAMPLES

```
// Examples for: setGraphPlotLineColor(G,m,n)

// Perform setGraphPlotLineColor(G2,m2,n2) for
// m2 =2, n2= 7 where G2 isthe 2-dimensional graph
// containing 3 plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets X, Y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("setGraphPlotLineColorGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
m2 = 2;
n2 = 7;
setGraphPlotLineColor(G2,m2,n2);

// The following statement is for displaying the
// resultant line color of P23 in G2

LineColor2 = getGraphPlotLineColor(G2,m2);

// Result:
// LineColor2: 7
```

```
// Perform setGraphPlotLineColor(G3,m3) for m3 =0
// n3 = 3; where G3 isthe 3-dimensional graph containing
// a plot generated by the functions new3DGraph,new3DDataPlot
// and addPlot using the data sets x, y, z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotLineColor3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = 3;
setGraphPlotLineColor(G3,m3,n3);

// The following statement is for displaying the
// resultant line color of P3 in  G3

LineColor3 = getGraphPlotLineColor(G3,m3);

// Result:
// LineColor3:  3
```

ALGORITHM AND COMMENTS

The line color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

■ setGraphPlotLineWidth

FUNCTION

setGraphPlotLineWidth (G, m, n)

PURPOSE

Change the line width to draw edges of a plot in a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

n (Integer Scalar): the line width of the m-th plot in graph G; the possible values are 1 through 255, where 255

is the thickest

OUTPUT

(The m-th plot of graph G is changed with the new line width)

EXAMPLES

```
// Examples for: setGraphPlotLineWidth(G,m,n)

// Perform setGraphPlotLineWidth(G2,m2,n2) for
// m2 =2, n2= 7 where G2 isthe 2-dimensional graph
// containing 3 plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets X, Y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};

G2 = new2DGraph("setGraphPlotLineWidthGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
m2 = 2;
n2 = 4;
setGraphPlotLineWidth(G2,m2,n2);

// The following statement is for displaying the
// resultant line width of P23 in G2

LineWidth2 = getGraphPlotLineWidth(G2,m2);

// Result:
// LineWidth2: 4

// Perform setGraphPlotLineWidth(G3,m3) for m3 =0
// n3 = 3; where G3 isthe 3-dimensional graph containing
// a plot generated by the functions new3DGraph,new3DDataPlot
// and addPlot using the data sets X, Y, Z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotLineWidth3DGRaph");
P3 = new3DDataPlot(" ",x,y,z);
```

```
addPlot(G3, P3);
m3 = 0;
n3 = 3;
setGraphPlotLineWidth(G3,m3,n3);

// The following statement is for displaying the
// resultant line width of P3 in  G3

LineWidth3 = getGraphPlotLineWidth(G3,m3);

// Result:
// LineWidth3:  3
```

ALGORITHM AND COMMENT

The Graph Editor interface only allows setting the line widths in the range from 1 to 4.

■ setGraphPlotMarkerColor

FUNCTION

setGraphPlotMarkerColor (G, m, n)

PURPOSE

Change the color to draw vertices of a plot in a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

n (Integer Scalar): the marker color of the m-th plot in graph G where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

OUTPUT

(The m-th plot of graph G is changed with the new marker color)

EXAMPLES

```
// Examples for: setGraphPlotMarkerColor(G,m,n)

// Perform setGraphPlotMarkerColor(G2,m2,n2) for
// m2 =2, n2= 7 where G2 isthe 2-dimensional graph
// containing 3 plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets x, y
// listed below
```

```
x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("setGraphPlotMarkerColorGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
m2 = 2;
n2 = 4;
setGraphPlotMarkerColor(G2,m2,n2);

// The following statement is for displaying the
// resultant marker color of P23 in G2

MarkerColor2 = getGraphPlotMarkerColor(G2,m2);

// Result:
// MarkerColor2: 4

// Perform setGraphPlot
// n3 = 3; where G3 isthe 3-dimensional graph containing
// a plot generated by the functions new3DGraph,new3DDataPlot
// and addPlot using the data sets x, y, z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotMarkerColor3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = 3;

setGraphPlotMarkerColor(G3,m3,n3);

// The following statement is for displaying the
// resultant marker color of P3 in  G3

MarkerColorC3 = getGraphPlotMarkerColor(G3,m3);

// Result:
// MarkerColor3:  3
```

SEE ALSO

setGraphPlotDisplayFormat

ALGORITHM AND COMMENTS

The marker color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

Markers are seen only in point plots or connected plots.

■ setGraphPlotMarkerStyle

FUNCTION

setGraphPlotMarkerStyle (G, m, n)

PURPOSE

Change the marker style to draw vertices of a plot in a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

n (Integer Scalar): the marker style of the m-th plot in graph G where n = 0 for circular-shaped marker, n = 1 for square-shaped marker, n = 2 for diamond-shaped marker, n = 3 for triangular-shaped marker, n = 4 for cross-shaped marker and n = 5 for x-shaped marker (See the Algorithm and Comments section)

OUTPUT

(The m-th plot in graph G is changed to the new marker style)

EXAMPLES

```
// Examples for: setGraphPlotMarkerStyle(G,m,n)

// Perform setGraphPlotMarkerStyle(G2,m2,n2) for
// m2 =2, n2= 7 where G2 isthe 2-dimensional graph
// containing 3 plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets X, Y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("setGraphPlotMarkerStyleGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
```

```
addPlot(G2, P23);
m2 = 2;
n2 = 4;
setGraphPlotMarkerStyle(G2,m2,n2);

// The following statement is for displaying the
// resultant marker style of P23 in G2

MarkerStyle2 = getGraphPlotMarkerStyle(G2,m2);

// Result:
// MarkerStyle2: 4

// Perform setGraphPlot
// n3 = 3; where G3 isthe 3-dimensional graph containing
// a plot generated by the functions new3DGraph,new3DDataPlot
// and addPlot using the data sets X, Y, Z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotMarkerStyle3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = 3;
setGraphPlotMarkerStyle(G3,m3,n3);

// The following statement is for displaying the
// resultant marker style of P3 in  G3

MarkerStyle3 = getGraphPlotMarkerStyle(G3,m3);

// Result:
// MarkerStyle3:  3
```

SEE ALSO

setGraphPlotDisplayFormat

ALGORITHM AND COMMENTS

The marker style parameter n is also available as a HiQ-Script Language Constant; the values are: <circular>, <square>, <diamond>, <triangular>, <cross>, and <x_shape>.

Markers are seen only in point plots or connected plots.

■ setGraphPlotProjectedContour

FUNCTION

setGraphPlotProjectedContour (G, m, n)

PURPOSE

Change the contour placement of a 3-dimensional plot in a graph to project on the contour coordinate plane or overlay on the plot itself

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

n (Integer Scalar): the contour placement of the m-th plot in graph G where n = 0 for overlaid contours and n = 1 for projected contours

OUTPUT

(The m-th plot in graph G is changed to the new contour placement)

EXAMPLES

```
// An example for: setGraphPlotProjectedContour(G,m,n)

// Perform setGraphPlotProjectedContour(G3,m3,n3) for
// m3 =0 n3 = 1 where G3 isthe 3-dimensional graph
// containing a plot generated by the functions new3DGraph,
// new3DDataPlot and addPlot using the data sets x, y, z
// listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
G3 = new3DGraph("setGraphPlotProjectedContourGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
m3 = 0;
n3 = 1;
setGraphPlotProjectedContour(G3,m3,n3);
setGraphPlotContourPlane(G3,m3,<yz_plane>);

// The following statement is for displaying the result

Contour3 = getGraphPlotProjectedContour(G3,m3);
```

```
// Result:
// Contour3:   1
```

ALGORITHM AND COMMENTS

The contour placement parameter n is also available as a HiQ Language Constant; the values are: <overlaid> and <projected>.

The default contour placement for newly-created graphs is <overlaid>. If you set the projected contour to <projected>, you must also call setGraphPlotContourPlane() to specify an axis to project onto.

If the m-th plot in graph G is not a faceted 3-dimensional plot, an error message will be returned.

■ setGraphPlotTitle

FUNCTION

setGraphPlotTitle (G, m, T)

PURPOSE

Change the title of a plot in a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where $m \geq 0$

T (String): the title of the m-th plot in graph G

OUTPUT

(The title of the m-th plot in graph G is changed to the new title)

EXAMPLES

```
// Examples for: setGraphPlotTitle(G,m,T)

// Perform setGraphPlotTitle(G2,m2,Title2) for
// m2 = 0, Title2 ="2D Sample Plot" where G2 is
// the 2-dimensional graph generated by thefunction
// new2DGraph and containing a plot P2

G2 = new2DGraph("setGraphPlotTitleGraph");
x ={1,2,3};
P2 = new2DDataPlot(" ",x,x);
addplot(G2,P2);
m2 = 0;
Title2 = "2D Sample Plot";
setGraphPlotTitle(G2,m2,Title2);
```

```
//    The following statement is for displaying the result

PlotTitle2 = getGraphPlotTitle(G2,m2);

// Result:
// PlotTitle2:  2D Sample Plot

// Perform setGraphPlotTitle(G3,m2,Title3) for
// m3 = 0, Title3 ="3D Sample Plot" where G3 is
// the 3-dimensional graph generated by thefunction
// new3DGraph and containing a plot P3

G3 = new3DGraph("setGraphPlotTitle3DGraph");
x ={1,2,3};
P3 = new3DDataPlot(" ",x,x,x);
addplot(G3,P3);
m3 = 0;
Title3 = "3D Sample Plot";
setGraphPlotTitle(G3,m3,Title3);

//    The following statement is for displaying the result
PlotTitle3 = getGraphPlotTitle(G3,m3);

// Result:
// PlotTitle3:  3D Sample Plot
```

ALGORITHM AND COMMENTS

Note: Plot titles are displayed in 3D graphs only if the legend option is selected

■ setGraphPlotTitleColor

FUNCTION

setGraphPlotTitleColor (G, m, n)

PURPOSE

Change the color used to draw title of a plot in a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

m (Integer Scalar): the index of the plot in graph G where m ≥ 0

n (Integer Scalar): the title color to be changed to where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

OUTPUT

(The title of the m-th plot of graph G is changed to the new title color)

EXAMPLES

```
// Examples for: setGraphPlotMarkerStyle(G,m,n)

// Perform setGraphPlotMarkerStyle(G2,m2,n2) for
// m2 =2, n2= 7 where G2 isthe 2-dimensional graph
// containing 3 plots generated by the functions new2DGraph,
// new2DDataPlot and addPlot using the data sets X, Y
// listed below

x = {  0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.0};
y = {  0; 0.01; 0.04; 0.09; 0.16; 0.25; 0.36; 0.49; 0.64; 0.81; 1};
G2 = new2DGraph("setGraphPlotMarkerStyleGraph");
P21 = new2DDataPlot(" ",x,y);
P22 = new2DDataPlot(" ",y,x);
P23 = new2DDataPlot(" ",x,x);
addPlot(G2, P21);
addPlot(G2, P22);
addPlot(G2, P23);
m2 = 2;
n2 = 4;
setGraphPlotTitleColor(G2,m2,n2);

// The following statement is for displaying the
// resultant title color of P23 in G2

TitleColor2 = getGraphPlotTitleColor(G2,m2);

// Result:
// TitleColor2: 4

// Perform setGraphPlot
// n3 = 3; where G3 isthe 3-dimensional graph containing
// a plot generated by the functions new3DGraph,new3DDataPlot
// and addPlot using the data sets x, y, z listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
       G3 = new3DGraph("setGraphPlotMarkerStyle3DGraph");
P3 = new3DDataPlot(" ",x,y,z);
addPlot(G3, P3);
```

```
m3 = 0;
n3 = 3;
setGraphPlotTitleColor(G3,m3,n3);

// The following statement is for displaying the
// resultant title color of P3 in  G3

TitleColor3 = getGraphPlotTitleColor(G3,m3);

// Result:
// TitleColor3:  3
```

ALGORITHM AND COMMENTS

The title color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

The plot title of a 3-dimensional plot is not displayed.

■ setGraphShading

FUNCTION

setGraphShading (G, m)

PURPOSE

Change the shading mode of a 3-dimensional graph

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the shading mode of the graph where m = 0 for wireframe, m = 1 for hidden-line, m= 2 for shading in height and m = 3 for shading with light sources

OUTPUT

(The graph G is redrawn to the new shading mode)

EXAMPLES

```
// An example for: setGraphShading(G,m)

// Perform setGraphShading(G3,m3) for m3 = 3 where
// G3 is the 3-dimensional graph  generated by
// the functions new3DGraph

G3 = new3DGraph("setGraphShadingGraph");
m3 = 3;
setGraphShading(G3,m3);
```

```
// The following statement is for displaying the result
Shading3 = getGraphShading(G3);
// Result:  Shading3:  3
```

ALGORITHM AND COMMENTS

This function supersedes the functions wireFrameGraph, heightShading, hiddenLineGraph, and lightSource-Shading.

If graph G is not a 3-dimensional graph, then an error message will be returned.

The shading mode parameter m is available as a HiQ Language Constant; the values are: <wire>, <line>, <height>, and <light>.

■ setGraphTitle

FUNCTION

setGraphTitle (G, T)

PURPOSE

Change the title of a graph

INPUT

G (Graph): a 2- or 3-dimensional graph

T (String): title of the graph

OUTPUT

(The graph G is redrawn with the new graph title)

EXAMPLES

```
// Examples for: setGraphTitle(G,T)

// Perform setGraphShading(G2,Title2) where G2
// is the 2-dimensional graph generated by the function
// new3DGraph with blank title and Title2 is the string
// "2D Sample Graph"

G2 = new3DGraph("setGraphTitleGraph");
Title2 = "2D Sample Graph";
setGraphTitle(G2,Title2);

// The following statement is for displaying the result

Title2 = getGraphTitle(G2);

// Result:  Title2:  2D Sample Graph
```

```
// Perform setGraphShading(G3,Title3) where G3
// is the 3-dimensional graph cgenerated by the functions
// new3DGraph with blank title and Title3 is the string
// "3D Sample Graph"

G3 = new3DGraph("setGraphTitle3DGraph");
Title3 ="3D Sample Graph";
setGraphTitle(G3,Title3);

// The following statement is for displaying the result

Title3 = getGraphTitle(G3);
// Result:   Title3:  3D Sample Graph
```

■ setLightDirection

FUNCTION

setLightDirection (G, m, theta, phi)

PURPOSE

Change the direction of a light source of a 3-dimensional graph

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the index of the light source where m = 0, 1, 2 or 3

theta (Real Scalar): the azimuth of the light source in degrees where $0 \leq \text{theta} \leq 360$

phi (Real Scalar): the ascension of the light source in degrees where $-90 \leq \text{phi} \leq 90$

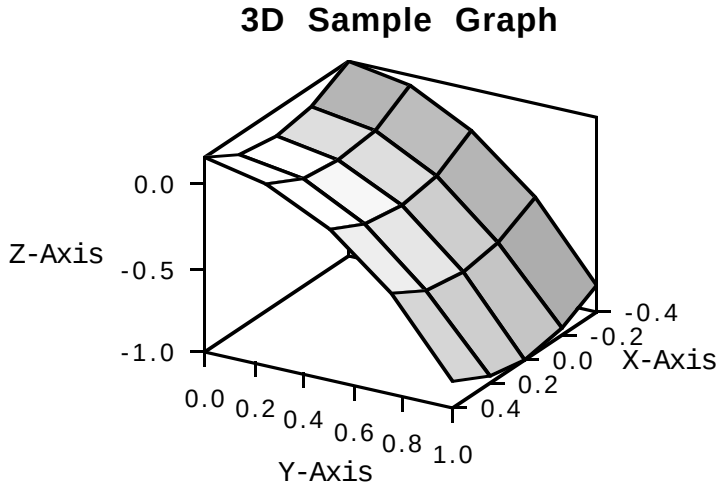
OUTPUT

(The graph G is redrawn to the new light direction)

EXAMPLES

```
// An example for: setLightDirection(G,m,theta,phi)

// Perform setLightDirection(G3,m3,theta,phi) where
// G3 is the 3-dimensional graph  generated by the
// functions new3DGraph, m3 = 1, theta = 135 and phi= 60

G3 = new3DGraph("3D Sample Graph");
// Turn on the light sources
setGraphShading(G3,3);
m3 = 1;
theta = 135;
```

```
phi= 60;
setLightDirection(G3,m3,theta,phi);

// The following statements are displaying for the
// result of setLightDirection using the 3-dimensional
// plot, P3, generated by the data sets x,y listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ",x,y,z);
addPlot(G3, P3);

// Result: (Display of effect of setLightDirection on G3)
```



SEE ALSO

setGraphShading,setLightType,setLightState, setLightIntensity

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph or the light source is an ambient light, then an error message will be returned.

No effect is seen if the current shading mode is notshading with light sources.

No effect is seen on a curved plot in graph G.

■ setLightIntensity

FUNCTION

setLightIntensity (G, m, n)

PURPOSE

Change the intensity of a light source of a 3-dimensional graph

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the index of the light source where m = 0, 1, 2 or 3

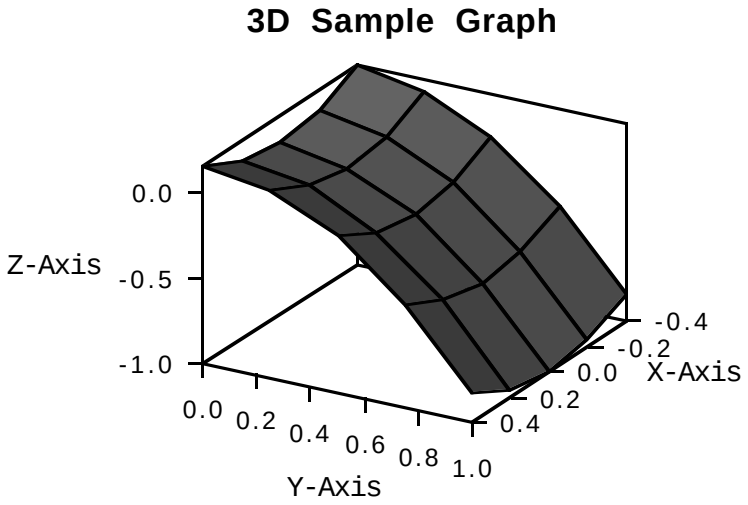
n (Integer Scalar): the intensity of the light source where $0 \leq n \leq 1$

OUTPUT

(The graph G is redrawn to the new light intensity)

EXAMPLES

```
// An example for: setLightIntensity(G,m,n)

// Perform setLightIntensity(G3,m3,n3) where
// G3 is the 3-dimensional graph  generated by the
// functions new3DGraph, m3 = 1, n3 = 0.5

G3 = new3DGraph("3D Sample Graph ");
// Turn on the light sources
setGraphShading(G3,3);
m3 = 1;
n3 = 0.2;
setLightIntensity(G3,m3,n3);

// The following statements are displaying for the
// result of setLightIntensity using the 3-dimensional
// plot, P3, generated by the data sets x,y listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ",x,y,z);
addPlot(G3, P3);

// Result: (Display of effect of setLightIntensity on G3)
```



SEE ALSO

setGraphShading,setLightType, setLightState,setLightDirection

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph or the light source is an ambient light, then an error message will be returned.

No effect is seen if the current shading mode is notshading with light sources.

No effect is seen on a curved plot in graph G.

■ setLightState

FUNCTION

setLightState (G, m, n)

PURPOSE

Change the state of a light source of a 3-dimensional graph

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the index of the light source where m = 0, 1, 2 or 3

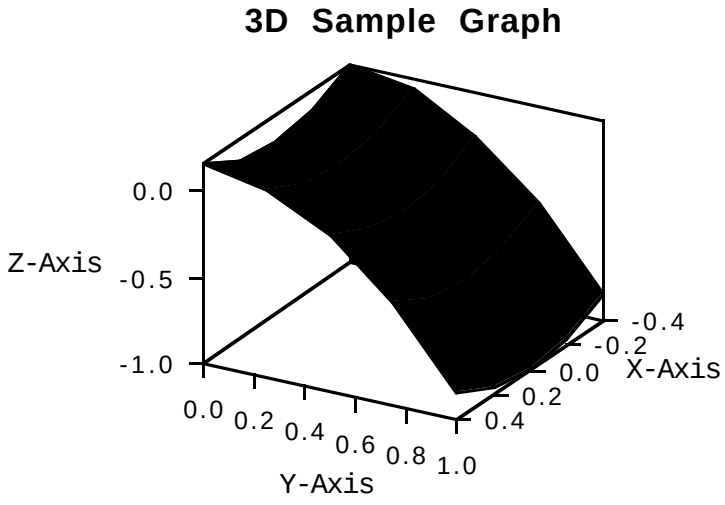
n (Integer Scalar): the state of the light source where n = 0 for turning the light source off, and n = 1 for turning on the light source

OUTPUT

(The graph G is redrawn to the new light state)

EXAMPLE

```
// An example for: setLightState(G,m,n)

// Perform setLightState(G3,m3,n3) where
// G3 is the 3-dimensional graph  generated by the
// functions new3DGraph, m3 = 1, n3 = 0 (i.e., turn off the light
// source)

G3 = new3DGraph("3D Sample Graph");
// Turn on the light sources
setGraphShading(G3,3);
m3 = 1;
n3 = 0;
setLightState(G3,m3,n3);

// The following statements are displaying for the
// result of setLightState using the 3-dimensional
// plot, P3, generated by the data sets x,y listed
//  below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ",x,y,z);
addPlot(G3, P3);

// Result: (Display of effect of setLightState on G3)
```



SEE ALSO

setGraphShading,setLightType,setLightIntensity, setLightDirection

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph or the light source is an ambient light, then an error message will be returned.

No effect is seen if the current shading mode is notshading with light sources.

No effect is seen on a curved plot in graph G.

■ setLightType

FUNCTION

setLightType (G, m, n)

PURPOSE

Change the type of a light source in a 3-dimensional graph

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the index of the light source where m = 0 , 1, 2 or 3

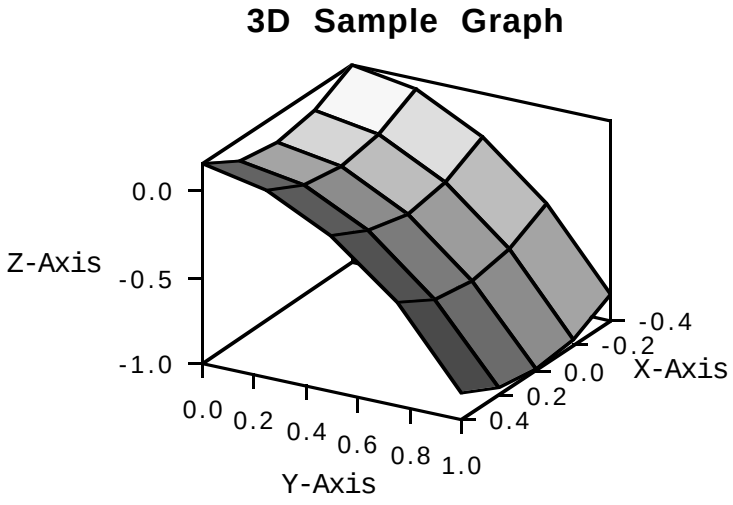
n (Integer Scalar): the type of the light source where n = 0 for ambient light and n = 1 for directional light

OUTPUT

(The graph G is redrawn to reflect the new light type)

EXAMPLES

```
// An example for: setLightType(G,m,n)

// Perform setLightType(G3,m3,n3) where
// G3 is the 3-dimensional graph  generated by the
// functions new3DGraph, m3 = 1, n3 = 0

G3 = new3DGraph("3D Sample Graph");
// Turn on the light sources
setGraphShading(G3,3);
m3 = 1;
n3 = 0;
setLightType(G3,m3,n3);

// The following statements are displaying for the
// result of setLightType using the 3-dimensional
// plot, P3, generated by the data sets x,y listed below

x = { -0.4; -0.2; 0; 0.2; 0.4 };
y = {0; 0.25; 0.5; 0.75; 1};
z = {   0.16,   0.0975,  -0.09,  -0.4025,   -0.84;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0,     -0.0625,  -0.25,  -0.5625,   -1;
        0.04,  -0.0225,  -0.21,  -0.5225,   -0.96;
        0.16,   0.0975,  -0.09,  -0.4025,   -0.84};
P3 = new3DDataPlot("   ",x,y,z);
addPlot(G3, P3);
// Result: (Display of effectness of setLightType on G3)
```



SEE ALSO

setGraphShading,setLightState,setLightIntensity, setLightDirection

ALGORITHM AND COMMENTS

The light source type parameter n is also available as a HiQ Language Constant; the values are: <ambient> and <directional>.

If graph G is not a 3-dimensional graph, then an error message will be returned.

No effect is seen if the current shading mode is notshading with light sources.

Ambient lighting is non-directional. If more than one light source is set to "ambient", the intensity of ambient lighting is set to the sum of the intensities of all the ambient light sources whose state is "on".

■ setPlotCoordSystem

FUNCTION

setPlotCoordSystem (P, m)

PURPOSE

Change the coordinate system of a plot

INPUT

P (Plot): a 2- or 3-dimensional plot

m (Integer Scalar): the coordinate system of the plot where m = 0 for Cartesian coordinates; m = 1 for polar coordinates (if P is a 2-dimensional plot) or spherical coordinates (if P is a 3-dimensional plot); and m = 2 for (3-dimensional) cylindrical coordinates.

OUTPUT

(The graph is redrawn to the new plot coordinate system)

EXAMPLES

```
//    EXPlot2is the 2-dimensional plot generated by the
//   function new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
EXPlot2 = new2DDataPlot(" ",x,y);
m2 = 1;
setPlotCoordSystem(EXPlot2,m2);

//    The following statement is for displaying the result

EXsetPCoordS2 = getPlotCoordSystem(EXPlot2);

// Result:
//     EXsetPCoordS2:  1

// Perform
//    EXPlot3 is the 3-dimensional plot generated by the
// function new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0,      0;
             0, -0.05};
EXPlot3 = new3DDataPlot(" ",x,y,z);
m3 = 2;
setPlotCoordSystem(EXPlot3,m3);

//    The following statement is for displaying the result

EXsetPCoordS3 = getPlotCoordSystem(EXPlot3);

// Result:
//     EXsetPCoordS3:  2
```

SEE ALSO

getPlotCoordSystem

■ setPlotDisplayFormat

FUNCTION

setPlotDisplayFormat (P, n)

PURPOSE

Change the display format of a plot

INPUT

P (Plot): a 2- or 3-dimensional plot

n (Integer Scalar): the display format of the plot P where n has the following effect:

	P is two-dimensional	**P is three-dimensional**
n=0	line plot	surface plot
n=1	point plot	point plot
n=2	point-line plot	contour plot

OUTPUT

None

EXAMPLES

```
// P2 is the 2-dimensional plot generated by the
// function new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot(" ",x,y);
n2 = 1;
setPlotDisplayFormat(P2, n2);
// The following statement is for displaying the result

Format2 = getPlotDisplayFormat(P2);

// Result:   Format2: 1

// P3 is the 3-dimensional plot generated by the
// function new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
```

```
z = {0,      0;
     0, -0.05};
P3 = new3DDataPlot(" ",x,y,z);
n3 = 2;
setPlotDisplayFormat(P3, n3);
// The following statement is for displaying the result

Format3 = getPlotDisplayFormat(P3);
// Result:   Format3: 2
```

■ setPlotFillColor

FUNCTION

setPlotFillColor (P, m)

PURPOSE

Change the color used to fill facets of a 3-dimensional plot

INPUT

P (Plot): a 3-dimensional plot

m (Integer Scalar): the color index where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

OUTPUT

None

EXAMPLES

```
 //An example for: getPlotFillColor(P,m)

// Perform getPlotFillColor(P3,m3) for m3= 1 where
// P3  is the 3-dimensional plot generated by the function
// new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0,  0;
        0, -0.05};
P3 = new3DDataPlot(" ",x,y,z);
m3 = 1;
setPlotFillColor(P3,m3);

// The following statement is for displaying the result

FillColor3 = getPlotFillColor(P3);
```

```
// Result:
// FillColor3:     1
```

SEE ALSO

setGraphShading, setPlotDisplayFormat

ALGORITHM AND COMMENTS

The fill color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

If plot P is not a 3-dimensional plot, then an error message will be returned.

No effect is seen if the current shading mode is wireframe.

No effect is seen if the current display format is point plot.

■ setPlotLineColor

FUNCTION

setPlotLineColor (P, m)

PURPOSE

Change the color used to draw lines

INPUT

P (Plot): a 2- or 3-dimensional plot

m (Integer Scalar): the color index where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

OUTPUT

(The color attribute of the plot is changed)

EXAMPLES

```
// P2 is the 2-dimensional plot generated by the
// function new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot(" ",x,y);
m2 = 7;
setPlotLineColor(P2,m2);

// The following statement is for displaying the result
```

```
EXsetOKubeC2 = getPlotLineColor(P2);

// Result:
// EXsetPLineC2:     7

// Perform setPlotLineColor(P3,m3) for m3 = 2 where
// P3 is the 3-dimensional plot generated by the
// function new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0,      0;
           0, -0.05};
P3 = new3DDataPlot(" ",x,y,z);
m3 = 2;
setPlotLineColor(P3,m3);

// The following statement is for displaying the result

EXsetPLineC3 = getPlotLineColor(P3);

// Result:
// EXsetPLineC3:     2
```

ALGORITHM AND COMMENTS

The line color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

■ setPlotLineWidth

FUNCTION

setPlotLineWidth (P, l)

PURPOSE

Change the width of lines used to draw a plot

INPUT

P (Plot): a 2- or 3-dimensional plot

l (Integer Scalar): the line width of the plot, which can assume one of the values 1 through 255, where 255 is the thickest (See the Algorithm and Comments section)

OUTPUT

(The linewidth attribute of the plot is changed)

EXAMPLES

```
// P2 is the 2-dimensional plot generated by the
// function new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot(" ");
m2 = 7;
setPlotLineWidth(P2,m2);

// The following statement is for displaying the result

LineWidth2 = getPlotLineWidth(P2);

// Result:
// LineWidth2:     7

// Perform setPlotLineWidth(P3,m3) for m3 = 2 where
// P3 is the 3-dimensional plot generated by the
// function new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0,      0;
            0, -0.05};
P3 = new3DDataPlot(" ");
m3 = 2;

// The following statement is for displaying the result

LineWidth3 = getPlotLineWidth(P3);

// Result:
// LineWidth3:
```

ALGORITHM AND COMMENTS

The Graph Editor interface only allows you to set the line widths between the values 1 and 4.

■ setPlotMarkerColor

FUNCTION

setPlotMarkerColor (P, m)

PURPOSE

Change the color used to draw vertices of a plot

INPUT

P (Plot): a 2- or 3-dimensional plot

m (Integer Scalar): the color index where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

OUTPUT

(The markercolor attribute is changed)

EXAMPLES

```
// P2 is the 2-dimensional plot generated by the
// function new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot(" ");
pointPlot(P2);
m2 = 7;
setPlotMarkerColor(P2,m2);

// The following statement is for displaying the result

MarkerColor2 = getPlotMarkerColor(P2);

// Result:
// MarkerColor2:    7

// Perform setPlot
// P3 is the 3-dimensional plot generated by the
// function new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0,      0;
             0, -0.05};
P3 = new3DDataPlot(" ");
m3 = 2;
setPlotMarkerColor(P3,m3);

// The following statement is for displaying the result

MarkerColor3 = getPlotMarkerColor(P3);

// Result:
// MarkerColor3:     2
```

SEE ALSO

setPlotDisplayFormat

ALGORITHM AND COMMENTS

The marker color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

Markers are seen only in point plots or connected plots.

■ setPlotMarkerStyle

FUNCTION

setPlotMarkerStyle (P, m)

PURPOSE

Change the marker style used to draw vertices of a plot

INPUT

P (Plot): a 2- or 3-dimensional plot

m (Integer Scalar): the marker style of the plot P where m = 0 for circular-shaped marker, m = 1 for square-shaped marker, m = 2 for diamond-shaped marker, m = 3 for triangular-shaped marker, m = 4 for cross-shaped marker and m = 5 for x-shaped marker

OUTPUT

(The marker style attribute is changed)

EXAMPLE

```
// Examples for:

// Perform
// P2is the 2-dimensional plot generated by the
// function new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot(" ");
m2 = 5;
setPlotMarkerStyle(P2,m2);

// The following statement is for displaying the result

MarkerStyle2 = getPlotMarkerStyle(P2);

// Result:
```

```
// MarkerStyle2:     5

// Perform setPlot
// P3 is the 3-dimensional plot generated by the
// function new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0,      0;
             0, -0.05};
P3 = new3DDataPlot(" ");
m3 = 2;
setPlotMarkerStyle(P3,m3);

// The following statement is for displaying the result

MarkerStyle3 = getPlotMarkerStyle(P3);

// Result:
// MarkerStyle3:     2
```

SEE ALSO

setPlotDisplayFormat

ALGORITHM AND COMMENTS

The marker style parameter m can also be set using one of the HiQ-Script Language Constants; the values are: <circular>, <square>, <diamond>, <triangular>, <cross>, and <x_shape>.

Markers are seen only in point plots or connected plots.

■ setPlotTitle

FUNCTION

setPlotTitle (P, T)

PURPOSE

Change the title of a plot

INPUT

P (Plot): a 2- or 3-dimensional plot

T (String): the title of the plot

OUTPUT

(The title of Plot P is changed to string T)

EXAMPLE

```
// Examples for: setPlotTitle(P,T)

// Perform setPlotTitle(P2,Title2) where P2 is the
// 2-dimensional plot generated by the function new2DDataPlot
//  with blank title and Title2 is the string "2D Plot"

x={0; 0.1; 1};
P2 = new2DDataPlot("setPlotTitleGraph",x,x);
Title2 = "2D Plot";
setPlotTitle(P2,Title2);

// The following statements are for displaying the resultant
// title of plot P2 in graph G2

G2 = new2DGraph("setPlotTitle3DGraph");
addPlot(G2,P2);
PlotTitle2 = getGraphPlotTitle(G2,0);

// Result:
// PlotTitle2:  2D Plot

// Perform setPlotTitle(P3,Title3) where P3 is the
// 3-dimensional plot generated by the function new3DDataPlot
//  with blank title and Title3 is the string "3D Plot"

x={0; 0.1; 1};
P3 = new3DDataPlot("setPlotTitle3DGraph",x,x,x);
Title3 = "3D Plot";
setPlotTitle(P3,Title3);

// The following statements are for displaying the resultant
// title of plot P3 in graph G3

G3 = new3DGraph(" ");
addPlot(G3,P3);
PlotTitle3 = getGraphPlotTitle(G3,0);

// Result:
// PlotTitle3:  3D Plot
```

ALGORITHM AND COMMENTS

Important Note:

Plot title is not displayed for a 3-dimensional plot, unless the graph legend option is selected.

■ setPlotTitleColor

FUNCTION

setPlotTitleColor (P, m)

PURPOSE

Change the color used to draw the title of a plot

INPUT

P (Plot): a 2- or 3-dimensional plot

m (Integer Scalar): the color index where: n = 0 for black, n = 1 for white, n = 2 for red, n = 3 for green, n = 4 for blue, n = 5 for cyan, n = 6 for magenta, n = 7 for yellow (See the Algorithm and Comments section)

OUTPUT

(The title color attribute is changed)

EXAMPLES

```
// Examples for:

// Perform
// P2 is the 2-dimensional plot generated by the
// function new2DDataPlot using the following data sets x,y

x = { 0; -0.2};
y = { 0; 0.25};
P2 = new2DDataPlot("setPlot");
m2 = 7;
setPlotTitleColor(P2,m2);

// The following statement is for displaying the result

TitleColor2 = getPlotTitleColor(P2);

// Result:
// TitleColor2:     7

// Perform setPlot
// P3 is the 3-dimensional plot generated by the
// function new3DDataPlot using the following data sets x,y,z

x = { 0; -0.2};
y = { 0; 0.25};
z = {   0,      0;
            0, -0.05};
P3 = new3DDataPlot("setPlot");
```

```
m3 = 2;

// The following statement is for displaying the result

TitleColor3 = getPlotTitleColor(P3);

// Result:
// TitleColor3:      NONE DISPLAYED
```

ALGORITHM AND COMMENTS

The title color parameters can also be set as HiQ-Script Language Constants in the parameter list; they are: <black>, <white>, <red>, <green>, <blue>, <cyan>, <magenta>, <yellow>.

Plot title is not displayed for a 3-dimensional plot, unless the graph legend option is selected.

■ setProjectionType

FUNCTION

setProjectionType (G, m)

PURPOSE

Change the projection type of a 3-dimensional graph

INPUT

G (Graph): a 3-dimensional graph

m (Integer Scalar): the projection type where m = 0 for orthographic projection, and m = 1 for perspective projection

OUTPUT

(The graph is redrawn to the new projection type)

EXAMPLES

```
// An example for: setProjectionType(G,m)
// Perform setProjectionType(G3,m3) for m3= 1 where
// G3 is the 3-dimensional graph generated by the
// function new3DGraph

G3 = new3DGraph("setProjectionTypeGraph");
m3 = 1;
setProjectionType(G3,m3);
// The following statement is for displaying the result
Type3 = getProjectionType(G3);
// Result:
// Type3:     1
```

SEE ALSO

focusCamera

ALGORITHM AND COMMENTS

The projection type can also be set by using the HiQ-Script Language Constants <orthographic> or <perspective>.

If graph G is not a 3-dimensional graph, then an error message will be returned.

■ surfacePlot

FUNCTION

surfacePlot(P)

PURPOSE

Render a graph hiding all lines not visible to the viewer and showing the data's surface

INPUT

P (Plot): a 3-dimensional plot

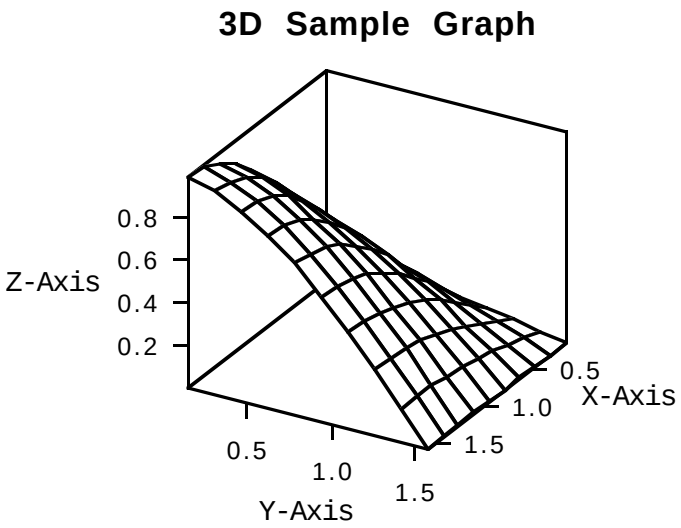
OUTPUT

An attribute of the plot P is changed. There is no visible effect until the plot P is displayed on a graph using the function addPlot().

EXAMPLES

```
// An example for: surfacePlot(P)

// Perform surfacePlot(P3) where P3 is the
// 3-dimensional plot generated by the function
// new3DDataPlot using the data sets x,y z generated
// by the following formulae

for i = 1 to 10 do
   x[i] = i*<pi>/20;
   y[i] = x[i];
end for;
for i = 1 to 10 do
   for j = 1 to 10 do
      z[i,j] = sin(x[i])*cos(y[j]);
   end for;
end for;
P3 = new3DDataPlot(" ",x,y,z);
surfacePlot(P3);
```

```
//    The following statements are for displaying the
//    resultant plot P3 in graph G3

G3 = new3DGraph("3D Sample Graph");
addPlot(G3,P3);

//    Result: (Display for the resultant plot P3 in graph G3)
```



SEE ALSO

linePlot, pointPlot, pointLinePlot

ALGORITHM AND COMMENTS

The same effect can be achieved using the function setPlotDisplayFormat

■ viewFromFront

FUNCTION

viewFromFront (G)

PURPOSE

View a 3-dimensional graph from the positive end of the x-axis; this corresponds to azimuth and ascension

values of zero degrees

INPUT

G (Graph): a 3-dimensional graph

OUTPUT

(The graph G is redrawn in its front view)

EXAMPLE

```
 //An example for: viewFromFront(G)

// Perform viewFromFront(G3) where G3 is the
// 3-dimensional graph generated by the function new3DGraph

G3 = new3DGraph("3D Graph View From Front");
viewFromFront(G3);

// Result: (Display of resultant graph G3)
```

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph, then an error message will be returned.

■ viewFromSide

FUNCTION

viewFromSide (G)

PURPOSE

View a 3-dimensional graph sideward from the positive end of the y-axis; this corresponds to an azimuth of -90 degrees and an ascension value of zero degrees

INPUT

G (Graph): a 3-dimensional graph

OUTPUT

(The graph G is redrawn in its side view)

EXAMPLES

```
// An example for: viewFromSide(G)
// Perform viewFromSide(G3) where G3 is the
// 3-dimensional graph generated by the function
// new3DGraph
G3 = new3DGraph("3D Graph View From Side");
```

```
viewFromSide(G3);
// Result: (Display of resultant graph G3)
```

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph, then an error message will be returned.

■ viewFromTop

FUNCTION

viewFromTop (G)

PURPOSE

View a 3-dimensional graph from the top, i.e., the positive end of the z-axis; this corresponds to azimuth and ascension values of 90 degrees.

INPUT

G (Graph): a 3-dimensional graph

OUTPUT

(The graph G is redrawn in its top view)

EXAMPLES

```
// An example for: viewFromTop(G)

// Perform viewFromTop(G3) where G3 is the
// 3-dimensional graph generated by the function
// new3DGraph

G3 = new3DGraph("3D Graph View From Top");
viewFromTop(G3);

// Result: (Display of resultant graph G3)
```

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph, then an error message will be returned.

■ wireFrameGraph

FUNCTION

wireFrameGraph(G)

PURPOSE

Render a graph in wireframe mode (all surfaces show through to the front).

INPUT

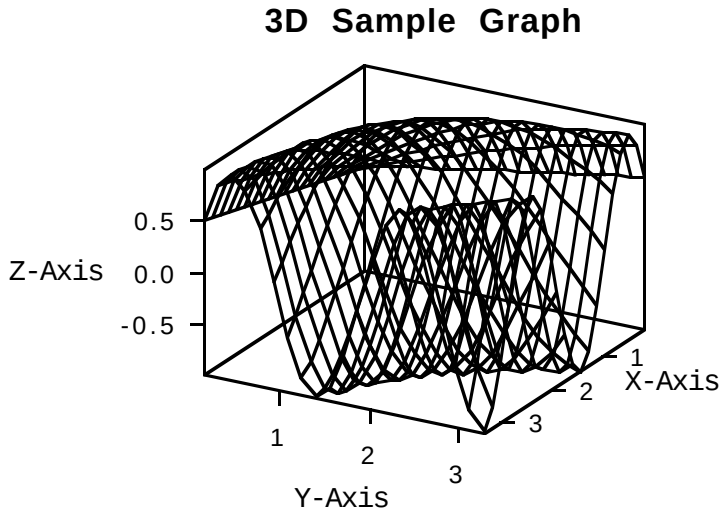
G (Graph): a 3-dimensional graph

OUTPUT

(The graph G is redrawn in wireframe mode)

EXAMPLES

```
// An example for: wireFrameGraph(G)

// Perform wireFrameGraph(G3) where G3 is the
// 3-dimensional graph generated by the function new3DGraph
// and containing a surface generated by the following formulae

for i = 1 to 21 do
  x[i] = i*<pi>/20;
  y[i] = x[i];
end for;
for i = 1 to 21 do
  for j = 1 to 21 do
    z[i,j] = sin(x[i]*y[j]);
  end for;
end for;
P3 = new3DDataPlot(" ",x,y,z);
G3 = new3DGraph("3D Sample Graph");
wireFrameGraph(G3);

// The following statement is for displaying the
// resultant plot P3 in  G3

addPlot(G3,P3);
// Result: (Display of resultant plot P3 in graph G3)
```



SEE ALSO

hiddenLineGraph, heightShading, lightSourceShading, setGraphShading

ALGORITHM AND COMMENTS

The same effect can be achieved using the function setGraphShading(•).

If graph G is not a 3-dimensional graph, an error message will be returned.

No effect is seen on curved plots.

■ zoomCamera

FUNCTION

zoomCamera (G, kx, ky)

PURPOSE

Change the zooming factor used to view a 3-dimensional graph

INPUT

G (Graph): a 3-dimensional graph

kx (Integer or Real Scalar): the zooming factor in the horizontal, i.e., x, direction

ky (Integer or Real Scalar): the zooming factor in the vertical, i.e., y, direction

OUTPUT

(The graph G is redrawn to the new zooming factors)

EXAMPLES

```
// An example for: zoomCamera(G,kx,ky)
//

// Perform zoomCamera(G3,kx,ky) where G3 is
// the 3-dimensional graph generated by the function
// new3DGraph and kx = 2, ky= 6

G3 = new3DGraph("3D Sample Graph");
kx = 2;
ky = 6;
zoomCamera(G3,kx,ky);
// Result:    NONE DISPLAYED
```

ALGORITHM AND COMMENTS

If graph G is not a 3-dimensional graph, or kx or ky is non-positive, then an error message will be returned.

CHAPTER 22
ANIMATION FUNCTIONS

■ goToFrame

FUNCTION

goToFrame(aMovie, frame)

PURPOSE

Position frame pointer to a particular frame.

INPUT

aMovie (Movie): movie of interest.

frame (Integer): number of frame to position to.

OUTPUT

None.

EXAMPLES

```
// Add frames to an existing movie.
lastFrame = numberOfFrames(aMovie);
goToFrame(aMovie, lastFrame);

// Make sure we have a graph.
if(SymbolGetType(aGraph) = <Untyped>) then
   aGraph = New2DGraph("My Graph");
else
   // Remove any plots.
   if getNumberOfPlots(aGraph) > 0 then
      for i = 1 to getNumberOfPlots(aGraph) do
         removePlot(aGraph, 0);
      end for;
   end if;
end if;

// Loop. This will create 11 frames.
for fac = 1 to 0 step -.1 do

   // Create a sinusoid.
   for j = 1 to 100 do
      x[j] = j/10;
      y[j] = sin(x[j]*fac);
   end for;
```

```
        // Create a plot of the sinusoid.
        aPlot = New2DDataPlot("My Plot", x, y);

        // Add it to our graph.
        addPlot(aGraph, aPlot);

        // Record a frame of the movie.
        recordFrame(aMovie, aGraph);

        // Remove the plot.
        removePlot(aGraph, 0);
    end for;

    // Rewind the movie to the first frame.
    rewindMovie(aMovie);
```

ALGORITHM AND COMMENTS

The first frame in a movie is frame 0.

■ newMovie

FUNCTION

theMovie = newMovie(type, width, height, dithered, spooled)

PURPOSE

Create a new movie symbol for use with the other movie functions.

INPUT

type (Integer): the type of the movie taken from the following:

```
<BitMap> = 1
<PixMap> = 2
<PICT>   = 3
```

width (Integer): number of pixels wide to make movie frame.

height (Integer): number of pixels high to make movie frame.

dithered (Boolean): true means to use dithering when creating the movie.

spooled (Boolean): true means to always spool the movie to disk when playing it. If this value is false then it only spools to disk when there isn't enough memory.

OUTPUT

A movie with no frames.

EXAMPLES

```
// Create a movie.
aMovie = newMovie(<PICT>, 300, 200, false, false);


// Make sure we have a graph.
if(SymbolGetType(aGraph) = <Untyped>) then
   aGraph = New2DGraph("My Graph");
else
   // Remove any plots.
   if getNumberOfPlots(aGraph) > 0 then
      for i = 1 to getNumberOfPlots(aGraph) do
         removePlot(aGraph, 0);
      end for;
   end if;
end if;

// Loop. This will create 11 frames.
for fac = 0 to 1 step .1 do

   // Create a sinusoid.
   for j = 1 to 100 do
      x[j] = j/10;
      y[j] = sin(x[j]*fac);
   end for;

   // Create a plot of the sinusoid.
   aPlot = New2DDataPlot("My Plot", x, y);

   // Add it to our graph.
   addPlot(aGraph, aPlot);

   // Record a frame of the movie.
   recordFrame(aMovie, aGraph);

   // Remove the plot.
   removePlot(aGraph, 0);
end for;

// Rewind the movie to the first frame.
rewindMovie(aMovie);
```

ALGORITHM AND COMMENTS

When choosing the type of movie that you wish to create keep in mind that a color (<PixMap>) movie takes 8 times the storage space as a black and white (<BitMap>) movie. A <PICT> movie takes very little storage space but is too slow to be suitable for very complex movies. It is however fine for simple 2D movies.

■ numberOfFrames

FUNCTION

frameCount = numberOfFrames(aMovie)

PURPOSE

Get number of frames in a movie.

INPUT

aMovie (Movie): movie of interest.

OUTPUT

Integer scalar containing the number of frames in the movie.

EXAMPLES

```
// Add frames to an existing movie.
lastFrame = numberOfFrames(aMovie);
goToFrame(aMovie, lastFrame);

// Make sure we have a graph.
if(SymbolGetType(aGraph) = <Untyped>) then
   aGraph = New2DGraph("My Graph");
else
   // Remove any plots.
   if getNumberOfPlots(aGraph) > 0 then
      for i = 1 to getNumberOfPlots(aGraph) do
         removePlot(aGraph, 0);
      end for;
   end if;
end if;

// Loop. This will create 11 frames.
for fac = .1 to 0 step -.1 do

   // Create a sinusoid.
   for j = 1 to 100 do
      x[j] = j/10;
      y[j] = sin(x[j]*fac);
   end for;

   // Create a plot of the sinusoid.
   aPlot = New2DDataPlot("My Plot", x, y);

   // Add it to our graph.
   addPlot(aGraph, aPlot);

   // Record a frame of the movie.
```

```
    recordFrame(aMovie, aGraph);

    // Remove the plot.
    removePlot(aGraph, 0);
end for;

// Rewind the movie to the first frame.
rewindMovie(aMovie);
```

ALGORITHM AND COMMENTS

The first frame in a movie is frame 0.

■ recordFrame

FUNCTION

recordFrame(aMovie, aGraph)

PURPOSE

Record a frame in a movie from a graph.

INPUT

aMovie (Movie): movie in which to record a frame.

aGraph (Graph): graph to create frame from. The size of the graph is set to the size of the movie frame as determined when the movie was created.

OUTPUT

None. A frame is added to aMovie.

EXAMPLES

```
// Create a movie.
aMovie = newMovie(<PICT>, 300, 200, false, false);


// Make sure we have a graph.
if(SymbolGetType(aGraph) = <Untyped>) then
    aGraph = New2DGraph("My Graph");
else
    // Remove any plots.
    if getNumberOfPlots(aGraph) > 0 then
        for i = 1 to getNumberOfPlots(aGraph) do
            removePlot(aGraph, 0);
        end for;
    end if;
end if;
```

```
// Loop. This will create 11 frames.
for fac = 0 to 1 step .1 do

   // Create a sinusoid.
   for j = 1 to 100 do
      x[j] = j/10;
      y[j] = sin(x[j]*fac);
   end for;

   // Create a plot of the sinusoid.
   aPlot = New2DDataPlot("My Plot", x, y);

   // Add it to our graph.
   addPlot(aGraph, aPlot);

   // Record a frame of the movie.
   recordFrame(aMovie, aGraph);

   // Remove the plot.
   removePlot(aGraph, 0);
end for;

// Rewind the movie to the first frame.
rewindMovie(aMovie);
```

ALGORITHM AND COMMENTS

The function recordFrame allows you the freedom to create a graph and take a snapshot of it. The graph size is temporarily set to the size of the movie for purposes of recording the frame.

■ rewindMovie

FUNCTION

rewindMovie(aMovie)

PURPOSE

Set a movie’s frame counter to the first frame.

INPUT

aMovie (Movie): movie to rewind.

OUTPUT

None.

EXAMPLES

```
// Create a movie.
aMovie = newMovie(<PICT>, 300, 200, false, false);


// Make sure we have a graph.
if(SymbolGetType(aGraph) = <Untyped>) then
   aGraph = New2DGraph("My Graph");
else
   // Remove any plots.
   if getNumberOfPlots(aGraph) > 0 then
      for i = 1 to getNumberOfPlots(aGraph) do
         removePlot(aGraph, 0);
      end for;
   end if;
end if;

// Loop. This will create 11 frames.
for fac = 0 to 1 step .1 do

   // Create a sinusoid.
   for j = 1 to 100 do
      x[j] = j/10;
      y[j] = sin(x[j]*fac);
   end for;

   // Create a plot of the sinusoid.
   aPlot = New2DDataPlot("My Plot", x, y);

   // Add it to our graph.
   addPlot(aGraph, aPlot);

   // Record a frame of the movie.
   recordFrame(aMovie, aGraph);

   // Remove the plot.
   removePlot(aGraph, 0);
end for;

// Rewind the movie to the first frame.
rewindMovie(aMovie);
```

ALGORITHM AND COMMENTS

The first frame in a movie is frame 0.

CHAPTER 23
UTILITY FUNCTIONS

■ convertUnits

FUNCTION

convertedSymbol = convertUnits(aSymbol, unitName1, unitName2)

PURPOSE

Convert the units of the entered numeric symbol from those of type unitName1 to those of type unitName2

INPUT

aSymbol (Scalar, Vector or Matrix of type Integer, Real or Complex): the symbol to be converted

unitName1 (String): the name of unit type to be converted

unitName2 (String): the name of the type of units that is converted to

OUTPUT

convertedSymbol (Scalar, Vector or Matrix of type Integer, Real or Complex): the new symbol (of the same type) expressed in the new system of units

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   convertUnits(aSymbol, unitName1, unitName2)

x = {0; 1; 2; 3};
y = convertUnits(x, "in", "m");
x1 = convertUnits(y, "m", "in");

//   Results:
//x:  4 rows
// 0
// 1
// 2
// 3
//
//   x1:  4 rows
//                  0
//                  1
//                  2
//                  3
//
//   y:  4 rows
```

```
//                    0
//               0.0254
//               0.0508
//               0.0762
```

ALGORITHM AND COMMENTS

The unit conversion is based on the conversions defined in the file HiQ•Conversions. A number of conversions are already in the file but you can add your favorite conversions to the file and then be able to access them using the convertUnits function. So if you would like to express velocity in terms of furlongs per fortnight it would be quite simple to add the conversion. The file format may seem a bit confusing at first but it is actually quite straight forward. The following is an example of what this file looks like:

```
 1 //-----------------------------------------
 2 Type:Distance
 3    Length
 4 Name:Meters
 5    m
 6 Conv: * 1.0
 7 Name:Mils
 8 Conv:* 0.000025400
 9 Name:Inches
10    in
11 Conv:* 0.0254
```

The above is part of the conversion table for distance or length. The numbers to the left are not part of the file and will be used in the following to describe each line.

1) This is a comment. Anything following // to the end of the line is a comment.

2) Type designator. This designates a type of unit like distance, time, velocity, etc. Everything following a type command until another type command is encountered is part of the same group. There must only be one type designator for each type of unit. The second part of the name is the unit type name.

3) Another name for this type of unit. Up to four names may be specified with one name per line.

4) Name of unit. In this case meters.

5) Another name for meters. In this case the abbreviation m. Up to four names may be specified with one name per line.

6) The conversion. This is what you would have to do to this unit to convert it to the base unit type. The base unit type has a conversion of * 1. In this case the base unit type is meters. The conversion must follow the unit name.

7) Another unit name.

8) The conversion from Mils to the base unit of meters.

9) Another unit name.

10) Another name for inches.

11) The conversion from inches to meters (the base unit).

Some unit conversions are a little more complicated than just a multiplication. One example is temperature conversions. The following illustrates how temperature conversions are done:

```
//-------------------------------------------------
Type:Temperature
Name:Kelvin
       K
Conv: * 1.0
Name:Fahrenheit
       F
Conv:+ 459.67
       / 1.8
Name:Celsius
       C
       Centigrade
Conv:+ 273.15
Name:Rankine
       R
Conv:/ 1.8
```

In this case, the base temperature unit is chosen to be degrees Kelvin. So if you want to convert from degrees Fahrenheit to degrees Kelvin you would add 459.67 and then divide by 1.8. Note that the operations are done in the order that they are shown with no regard to normal precedence. The valid operators are + - * / exp and log. This provides the ability to do virtually any kind of unit conversion imaginable. You can even convert to and from units of dB.

■ deleteSymbol

FUNCTION

deleteSymbol(aSymbol)

PURPOSE

Mark the specified symbol to be marked for deletion when the script finishes execution.

INPUT

aSymbol (Any symbol): the symbol to be deleted.

OUTPUT

None

EXAMPLES

```
// Delete the symbol x.
```

```
x = 1;
deleteSymbol(x);
```

ALGORITHM AND COMMENTS

The deletion is accomplished by changing the type of a symbol to untyped. If the symbol's type is subsequently changed to another type the symbol will not be deleted.

■ error

FUNCTION

error(errorString)

PURPOSE

Display an error message in a modal dialog box and cause the script to terminate

INPUT

errorString (String): the error message to be displayed

OUTPUT

(A dialog box on the screen displays the error message and the script will be terminated)

EXAMPLES

```
//        Perform error(S) where S="Error Message"
local S;
S = "Example Error Message Display";
error(S);
// Result: (Display of resultant dialog box)
```

■ getNumber

FUNCTION

result = getNumber("prompt", "default")

PURPOSE

Retrieve a numerical input value from an entered string. An optional default response may be specified which may be edited.

INPUT

"prompt" (String): the message prompting the user to enter a data value

"default" (String): the string that is converted to a number for the default case

OUTPUT

result (Integer, Real, or Complex Scalar): the numerical value converted from the entered string

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   getNumber("prompt", "default")

a = getNumber("Enter number of significant digits.  Default is 6","6");

//   Results (for default input):    a:     6
```

Both parameters are string expressions, so the following script would be equivalent:

```
question = "Enter number of significant digits.  Default is 6";
first_guess = "6";
a = getNumber(question, first_guess);
```

Even if you don't wish to display a default value, you must provide an empty string as the second parameter.

```
a = getNumber("Enter number of significant digits","");
```

■ getString

FUNCTION

result = getString("prompt", "default")

PURPOSE

Retrieve a string from an entered one. An optional default response may be specified which may be edited.

INPUT

"prompt" (String): the message prompting the user to enter a message

"default" (String): the string that is retrieved for the default case

OUTPUT

result (String): the string entered by the user

EXAMPLES

```
//   HiQ-Script Example for the utility function
//   getString("prompt", "default")

b = getString("Enter the axis name.  Default is x-axis",
"x-axis");
```

```
//   Results (for default input):   b:  x-axis
//
```

The symbol s is set to whatever is in the edit box when OK is clicked. Editing capabilities are currently limited to:

1) navigation by arrow keys and mouse

2) text deletion with the delete key

3) text insertion by typing: text will appear at the insertion point and will replace text that is highlighted.

■ getSymbol

FUNCTION

y = getSymbol (aSymbolName)

PURPOSE

To return a symbol by its name

INPUT

aSymbolName (String): the name of the symbol to be returned

OUTPUT

y (Symbol): the requested symbol

EXAMPLES

```
//   HiQ-Script Example for the utility function
//   getSymbol(aSymbolName)

b   = "x-axis";
b1 = getSymbol("b");

//   Results (for default input):
//   b:  x-axis
//   b1:  x-axis
```

ALGORITHM AND COMMENTS

This function is useful for getting a copy of data in a symbol of a particular name. This doesn't return a reference to the symbol so the original symbol won't be modified. In the following example x will equal 1 and y will equal 3 after the script executes.

```
x = 1;
y = GetSymbol("x");
```

```
y = y + 2;
```

Currently there isn't a way to modify x directly without specifying it as a symbol.

■ isProjectSymbol

FUNCTION

status = isProjectSymbol(aSymbolName)

PURPOSE

To determine if a symbol of the given name is in the project symbol table

INPUT

aSymbolName (String): the name of the symbol to be searched for

OUTPUT

status (Integer Scalar): a boolean with the value <true> if a symbol of the given name exists in the project symbol table

EXAMPLES

```
//   HiQ-Script Example for the utility function
//   status = isProjectSymbol(aSymbolName)
A   = {1, 2;
       0, 0};
status_true  = isProjectSymbol("A");
status_false = isProjectSymbol("NO_SUCH_SYMBOL");

//   Results :
//   A:  2 rows, 2 columns
//   1 2
//   0 0
//   status_false:  0
//   status_true:  1
```

ALGORITHM AND COMMENTS

This function is useful in conjunction with SymbolRename or GetSymbol to find out if a given symbol name exists or not.

■ merge

FUNCTION

w = merge(u,v)

PURPOSE

Merge two vectors of dimensions m and n with each already in non-descending order into a single (m+n)-dimensional vector such that its elements are in non-descending order

INPUT

u (Real Vector): the first sorted vector of dimension m

v (Real Vector): the second sorted vector of dimension n

OUTPUT

w (Real Vector) : the (m+n)-dimensional vector containing the elements of input vectors u and v arranged in non-descending order

EXAMPLE

```
//   HiQ-Script Example for the utility function
//  merge(u,v) : to  merge two sorted vectors (both in
//  non-descending ordes) into a single vectors with elements
//  in non-descending order

//  The vector u is:
u = {-22; -7; 0; 3; 10; 12};
//  The vector v is:
v = {-9; -4; -3; 1; 2; 3; 5; 7; 8};
w = merge(u,v);

//  Result :
//   w:  15 rows
//                      -22
//                       -9
//                       -7
//                       -4
//                       -3
//                        0
//                        1
//                        2
//                        3
//                        3
//                        5
//                        7
//                        8
//                       10
//                       12
```

ALGORITHM AND COMMENTS

If the elements of any of the two input vectors, u and v, is not arranged in non-descending order, the user should run the function sort to rearrange the vector in the desired order before using this function. Otherwise, the result from merge will be incorrect.

■ message

FUNCTION

message("message")

PURPOSE

Display a message to the user in a modal dialog box which must be clicked away before the script continues

INPUT

"message" (String): the message to be displayed

OUTPUT

None (A dialog box on the screen displays the message)

EXAMPLES

Either of the two following examples brings up the same message:

```
//   HiQ-Script Example for the utility function
//  message("message")

//   string of the message
s = "Hello";
message(s);
//  Result :
//   Display a message
```

■ numberToString

FUNCTION

aString = numberToString(a , d, format)

PURPOSE

Convert a number to a string form with specified format

INPUT

a (Integer, Real or Complex Scalar): the number whose value is converted to a string form

d (Integer Scalar): the number of decimal places in the resulting string form

format (Integer Scalar): the numerical type of the resulting string where format = 0 for real number fixed point format, format = 1 for real number scientific format, format = 2 for complex number fixed point format, format = 3 for complex number scientific format, and format = 4 for integer format

OUTPUT

aString (string): the string converted from the input number

EXAMPLES

```
//        Examples for:  numberToString(a , d, fmt)

//        Perform numberToString(a,d,f) where a = 3.21
//        d = 1 and f = 1 (for scientific format)

local a,d,f,b,e,g;
a = 3.21;
d = 1;
f = <scientific>;
aString1 = numberToString(a,d,f);

// Result:
//             aString1:  3.2e+00

//        Perform numberToString(b,e,g) where b = (3, -3.333)
//        e = 3 and g = 2 (for fixed point format)

b =  (3, -3.333);
e = 3;
g = 2;
aString2 = numberToString(b,e,g);

// Result:
//             aString2:  3.000 + -3.333 i
```

ALGORITHM AND COMMENTS

For non-integer format conversion, a maximum of 18 decimal places is output for any input d > 18

For integer format conversion, the input d is ignored and the result is a string of length no larger than 5

■ sort

FUNCTION

w = sort(v)

PURPOSE

Arrange the elements of a given n-dimensional vector in non-descending order

INPUT

v (Real Vector): the original n-dimensional vector

OUTPUT

w (Real Vector): the n-dimensional vector containing the elements of input vector v in non-descending order

EXAMPLE

```
//   HiQ-Script Example for the utility function
//  sort(v) : to arrange the elements of vector
//  v in non-descending order

//  The vector v is:
v = {8; 3; 5; 12; 10; -4; -22; 7; 3; 0; -3; 2; 1; -7; -9};

w =sort(v);

//  Result :
//   w:  15 rows
//                      -22
//                       -9
//                       -7
//                       -4
//                       -3
//                        0
//                        1
//                        2
//                        3
//                        3
//                        5
//                        7
//                        8
//                       10
//                       12
```

ALGORITHM AND COMMENTS

The sort function implements the well-known Quicksort algorithm with an auxiliary stack of size 50 for keeping indices of the intermediate unsorted sub-vectors.

If the given vector requires more than 50 intermediate stacks during the Quicksort, an error message will be returned.

REFERENCE

Knuth, D.E.: *Sorting and Searching : The Art of Computer Programming*, vol. 3, Addison-Wesley Publishing, 1973, p. 114.

■ stringToNumber

FUNCTION

aNumber = stringToNumber("aNumber")

PURPOSE

Converts an entered string to a number

INPUT

"aNumber" (String): a number in string form ("500", "1.23", "3+4i"). Contents of the string must be a legal number to prevent an error return. The input may be a string expression.

OUTPUT

aNumber (Integer, Real, or Complex Scalar): a number whose value was given in the entered string

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  aNumber = stringToNumber("aNumber")

//  two strings that contain number
//
s1 = " 1.2e4 ";
s2 = " 1.2e-4";

// actual numbers converted from these strings
//
x1 = stringToNumber(s1);
x2 = stringToNumber(s2);

//  Results :
//  s1:   1.2e4
//  s2:   1.2e-4
//  x1:   12000
//  x2:   0.00012
```

■ symbolGetCols

FUNCTION

numOfCols = symbolGetCols(aSymbol)

PURPOSE

To return the number of columns in a specified symbol

INPUT

aSymbol (Symbol): the numeric symbol whose number of columns is being determined

OUTPUT

numOfCols (Integer Scalar): the number of columns contained in the specified symbol

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  numOfCols = symbolGetCols(aSymbol)
//   that returns the number of columns in a specified symbol

//  2 x 4 matrix A
//
A = { 1, 2, 3, 4;
      5, 6, 7, 8} ;
n = symbolGetCols(A);

//  Results :
//  A:  2 rows, 4 columns
//  1 2 3 4
//  5 6 7 8
//
//  n:  4
```

ALGORITHM AND COMMENTS

Valid only for symbols of type vector or matrix.

■ symbolGetLock

FUNCTION

lockStatus = symbolGetLock(aSymbol, lockID)

PURPOSE

Returns the status of the lock of a specified symbol

INPUT

aSymbol (Symbol): the symbol whose lock status is queried

lockID (Integer Scalar): ID of the lock to query from among the following language constants:

```
<NameLock>   =0
<DataLock>   =1
<TypeLock>   =2
<DimLock>    =3
<DeleteLock> =4
```

```
<MasterLock> =5
```

OUTPUT

lockStatus (Integer Scalar): returns a boolean; true indicates that the lock is set; false indicates that it is not set

EXAMPLE

```
//   HiQ-Script Example for the utility function
//  lockStatus = symbolGetLock(aSymbol, lockID)
//   that returns the status of the lock of a specified symbol

//  vector u
//
u = { 1, 2, 3, 4,  5, 6, 7, 8} ;

//   ID of the lock to query
lockID = <DimLock>;

status = symbolGetLock(u, lockID);

//  Results :
//  lockID:  3
//  status:  0
//  u:  8 columns
//  1 2 3 4 5 6 7 8
```

ALGORITHM AND COMMENTS

<NameLock> prevents the symbol from being renamed

<DataLock> prevents the data portion of the symbol from being modified

<TypeLock> prevents the type from changing. This is useful for coercing a real value to an integer value in an assignment. Types of symbols cannot be changed unless the symbol is untyped or it is of numeric type.

<DimLock> prevents the dimension of vectors and matrices from being changed

<DeleteLock> prevents symbols from being deleted

<MasterLock> prevents the other locks from being changed

■ symbolGetMatrixDim

FUNCTION

[m, n] = symbolGetMatrixDim(aSymbol)

PURPOSE

To return the number of rows and columns in a specified matrix symbol

INPUT

aSymbol (Integer, Real or Complex Matrix): the matrix symbol whose number of rows and columns is being returned

OUTPUT

m (Integer Scalar): the number of rows in the specified symbol

n (Integer Scalar): the number of columns in the specified symbol

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  [m, n] = symbolGetMatrixDim(aSymbol)
//

//  matrix A
//
A = { 1, 2, 3, 4;
      5, 6, 7, 8} ;

//  its dimensions
//
[m, n]  = symbolGetMatrixDim(A);

//  Results :
//  A:  2 rows, 4 columns
//  1 2 3 4
//  5 6 7 8
//
//  m:  2
//  n:  4
```

ALGORITHM AND COMMENTS

Valid only for symbols of type matrix.

■ symbolGetRows

FUNCTION

numOfRows = symbolGetRows(aSymbol)

PURPOSE

Return the number of rows in the specified numeric symbol

INPUT

aSymbol (Symbol): the specified numeric symbol

OUTPUT

numOfRows (Integer Scalar): the number of rows contained in the specified symbol

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  numOfCols = symbolGetCols(aSymbol)
//   that returns the number of rows in a specified symbol

//  2 x 4 matrix A
//
A = { 1, 2, 3, 4;
      5, 6, 7, 8} ;

m = symbolGetRows(A);
//  Results :
//  A:  2 rows, 4 columns
//  1 2 3 4
//  5 6 7 8
//
//  m:  2
```

ALGORITHM AND COMMENTS

Valid only for symbols of type vector or matrix.

■ symbolGetType

FUNCTION

symbolGetType(aSymbol, type)

PURPOSE

To return the type of a specified symbol

INPUT

aSymbol (Symbol): the symbol whose type is to be determined

OUTPUT

type (Integer Scalar): The valid type language constants are as follows:

```
<Untyped>          Untyped symbol
<IntegerScalar>    Integer scalar.
<RealScalar>       Real scalar.
<ComplexScalar>    Complex scalar.
<IntegerVector>    Integer vector.
<RealVector>       Real vector.
```

```
<ComplexVector>     Complex vector.
<IntegerMatrix>     Integer matrix.
<RealMatrix>        Real matrix.
<ComplexMatrix>     Complex matrix.
<HiQFunction>       Compiled HiQ-Script function
<HiQBIFunction>     Built-in function
<String>            String symbol
<Plot>              Plot symbol
<Graph>             Graph symbol
<Movie>             Movie symbol
<HiQScript>         HiQ-Script symbol
<BVPODESolver>      Boundary Value ODE problem solver
<ExprEvaluator>     Expression Evaluator
<Integrator>        Integrator problem solver
<IVPODESolver>      Initial Value ODE problem solver
<NonLinSysSolver>   Nonlinear Systems problem solver
<Optimizer>         Optimizer problem solver
<PolyRootSolver>    Polynomial Root solver
<IntegralEqnSolver> Integral Equation problem solver
<GeneralRootSolver> General Root solver
<StatSolver>        Statistical Analyzer problem solver
<Picture>           Picture symbol
```

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  type = symbolGetType(aSymbol)
//   that returns the type of a specified symbol

//  2 x 4 matrix A
//
A = { 1, 2, 3, 4;
      5, 6, 7, 8} ;
type = symbolGetType(A);

//  Results :
//  A:  2 rows, 4 columns
//  1 2 3 4
//  5 6 7 8
//
//  type:  7
```

■ symbolGetVectorDim

FUNCTION

theDimension = symbolGetVectorDim(aSymbol)

PURPOSE

Get the number of elements in a vector.

INPUT

aSymbol: a vector Symbol.

OUTPUT

Number of elements (Integer).

EXAMPLES

```
// Get number of elements in x.
n = symbolGetVectorDim(x);
```

ALGORITHM AND COMMENTS

Valid only for symbols of type vector and string.

■ symbolLock

FUNCTION

symbolLock(aSymbol, lockID)

PURPOSE

Set the specified symbol lock to true (locked), i.e., lock the specified symbol attribute given by lockID

INPUT

aSymbol (Symbol): the symbol to be locked.

lockID (Integer Scalar): ID of the lock to be set from the following language constants:

```
<NameLock>=0
<DataLock>=1
<TypeLock>=2
<DimLock>=3
<DeleteLock>=4
<MasterLock>=5
```


OUTPUT

None (The indicated attribute of aSymbol is locked)

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  symbolLock(aSymbol, lockID)
//   that locks the lockID symbol attribute

u = { 1, 2, 3, 4, 5, 6, 7, 8} ;
lockID = <DataLock>;
symbolLock(u, lockID);
u[1] = 10;

//  Results :
//   None (output)
//   The error message appears since
//   the data of u is locked
```

ALGORITHM AND COMMENTS

<NameLock> prevents the symbol from being renamed

<DataLock> prevents the data portion of the symbol from being modified

<TypeLock> prevents the type from changing. This is useful for coercing a real value to an integer value in an assignment. Types of symbols cannot be changed unless the symbol is untyped or it is of numeric type.

<DimLock> prevents the dimension of vectors and matrices from being changed

<DeleteLock> prevents symbols from being deleted

<MasterLock> prevents the other locks from being changed

■ symbolRename

FUNCTION

symbolRename(aSymbol, newName)

PURPOSE

To change the name of an existing project Symbol.

INPUT

aSymbol (Symbol): the symbol to be renamed

newName (String): the new name of the symbol

OUTPUT

None (the specified symbol has a new name)

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  symbolRename(aSymbol, newName)
//   that renames a symbol with a new name

x_old = 0.5;
   symbolRename(x_old, "x_new");

//  Results :
//   x_new:     0.5
```

ALGORITHM AND COMMENTS

The symbol being renamed cannot be local. If a symbol of the same name as that which you wish to rename to already exists in the project symbol table then the function will fail and the script will stop execution. If you access the symbol that was renamed by its old name after the rename, it is an untyped symbol of the same name in the project. If the new symbol name is already taken by an untyped symbol then the rename will succeed.

■ symbolSave

FUNCTION

symbolSave(aSymbol)

PURPOSE

Save a particular symbol or all symbols to the worksheet document.

INPUT

aSymbol (Any Symbol): Symbol to be saved or <AllSymbols>.

OUTPUT

None

EXAMPLES

```
// Save x.
symbolSave(x);

// Save all symbols.
symbolSave(<AllSymbols>);
```

ALGORITHM AND COMMENTS

All symbols may be saved by specifying <AllSymbols>.

■ symbolSetCols

FUNCTION

symbolSetCols(aSymbol, numOfCols)

PURPOSE

To set the number of columns associated with a numeric symbol

INPUT

aSymbol (Symbol): numeric symbol whose number of columns is being set

numOfCols (Integer Scalar): the new number of columns for the specified symbol

OUTPUT

None (The specified symbol has a new number of columns)

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  symbolSetCols(aSymbol, numOfCols)

//   create two equal matrices
//
A = { 1, 2, 3, 4, 5;
      6, 7, 8, 9, 10 };
B = A;

//   set the number of rows of A
n = 2;
symbolSetCols(A, n);

//  Results :
//  A:  3 rows, 5 columns
//  1 2
//  6 7
//
//  B:  2 rows, 5 columns
//  1 2 3 4 5
//  6 7 8 9 10
//
//  m: 3
```

ALGORITHM AND COMMENTS

Valid only for symbols of type vector or matrix.

■ symbolSetMatrixDim

FUNCTION

symbolSetMatrixDim(aSymbol, m, n)

PURPOSE

To set the number of rows and columns associated with a specified matrix symbol

INPUT

aSymbol (Integer, Real or Complex Matrix): the matrix symbol whose number of rows and columns are being set

m (Integer Scalar): the new number of rows for the matrix

n (Integer Scalar): the new number of columns for the matrix

OUTPUT

None (the specified matrix has m rows and n columns)

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  symbolSetMatrixDim(aSymbol, m, n)

//   create two equal matrices
//
A = { 1, 2, 3, 4, 5;
      6, 7, 8, 9, 10 };
B = A;

//   set the dimensions of A
m = 2;
n  = 4;
symbolSetMatrixDim(A, m, n);


//  Results :
//  A:  2 rows, 4 columns
//  1 2 3 4
//  6 7 8 9
//
//  B:  2 rows, 5 columns
//  1 2 3 4 5
//  6 7 8 9 10
```

```
//
//  m:  2
//  n:  4
```

ALGORITHM AND COMMENTS

Valid only for symbols of type matrix.

■ symbolSetRows

FUNCTION

symbolSetRows(aSymbol, numOfRows)

PURPOSE

To set the number of rows associated with a numeric symbol

INPUT

aSymbol (Symbol): the symbol whose row dimension is being set

numOfRows (Integer Scalar): the new number of rows for the symbol

OUTPUT

None (The specified symbol has a new number of rows)

EXAMPLE

```
//   HiQ-Script Example for the utility function
//  symbolSetRows(aSymbol, numOfRows)

//   create two equal matrices
//
A = { 1, 2, 3, 4, 5;
      6, 7, 8, 9, 10 };
B = A;

//   set the number of rows of A
m = 3;
symbolSetRows(A, m);


//  Results :
//  A:  3 rows, 5 columns
//  1 2 3 4 5
//  6 7 8 9 10
//  0 0 0 0 0
//
//  B:  2 rows, 5 columns
//  1 2 3 4 5
```

```
//  6 7 8 9 10
//
//  m: 3
```

ALGORITHM AND COMMENTS

Valid only for symbols of type vector or matrix

■ symbolSetType

FUNCTION

symbolSetType(aSymbol, aType)

PURPOSE

To change the type of a numeric symbol

INPUT

aSymbol (Numeric Symbol): the symbol whose type is to be changed

aType (Integer Scalar): Valid types are among the following language constants:

```
<IntegerScalar>     Integer scala
<RealScalar>        Real scalar
<ComplexScalar>     Complex scalar
<IntegerVector>     Integer vector
<RealVector>        Real vector
<ComplexVector>     Complex vector
<IntegerMatrix>     Integer matrix
<RealMatrix>        Real matrix
<ComplexMatrix>     Complex matrix
```

OUTPUT

None (the type of the specified numeric symbol is set)

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  symbolSetType(aSymbol, aType)

//   create two equal real matrices
//
symbolSetType(A,<RealMatrix>);
A = { 1.1,  2.2,   3.3,   4.4;
      5.5,  6.6,   7.7,   8.8 };
B = A;
```

```
//   set the type of A to <IntegerMatrix>
symbolSetType(A, <IntegerMatrix>);


//  Results :
//  A:  2 rows, 4 columns
//  1 2 3 4
//  5 6 7 8
//
//  B:  2 rows, 4 columns
//              1.1         2.2         3.3      4.4
//              5.5         6.6         7.7       8.8
```

ALGORITHM AND COMMENTS

When converting from real or complex data to integer data no attempt is made to deal with real numbers greater than the largest integer. The result of this is undefined.

When converting from a higher type to a lower type some precision or data may be lost. When converting from a higher number of dimensions to a lower number of dimensions the first dimension is taken. So if you convert from a matrix to a vector the first column is taken. When you convert from a vector to a scalar the first cell is taken. When you convert from a matrix to a scalar the first cell is taken. No warning messages are given for any of these conversions.

When converting to a higher dimension or data type no data or precision is lost.

Types are automatically converted when you do assignments except when the type is locked.

Refer to the documentation of HiQ-Script for more information.

■ symbolSetVectorDim

FUNCTION

symbolSetVectorDim(aSymbol, m)

PURPOSE

To set the number of elements in a vector without affecting the orientation of the vector

INPUT

aSymbol (Integer, Real or Complex Vector): the vector symbol whose dimension is being set

m (Integer Scalar): the new number of elements for the specified vector

OUTPUT

None (the specified vector has m rows or columns)

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  symbolSetVectorDim(aSymbol, m)

//   create two equal vectors v
//
v = { 1, 2, 3, 4, 5,  6, 7, 8 };
v1 = v;

//   set the dimensions of v1
m = 4;
symbolSetVectorDim(v1, m);

//  Results :
//  m:  4
//  v:  8 columns
//  1 2 3 4 5 6 7 8
//
//  v1:  4 columns
//  1 2 3 4
```

ALGORITHM AND COMMENTS

Valid only for symbols of type vector

■ symbolUnlock

FUNCTION

symbolUnlock(aSymbol, lockID)

PURPOSE

Set the specified symbol lock to false (unlocked), i.e., unlock the specified symbol attribute given by lockID

INPUT

aSymbol (Symbol): the symbol to be unlocked

lockID (Integer Scalar): ID of the lock to be set from the following language constants:

```
<NameLock>    = 0
<DataLock>    = 1
<TypeLock>    = 2
<DimLock>     = 3
<DeleteLock>> = 4
<MasterLock>> = 5
```

OUTPUT

None (The indicated attribute of aSymbol is unlocked)

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  symbolUnlock(aSymbol, lockID)
//   that unlocks the lockID symbol attribute

p = { 1, 2, 3, 4, 5, 6, 7, 8} ;
lockID = <DataLock>;

// Data type of p is locked
//
symbolLock(p, lockID);

// Data type of p is unlocked
//
symbolUnlock(p, lockID);
p[1] = 10;

//  Results :
//  lockID:  1
//  p:  8 columns
//  10 2 3 4 5 6 7 8
```

ALGORITHM AND COMMENTS

<NameLock> prevents the symbol from being renamed

<DataLock> prevents the data portion of the symbol from being modified

<TypeLock> prevents the type from changing. This is useful for coercing a real value to an integer value in an assignment. Types of symbols cannot be changed unless the symbol is untyped or it is of numeric type.

<DimLock> prevents the dimension of vectors and matrices from being changed

<DeleteLock> prevents symbols from being deleted

<MasterLock> prevents the other locks from being changed

■ timer

FUNCTION

y = timer(s)

PURPOSE

To perform various CPU timing functions

INPUT

s (Integer): the type of CPU timing function to be performed where s = 0 for setting timer to zero second, s = 1 for starting the CPU timing process, s = 2 for returning the current cumulative CPU time when stop the timer, and s = 3 for rerunning the current cumulative CPU time without stopping the timer

OUTPUT

y (Real Scalar) : the returned value (in seconds)

EXAMPLES

```
//       An example for: timer(s)

//       Perform the timer(s) for
//           s = 0 : to zero the timer
//           s = 1 : to start the timer
//           s = 3 : to stop the timer
//       with a delay of 2 seconds using the
//       function wait

Time0 = timer(0);
Time1 = timer(1);
wait(2);
Time3 = timer(3);

//  Results:
//     Time0:           0
//     Time1:           0
//     Time3:           2
```

■ updateDisplay

FUNCTION

updateDisplay(aSymbol)

PURPOSE

Update the display representation(s) of a particular symbol or all symbols.

INPUT

aSymbol (Any Symbol): Symbol to be updated or <AllSymbols>.

OUTPUT

None

EXAMPLES

```
// Update display for x.
updateDisplay(x);

// Update display for all symbols.
updateDisplay(<AllSymbols>);
```

ALGORITHM AND COMMENTS

All symbols may be updated by specifying <AllSymbols>. This function can be useful for debugging scripts or for creating data animations on a worksheet.

■ useCoprocessor

FUNCTION

useCoprocessor(oneOrZero)

PURPOSE

Sets a flag to determine whether to use hardware or HiQ software version of those functions that have a hardware version (sin(x) for example). The hardware version is faster, but sacrifices the guaranteed accuracy of the software version.

INPUT

oneOrZero (Integer Scalar): An integer indicating which state:

1 use hardware version of function

0 use software version

OUTPUT

None (an internal flag is set)

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  useCoprocessor(oneOrZero)
//
theta = 1000000.01 ;

//   use software version of the function sin()
useCoprocessor(0);
x1 = sin(theta);

//   use hardware  version of the function sin()
useCoprocessor(1);
x2 = sin(theta);
```

```
//   to switch back to default mode
useCoprocessor(0);
//  Results :
//  theta:                1000000.01
//  x1:         -0.340608637491212
//  x2:         -0.340608637491211
```

■ wait

FUNCTION

wait(n)

PURPOSE

Delay the execution of a script by specified number of CPUseconds

INPUT

n (Integer): the desired delay time

OUTPUT

(The execution of script statements will be delayed for the specified number of seconds)

EXAMPLES

```
//       An example for:  wait(n)

//        Perform wait(n) where n = 5 using
//        timer(0), timer(1) and timer(2) to
//        obtain the returning time after the delay

local n;
n = 5;
timer(0);
timer(1);
wait(n);
timeWait = timer(2);

//  Result:
//     timeWait:         5
```

■ warning

FUNCTION

result = warning("warningMessage");

PURPOSE

Display an alert containing a warning message and retrieve the contents of a flag that indicates whether the user clicked the OK or Cancel button

INPUT

"warningMessage" (String): An entered expression warning the user of some state or condition that requires a response

OUTPUT

result (Integer Scalar): A modal dialog is displayed on the screen with the warning message; the value of result is:

```
result:1  OK
       0  Cancel
```

EXAMPLES

```
//   HiQ-Script Example for the utility function
//  result = warning("warningMessage")
//

result = warning("The next phase will take over an hour");

//  Result :
//  result:  1 (or 0) depending on the user's action
```

If the user clicks OK, result = 1; if the user clicks Cancel, result = 0.

CHAPTER 24
INPUT/OUTPUT FUNCTIONS

■ close

FUNCTION

close(FileID)

PURPOSE

Closes the specified file.

INPUT

FileID (Integer): the ID of a file opened with the open function.

OUTPUT

None.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

Files are automatically closed when a script finishes running. Use the open function to open a file.

■ flush

FUNCTION

flush(FileID)

PURPOSE

Flushes file buffers to disk.

INPUT

FileID (Integer): the ID of a file opened with the open function. A value of <FlushAllFiles> may be used to flush all open files.

OUTPUT

None.

EXAMPLES

Refer to the open function for examples.

■ IEEEInt8ToInteger

FUNCTION

anInteger = IEEEInt8ToInteger(ByteString)

PURPOSE

Converts the input bytes to an integer.

INPUT

ByteString (String): the byte string representing an 8-bit IEEE integer. The first byte in the string is used.

OUTPUT

An integer.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Read function when reading binary data.

■ IEEEInt16ToInteger

FUNCTION

anInteger = IEEEInt16ToInteger(ByteString)

PURPOSE

Converts the input bytes to an integer.

INPUT

ByteString (String): the byte string representing a 16-bit IEEE integer. The first two bytes in the string are used.

OUTPUT

An integer.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Read function when reading binary data.

■ IEEEInt32ToInteger

FUNCTION

anInteger = IEEEInt32ToInteger(ByteString)

PURPOSE

Converts the input bytes to an integer.

INPUT

ByteString (String): the byte string representing a 32-bit IEEE integer. The first 4 bytes in the string are used.

OUTPUT

An integer.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Read function when reading binary data.

■ IEEEReal32ToReal

FUNCTION

aRealScalar = IEEEReal32ToReal(ByteString)

PURPOSE

Converts the input bytes to a real scalar.

INPUT

ByteString (String): the byte string representing a 32-bit IEEE real. The first 4 bytes in the string are used.

OUTPUT

A real scalar.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Read function when reading binary data.

■ IEEEReal64ToReal

FUNCTION

aRealScalar = IEEEReal64ToReal(ByteString)

PURPOSE

Converts the input bytes to a real scalar.

INPUT

ByteString (String): the byte string representing a 64-bit IEEE real. The first 8 bytes in the string are used.

OUTPUT

A real scalar.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Read function when reading binary data.

■ integerToIEEEInt8

FUNCTION

ByteString = integerToIEEEInt8(anIntegerScalar)

PURPOSE

Converts the real scalar into a byte string.

INPUT

anIntegerScalar (Integer): the scalar to be converted. The scalar is converted to an IEEE 8-bit integer. Integers > than 127 or < -128 will be converted in an indeterminate manner.

OUTPUT

A byte string with a length of 1 byte.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Write function when writing binary data.

■ integerToIEEEInt16

FUNCTION

ByteString = integerToIEEEInt16(anIntegerScalar)

PURPOSE

Converts the real scalar into a byte string.

INPUT

anIntegerScalar (Integer): the scalar to be converted. The scalar is converted to an IEEE 16-bit integer. Integers > than 32767 or < -32768 will be converted in an indeterminate manner.

OUTPUT

A byte string with a length of 2 bytes.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Write function when writing binary data.

■ integerToIEEEInt32

FUNCTION

ByteString = integerToIEEEInt32(anIntegerScalar)

PURPOSE

Converts the real scalar into a byte string.

INPUT

anIntegerScalar (Integer): the scalar to be converted. The scalar is converted to an IEEE 32-bit integer.

OUTPUT

A byte string with a length of 4 bytes.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Write function when writing binary data.

■ isEOF

FUNCTION

HasFileBeenSet = isEOF(FileID)

PURPOSE

Checks to see if the EOF flag has been set for the specified file.

INPUT

FileID (Integer): the ID of a file opened with the open function.

OUTPUT

A non-zero value indicates that the EOF has been reached.

EXAMPLES

Refer to the open function for examples.

■ macReal80ToReal

FUNCTION

aRealScalar = macReal80ToReal(ByteString)

PURPOSE

Converts the input bytes to a real scalar.

INPUT

ByteString (String): the byte string representing a 80-bit IEEE real. The first 10 bytes in the string are used.

OUTPUT

A real scalar.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Read function when reading binary data.

This function converts the 80-bit real to 96 bits. Since the Macintosh doesn't use 16 bits of the data no precision is lost when saved in this format. This provides for more efficient use of disk space.

■ macReal96ToReal

FUNCTION

aRealScalar = macReal96ToReal(ByteString)

PURPOSE

Converts the input bytes to a real scalar.

INPUT

ByteString (String): the byte string representing a 96-bit IEEE real. The first 12 bytes in the string are used.

OUTPUT

A real scalar.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the Read function when reading binary data.

Since the Macintosh doesn't use 16 bits of the data we recommend that you use the function RealToMacReal80 for more efficient use of disk space without loss of precision when saving data.

■ open

FUNCTION

FileID = open(Filename, Mode)

PURPOSE

Opens a file using a specified mode.

INPUT

Filename (String): the name of the file to open.

Mode (String): a mode from the following list.

"r"— open an existing text file for reading.

"w"— create a text file or open and truncate an existing text file for writing.

"a"— create a text file or open an existing text filefile for writing. The file position indicator is positioned at the end of the file before each write.

"rb"— open an existing binary file for reading.

"wb"— create a binary file or open and truncate an existing binary file for writing.

"ab"— create a binary file or open an existing binary file for writing. The file position indicator is positioned at the end of the file before each write.

"r+"— open an existing text file for reading and writing.

"w+"— create a text file or open and truncate an existing text file for reading and writing.

"a+"— create a text file or open an existing text file for reading and writing. The file position indicator is positioned at the end of the file before each write.

"r+b" or "rb+"— open an existing binary file for reading and writing.

"w+b" or "wb+"— create a binary file or open and truncate an existing binary file for reading and writing.

OUTPUT

An integer ID that refers to the file opened. This ID is used as input in the read, readline, write, writeline, seek and close functions.

EXAMPLES

The following example writes a line of text to a file and then reads it back in.

```
// Open for writing and reading.
// The file will be created or truncated if it
// already exists.
theFileID = open("TestFile", "w+");

// Write a single line to the file.
writeLine(theFileID, "Hello, world!");

// Rewind the file.
seek(theFileID, 0, <SeekFromStart>);

// Read the line back in.
WhatWeWrote = readLine(theFileID, 100);

// Close the file.
close(theFileID);
```

The following example writes an array of integers to disk. The first 2 bytes contain the length followed by the integers in 16-bit format.

```
// Open for writing and reading.
// The file will be created or truncated if it
```

```
// already exists.
theFileID = open("TestBinaryFile", "wb+");

// Number of items to write.
n = 100;
write(theFileID, integerToIEEEInt16(n));

// Write out the items.
for i = 1 to n do
   write(theFileID, integerToIEEEInt16(i));
end for;

// Rewind the file.
seek(theFileID, 0, <SeekFromStart>);

// Flush the file.
flush(theFileID);

// Read it back in.
// The data is 2 bytes long.
n2 = IEEEInt16ToInteger(read(theFileID, 2));
x = 0;
for i = 1 to n2 do
   x[i] = IEEEInt16ToInteger(read(theFileID, 2));
end for;

// Close the file.
close(theFileID);
```

The following example makes a copy of a file.

```
// Open the two files.
theInputFile = open("TestBinaryFile", "rb");
theOutputFile = open("TestBinaryFile Copy", "wb");

// Loop until an end-of-file is reached copying
// 1024 bytes each time.
repeat forever
   buffer = read(theInputFile, 1024);
   write(theOutputFile, buffer);
   if isEOF(theInputFile) then
      exit repeat;
   end if;
end repeat;

// Close the files.
close(theInputFile);
close(theOutputFile);
```

ALGORITHM AND COMMENTS

Files are automatically closed when the script finishes execution. On Macintosh systems use a colon (:) to specify folder names (e.g., a file called foo inside of a folder called scratch that is on a hard disk named internal would be referred to as "internal:scratch:foo"). There is a limit of 255 characters in a path name specified in this manner.

The low level IO routines are modeled after the standard "C" file IO routines.

■ read

FUNCTION

ByteString = read(FileID, NumberOfBytes)

PURPOSE

Reads the specified number of bytes from the specified file.

INPUT

FileID (Integer): the ID of a file opened with the open function.

NumberOfBytes (Integer): the number of bytes that are to be read from the file.

OUTPUT

A string containing the bytes read.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

If this is a read from a text file then you may use StringToNumber to convert to numbers.

If this is a binary read then the string may contain null characters. Use the functions IEEEInt8ToInteger, IEEEInt16ToInteger, IEEEInt32ToInteger, IEEEReal32ToReal, IEEEReal64ToReal, macReal80ToReal, and macReal96ToReal to convert these strings to numbers.

■ readLine

FUNCTION

ByteString = readLine(FileID, MaxNumberOfBytes)

PURPOSE

Reads one line from the specified file or at most the maximum number of bytes specified.

INPUT

FileID (Integer): the ID of a file opened with the open function.

MaxNumberOfBytes (Integer): the maximum number of bytes that are to be read from the file.

OUTPUT

A string containing the bytes read.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

If this is a read from a text file then you may use stringToNumber to convert to numbers.

This function is not recommended for reads from binary files. Use the read function instead.

■ realToIEEEReal32

FUNCTION

ByteString = realToIEEEReal32(aRealScalar)

PURPOSE

Converts the real scalar into a byte string.

INPUT

aRealScalar (Real): the scalar to be converted. The scalar is converted to an IEEE 32-bit real. Some precision is lost in the process.

OUTPUT

A byte string with a length of 4 bytes.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the write function when writing binary data.

■ realToIEEEReal64

FUNCTION

ByteString = realToIEEEReal64(aRealScalar)

PURPOSE

Converts the real scalar into a byte string.

INPUT

aRealScalar (Real): the scalar to be converted. The scalar is converted to an IEEE 64-bit real. Some precision is lost in the process.

OUTPUT

A byte string with a length of 8 bytes.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the write function when writing binary data.

■ realToMacReal80

FUNCTION

ByteString = realToMacReal80(aRealScalar)

PURPOSE

Converts the real scalar into a byte string.

INPUT

aRealScalar (Real): the scalar to be converted. The unused 16 bits are stripped out and the remaining bytes are returned in a ByteString.

OUTPUT

A byte string with a length of 10 bytes.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is useful in conjunction with the write function when writing binary data.

This function converts 96-bit reals to 80 bits. Since the Macintosh doesn't use 16 bits of the data no precision is lost when saved in this format. This provides for more efficient use of disk space. This method is recommended when exporting binary data when disk space is critical.

■ realToMacReal96

FUNCTION

ByteString = realToMacReal96(aRealScalar)

PURPOSE

Converts the input real scalar into a byte string.

INPUT

aRealScalar (Real): the real scalar to be converted into a byte string. The bytes are copied without conversion.

OUTPUT

A byte string with a length of 12 bytes.

EXAMPLES

Refer to the open function for examples.

This function is useful in conjunction with the write function when writing binary data.

Since the Macintosh doesn't use 16 bits of the data we recommend that you use the function realToMac-Real80 for more efficient use of disk space without loss of precision when saving data.

■ seek

FUNCTION

seek(FileID, ByteOffset, Mode)

PURPOSE

Positions the file pointer for the specified file at the specified number of bytes from the beginning of the file.

INPUT

FileID (Integer): the ID of a file opened with the open function.

ByteOffset (Integer): number of bytes from the beginning of the file. 0 would be the beginning of the file.

Mode (Integer): where to seek from. Valid values are:

`<SeekFromStart>` — Positions the file pointer from the beginning of the file.

`<SeekFromCurrent>` — Positions the file pointer from the current position in the file.

`<SeekFromEnd>` — Positions the file pointer from the end of the file.

OUTPUT

None.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

This function is most useful for reading binary files, for rewinding text files or positioning at the end of a file.

■ stringLength

FUNCTION

integer = stringLength(aString)

PURPOSE

Returns the number of characters in a string.

INPUT

aString (String): a string.

OUTPUT

Length of the input string in characters.

EXAMPLES

Calculate the length of a string.

```
len = stringLength("1234567890");
```

This will return a length of 10 for the input string.

■ write

FUNCTION

write(FileID, ByteString)

PURPOSE

Writes the specified string to the specified file.

INPUT

FileID (Integer): the ID of a file opened with the open function.

ByteString (String): string containing the data to be written to file. The entire string is written.

OUTPUT

None.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

If this is a write to a text file then you may use numberToString to convert to numbers to text.

If this is a write to a binary file you may use the functions integerToIEEEInt8, integerToIEEEInt16, integerToIEEEInt32, realToIEEEReal32, realToIEEEReal64, realToMacReal80, and realToMacReal96 to convert numbers to byte strings.

■ writeLine

FUNCTION

writeLine(FileID, ByteString)

PURPOSE

Writes the specified string to the specified file. A new line character is automatically appended to the string.

INPUT

FileID (Integer): the ID of a file opened with the open function.

ByteString (String): string containing the data to be written to file. The entire string is written.

OUTPUT

None.

EXAMPLES

Refer to the open function for examples.

ALGORITHM AND COMMENTS

If this is a write to a text file then you may use numberToString to convert to numbers to text.

This function is not recommended for writes to binary files. Use the write function instead.

CHAPTER 25
IMPORT/EXPORT FUNCTIONS

■ exportComplexMatrix

FUNCTION

exportComplexMatrix(filename, symbol, headerFlag, fieldwidth, decimalPlaces, formatFlag, delimiterFlag, fieldsPerLine, lineFeedFlag)

PURPOSE

Export a complex matrix to an ASCII text file

INPUT

filename (String): any file name legal in the Operating System. Enclose the filename in quotes ("").

symbol (Symbol): HiQ symbol name of the data you wish to export

headerFlag (Integer): include the symbol type information header in the export file (<true> or <false>)

fieldwidth (Integer): number of spaces per exported data element (blank filled on the left)

decimalPlaces (Integer): number of decimal places to display to the right of the decimal point

formatFlag (Integer): <decimal> or <scientific>

delimiterFlag (Integer): the delimiter character separating each element to be exported. Choices are: tab (<delimit_tab>), space (<delimit_space>), comma (<delimit_comma>), and newline (<delimit_newline>).

fieldsPerLine (Integer): the number of elements to put on each line of the ASCII output file

lineFeedFlag (Integer): end each line with a linefeed (<true> or <false>)

OUTPUT

An ASCII text file

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   exportComplexMatrix(filename, symbol, headerFlag,
//          fieldwidth, decimalPlaces, formatFlag,delimiterFlag,
//          fieldsPerLine, lineFeedFlag)

//   the full path name
filename =  "expComplexMatrix";

//   to include the symbol header
headerFlag = <true> ;
```

```
//   to fix the number of spaces per exported data element
fieldwidth  = 9;

//   decimal places after the decimal point
decimalPlaces = 2;

//   notation
formatFlag = <scientific>;

//   the type of the delimeter character
delimiterFlag = <delimit_comma>;

//   the number of exported elements per line
fieldsPerLine = 3;

//   to end each line with a linefeed
lineFeedFlag  = <true>;

A = { (1.1, 1.1),   (2.2,2.2),   (3.3,3.3)  ;
          (4.4, 4.4),  (5.5, 5.5),   (6.6, 6.66666)  } ;
exportComplexMatrix(filename, A, headerFlag,
          fieldwidth, decimalPlaces, formatFlag,delimiterFlag,
          fieldsPerLine, lineFeedFlag);

//   Results :
//    A:  2 rows, 3 columns
 //            1.1 +  1.1i          2.2 +  2.2i         3.3 + 3.3i
//              4.4 + 4.4i          5.5 +  5.5i         6.6 + 6.66666i
//
//    decimalPlaces:  2
//    delimiterFlag:  203
//    fieldsPerLine:  3
//    fieldwidth:  9
//    filename:  expComplexMatrix
//    formatFlag:  1
//    headerFlag:  1
//    lineFeedFlag:  1
//
//    The file "expComplexMatrix"  contains :
//    > complex matrix "A"
//
//         Rows = 2
//      Columns = 3
//
//     ( 1.10e+00, 1.10e+00),( 2.20e+00, 2.20e+00),
//     ( 3.30e+00, 3.30e+00)
//     ( 4.40e+00, 4.40e+00),( 5.50e+00, 5.50e+00),
//     ( 6.60e+00, 6.67e+00)
```

■ exportComplexScalar

FUNCTION

exportComplexScalar(filename, symbol, headerFlag, fieldwidth, decimalPlaces, formatFlag)

PURPOSE

Export a complex scalar to an ASCII text file

INPUT

filename (String): any file name legal in the Operating System. Enclose the filename in quotes ("").

symbol (Symbol): HiQ symbol name of the data you wish to export

headerFlag (Integer): include the symbol type information header in the export file (<true> or <false>)

fieldwidth (Integer): number of spaces per exported data element (blank filled on the left)

decimalPlaces (Integer): number of decimal places to display to the right of the decimal point

formatFlag (Integer): <decimal> or <scientific>

OUTPUT

An ASCII text file

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   exportComplexScalar(filename, symbol,
//             headerFlag, fieldwidth,
//             decimalPlaces, formatFlag)

//   the full path name
filename =  "expComplexScalar";

//   to include the symbol header
headerFlag = <true> ;

//   to fix the number of spaces per exported data element
fieldwidth  = 9;

//   decimal places after the decimal point
decimalPlaces = 2;

//   notation
formatFlag = <scientific>;
x = (1.1,   1.11111);
 exportComplexScalar(filename, x, headerFlag, fieldwidth,
                     decimalPlaces, formatFlag);

//   Results :
//   decimalPlaces:  2
```

```
//   fieldwidth:  9
//   filename:  expComplexScalar
//   formatFlag:  1
//   headerFlag:  1
//   x:        1.1 +  1.11111i
//    The file "expComplexScalar"  contains :
//    > complex scalar "x"
//
//    ( 1.10e+00, 1.11e+00)
```

■ exportComplexVector

FUNCTION

exportComplexVector(filename, symbol, headerFlag, fieldwidth, decimalPlaces, formatFlag, delimiterFlag, fieldsPerLine)

PURPOSE

Export a complex vector to an ASCII text file

INPUT

filename (String): any file name legal in the Operating System. Enclose the filename in quotes ("").

symbol (Symbol): HiQ symbol name of the data you wish to export

headerFlag (Integer): include the symbol type information header in the export file (<true> or <false>)

fieldwidth (Integer): number of spaces per exported data element (blank filled on the left)

decimalPlaces (Integer): number of decimal places to display to the right of the decimal point

formatFlag (Integer): <decimal> or <scientific>

delimiterFlag (Integer): the delimiter character separating each element to be exported. Choices are: tab (<delimit_tab>), space (<delimit_space>), comma (<delimit_comma>), and newline (<delimit_newline>).

fieldsPerLine (Integer): the number of elements to put on each line of the ASCII output file

OUTPUT

An ASCII text file

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   exportComplexVector(filename, symbol, headerFlag,
//          fieldwidth, decimalPlaces, formatFlag,
//          delimiterFlag, fieldsPerLine)

//   the full path name
filename =  "expComplexVector";
```

```
//   to include the symbol header
headerFlag = <true> ;

//   to fix the number of spaces per exported data element
fieldwidth  = 9;
//   decimal places after the decimal point
decimalPlaces = 2;

//   notation
formatFlag = <scientific>;

//   the type of the delimiter character
delimiterFlag = <delimit_comma>;

//   the number of exported elements per line
fieldsPerLine = 3;

w = { (1.1, 1.1),   (2.2,2.2),   (3.3,3.3),   (4.4, 4.4),
          (5.5, 5.5),   (6.6, 6.66666)  } ;

exportComplexVector(filename, w, headerFlag,
          fieldwidth, decimalPlaces, formatFlag,delimiterFlag,
          fieldsPerLine);

//   Results :
//   decimalPlaces:  2
//   delimiterFlag:  203
//   fieldsPerLine:  3
//   fieldwidth:  9
//   filename:  expComplexVector
//   formatFlag:  1
//   headerFlag:  1
//   w:  6 columns
//         1.1 + 1.1i       2.2 + 2.2i    3.3 + 3.3i
//  4.4 +  4.4i       5.5 +  5.5i     6.6 +  6.66666i
//
//    The file "expComplexVector"  contains :
//> complex vector "w"
//
//    Dimension = 6
//  isRowVector = 1
// ( 1.10e+00, 1.10e+00),( 2.20e+00, 2.20e+00),
// ( 3.30e+00, 3.30e+00),
// ( 4.40e+00, 4.40e+00),( 5.50e+00, 5.50e+00),
// ( 6.60e+00, 6.67e+00)
```

■ exportIntegerMatrix

FUNCTION

exportIntegerMatrix(fileName, symbol, headerFlag, fieldWidth, delimiterFlag, fieldsPerLine, lineFeedFlag)

PURPOSE

Export an integer matrix to an ASCII text file

INPUT

filename (String): any file name legal in the Operating System. Enclose the filename in quotes ("").

symbol (Symbol): HiQ symbol name of the data you wish to export

headerFlag (Integer): include the symbol type information header in the export file (<true> or <false>)

fieldwidth (Integer): number of spaces per exported data element (blank filled on the left)

delimiterFlag (Integer): the delimiter character separating each element to be exported. Choices are: tab (<delimit_tab>), space (<delimit_space>), comma (<delimit_comma>), and newline (<delimit_newline>).

fieldsPerLine (Integer): the number of elements to put on each line of the ASCII output file

lineFeedFlag (Integer): end each line with a linefeed (<true> or <false>)

OUTPUT

An ASCII text file

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   exportIntegerMatrix(fileName, symbol, headerFlag,
//          fieldWidth,  delimiterFlag, fieldsPerLine, lineFeedFlag)

//   the full path name
filename =  "expIntegerMatrix";

//   to include the symbol header
headerFlag = <true> ;

//   to fix the number of spaces per exported data element
fieldwidth  = 9;

//   the type of the delimiter character
delimiterFlag = <delimit_comma>;

//   the number of exported elements per line
fieldsPerLine = 3;

//   to end each line with a linefeed
lineFeedFlag  = <true>;
```

```
A = { 1,   2,   3 ;
      4,   5,   6 } ;
exportIntegerMatrix(filename, A, headerFlag, fieldwidth,
                          delimiterFlag, fieldsPerLine, lineFeedFlag);

//   Results :
//    A:  2 rows, 3 columns
//            1        2        3
//            4        5        6
//
//    delimiterFlag:  203
//    fieldsPerLine:  3
//    fieldwidth:  9
//    filename:  expIntegerMatrix
//    headerFlag:  1
//    lineFeedFlag:  1
//
//    The file "expIntegerMatrix"  contains :
//    > integer matrix "A"
//
//            Rows = 2
//            Columns = 3
//
//             1,        2,        3
//             4,        5,        6
```

■ exportIntegerScalar

FUNCTION

exportIntegerScalar(filename, symbol, headerFlag, fieldwidth)

PURPOSE

Export an integer scalar to an ASCII text file.

INPUT

filename (String): any file name legal in the Operating System. Enclose the filename in quotes ("").

symbol (Symbol): HiQ symbol name of the data you wish to export

headerFlag (Integer): include the symbol type information header in the export file (<true> or <false>)

fieldwidth (Integer): number of spaces per exported data element (blank filled on the left)

OUTPUT

An ASCII text file

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   exportIntegerScalar(filename, symbol, headerFlag, fieldwidth)
//   the full path name
filename =  "expIntegerScalar";

//   to include the symbol header
headerFlag = <true> ;

//   to fix the number of spaces per exported data element
fieldwidth  = 7;

symbolSetType (m, <IntegerScalar>);
m = 12345 ;

exportIntegerScalar(filename, m, headerFlag,  fieldwidth);

//   Results :
//   fieldwidth:  7
//   filename:  expIntegerScalar
//   headerFlag:  1
//   m:  12345
//
//    The file "expIntegerScalar"  contains :
//   > integer scalar "m"
//
//      12345
```

■ exportIntegerVector

FUNCTION

exportIntegerVector(filename, symbol, headerFlag, fieldwidth, delimiterFlag, fieldsPerLine)

PURPOSE

Export an integer vector to an ASCII text file

INPUT

filename (String): any file name legal in the Operating System. Enclose the filename in quotes ("").

symbol (Symbol): HiQ symbol name of the data you wish to export

headerFlag (Integer): include the symbol type information header in the export file (<true> or <false>)

fieldwidth (Integer): number of spaces per exported data element (blank filled on the left)

delimiterFlag (Integer): the delimiter character separating each element to be exported. Choices are: tab (<delimit_tab>), space (<delimit_space>), comma (<delimit_comma>), and newline (<delimit_newline>).

fieldsPerLine (Integer): the number of elements to put on each line of the ASCII output file

OUTPUT

An ASCII text file

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   exportIntegerVector(filename, symbol, headerFlag,
//              fieldwidth, delimiterFlag, fieldsPerLine)

//   the full path name
filename =  "expIntegerVector";

//   to include the symbol header
headerFlag = <true> ;

//   to fix the number of spaces per exported data element
fieldwidth  = 7;

//   the type of the delimeter character
delimiterFlag = <delimit_comma>;

//   the number of exported elements per line
fieldsPerLine = 3;

symbolSetType (v, <IntegerVector>);
v = { 1,   2,   3 ,   4,   5,   6  } ;

exportIntegerVector(filename, v, headerFlag,
            fieldwidth, delimiterFlag, fieldsPerLine);

//   Results :
//   delimiterFlag:  203
//   fieldsPerLine:  3
//   fieldwidth:  7
//   filename:  expIntegerVector
//   headerFlag:  1
//   v:  6 columns
//   1 2 3 4 5 6
//
//    The file "expIntegerVector"  contains :
//   > integer vector "v"
//
//       Dimension = 6
//     isRowVector = 1
//
//          1,      2,      3,
//          4,      5,      6
```

■ exportNumeric

FUNCTION

exportNumeric(filename, symbol, delimiter, digits)

PURPOSE

To export a numeric symbol to an external file with a user-specified delimiter

INPUT

filename (String): the name of the file. Enclose in quotes (" ").

symbol: the symbol to be exported

delimiter (String): the character which separates row elements. Enclose in quotes.

digits (Integer): for real or complex numbers, the number of digits to the right of the decimal point; ignored for integers.

OUTPUT

An ASCII text file

EXAMPLES

```
exportNumeric("theData", a, "x", 3);

// Row elements the real matrix a are separated by "x" when written
// to the file "theData" in decimal format with 3 digits to the right
// of the decimal
```

■ exportRealMatrix

FUNCTION

exportRealMatrix(filename, symbol, headerFlag, fieldwidth, decimalPlaces, formatFlag, delimiterFlag, fieldsPerLine, lineFeedFlag)

PURPOSE

Export a real matrix to an ASCII text file

INPUT

filename (String): any file name legal in the Operating System. Enclose the filename in quotes ("").

symbol (Symbol): HiQ symbol name of the data you wish to export

headerFlag (Integer): include the symbol type information header in the export file (<true> or <false>)

fieldwidth (Integer): number of spaces per exported data element (blank filled on the left)

decimalPlaces (Integer): number of decimal places to display to the right of the decimal point

formatFlag (Integer): <decimal> or <scientific>

delimiterFlag (Integer): the delimiter character separating each element to be exported. Choices are: tab (<delimit_tab>), space (<delimit_space>), comma (<delimit_comma>), and newline (<delimit_newline>).

fieldsPerLine (Integer): the number of elements to put on each line of the ASCII output file

lineFeedFlag (Integer): end each line with a linefeed (<true> or <false>)

OUTPUT

An ASCII text file

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   exportRealMatrix(filename, symbol, headerFlag,
//          fieldwidth, decimalPlaces, formatFlag,delimiterFlag,
//          fieldsPerLine, lineFeedFlag)

//   the full path name
filename =  "expRealMatrix";

//   to include the symbol header
headerFlag = <true> ;

//   to fix the number of spaces per exported data element
fieldwidth  = 9;

//   decimal places after the decimal point
decimalPlaces = 2;

//   notation
formatFlag = <scientific>;

//   the type of the delimeter character
delimiterFlag = <delimit_comma>;

//   the number of exported elements per line
fieldsPerLine = 5;

//   to end each line with a linefeed
lineFeedFlag  = <true>;

symbolSetType (A, <RealMatrix>);
A = { 1.1,    2.2,    3.3 ;
      4.4,    5.5,    6.66666} ;
exportRealMatrix(filename, A, headerFlag,
          fieldwidth, decimalPlaces, formatFlag,delimiterFlag,
          fieldsPerLine, lineFeedFlag);

//   Results :
```

```
//   A:  2 rows, 3 columns
//          1.1        2.2     3.3
//          4.4        5.5     6.66666
//
//   decimalPlaces:  2
//   delimiterFlag:  203
//   fieldsPerLine:  5
//   fieldwidth:  9
//   filename:  expRealMatrix
//   formatFlag:  1
//   headerFlag:  1
//   lineFeedFlag:  1
//
//    The file "expRealMatrix"  contains :
//   > real matrix "A"
//
//        Rows = 2
//     Columns = 3
//
//     1.10e+00, 2.20e+00, 3.30e+00
//     4.40e+00, 5.50e+00, 6.67e+00
```

■ exportRealScalar

FUNCTION

exportRealScalar(filename, symbol, headerFlag, fieldwidth, decimalPlaces, formatFlag)

PURPOSE

Export a real scalar to an ASCII text file

INPUT

filename (String): any file name legal in the Operating System. Enclose the filename in quotes ("").

symbol (Symbol): HiQ symbol name of the data you wish to export

headerFlag (Integer): include the symbol type information header in the export file (<true> or <false>)

fieldwidth (Integer): number of spaces per exported data element (blank filled on the left)

decimalPlaces (Integer): number of decimal places to display to the right of the decimal point

formatFlag (Integer): <decimal> or <scientific>

OUTPUT

An ASCII text file

EXAMPLE

```
//   HiQ-Script Example for the utility function
//  exportRealScalar(filename, symbol, headerFlag, fieldwidth,
//             decimalPlaces, formatFlag)

//   the full path name
filename =  "expRealScalar";

//   to include the symbol header
headerFlag = <true> ;

//   to fix the number of spaces per exported data element
fieldwidth  = 9;

//   decimal places after the decimal point
decimalPlaces = 2;

//   notation
formatFlag = <scientific>;
symbolSetType (x, <RealScalar>);
x =1.99999;
 exportRealScalar(filename, x,
            headerFlag, fieldwidth, decimalPlaces, formatFlag);

//   Results :
//   decimalPlaces:  2
//   fieldwidth:  9
//   filename:  expRealScalar
//   formatFlag:  1
//   headerFlag:  1
//   x:           1.99999
//
//    The file "expRealScalar"  contains :
//   > real scalar "x"
//
//     2.00e+00
```

■ exportRealVector

FUNCTION

exportRealVector(filename, symbol, headerFlag, fieldwidth, decimalPlaces, formatFlag, delimiterFlag, fieldsPerLine)

PURPOSE

Export a real vector to an ASCII text file

INPUT

filename (String): any file name legal in the Operating System. Enclose the filename in quotes ("").

symbol (Symbol): HiQ symbol name of the data you wish to export

headerFlag (Integer): include the symbol type information header in the export file (<true> or <false>)

fieldwidth (Integer): number of spaces per exported data element (blank filled on the left)

decimalPlaces (Integer): number of decimal places to display to the right of the decimal point

formatFlag (Integer): <decimal> or <scientific>

delimiterFlag (Integer): the delimiter character separating each element to be exported. Choices are: tab (<delimit_tab>), space (<delimit_space>), comma (<delimit_comma>), and newline (<delimit_newline>).

fieldsPerLine (Integer): the number of elements to put on each line of the ASCII output file

OUTPUT

An ASCII text file

EXAMPLE

```
//   HiQ-Script Example for the utility function
//   exportRealVector(filename, symbol, headerFlag,
//          fieldwidth, decimalPlaces, formatFlag,
//          delimiterFlag, fieldsPerLine)

//   the full path name
filename =  "expRealVector";

//   to include the symbol header
headerFlag = <true> ;

//   to fix the number of spaces per exported data element
fieldwidth  = 9;

//   decimal places after the decimal point
decimalPlaces = 2;

//   notation
formatFlag = <scientific>;

//   the type of the delimeter character
delimiterFlag = <delimit_comma>;

//   the number of exported elements per line
fieldsPerLine = 3;
w = { 1.1,    2.2,     3.3,     4.4,    5.5,    6.6666 } ;
symbolSetType (w, <RealVector>);
exportRealVector(filename, w, headerFlag,
         fieldwidth, decimalPlaces, formatFlag,delimiterFlag,
          fieldsPerLine);
//   Results :
```

```
//   decimalPlaces:  2
//   delimiterFlag:  203
//   fieldsPerLine:  3
//   fieldwidth:  9
//   filename:  expRealVector
//   formatFlag:  1
//   headerFlag:  1
//   w:  6 columns
//         1.1      2.2      3.3     4.4      5.5     6.6666
//    The file "expRealVector"  contains :
//   > real vector "w"
//
//      Dimension = 6
//     isRowVector = 1
//
//     1.10e+00, 2.20e+00, 3.30e+00,
//     4.40e+00, 5.50e+00, 6.67e+00
```

■ exportSymbol

FUNCTION

exportSymbol(filename, symbol)

PURPOSE

To export a numeric or text symbol to an external file

INPUT

filename (String): the name of the file. Enclose in quotes (" ").

symbol: the name of the symbol to be exported

OUTPUT

An ASCII text file

EXAMPLES

```
exportSymbol("Radius_File", radius);
```

■ getFileName

FUNCTION

pathName = getFileName(filePathName, fileType)

PURPOSE

Opens a dialog box that prompts the user to specify an existing file. Places the chosen path and file name into a string. This function would typically be used to interactively select a file from which to import data.

INPUT

filePathName (String): defines the default path from which the standard file is opened. The file name is ignored.

fileType (String): indicates the file type that is searched for. If it is empty, all files are displayed.

OUTPUT

pathName (String): the full path name of the file, i.e., volume, directories and the file name

EXAMPLES

```
aString   = getFileName("HD:Todays Data:Sales", "TEXT");
aPicture = getFileName("HD:Todays Data:SalesGraph", "PICT");
```

■ importNumeric

FUNCTION

x = importNumeric(filename, delimiterList)

PURPOSE

To import a numeric symbol from an external file with user-specified delimiters

INPUT

filename (String): the name of the file. Enclose in quotes (" ").

delimiterList (String): characters in this list are treated as delimiters (delimiters separate row elements; each row must end with a carriage return.) Enclose in quotes. Each delimiter denotes a position; thus importing "1,,2" results in "1 0 2".

OUTPUT

Numeric symbol

EXAMPLES

```
anArray = importNumeric("theData", "ax;");

// Row elements may be separated by "a", "x", or ";"
```

■ importSymbol

FUNCTION

x = importSymbol(filename, type)

PURPOSE

To import a numeric or text symbol from an external file

INPUT

filename (String): the name of the file. Enclose in quotes (" ").

type (Integer scalar): Defines the new symbol type to be created (<importNumeric> or <importText>)

OUTPUT

Numeric or text symbol

EXAMPLES

```
radius = importSymbol("Radius", <importNumeric>);
```

■ putFileName

FUNCTION

putFileName (pathAndDefaultName, fullFileName, MacVREFNumber)

PURPOSE

Opens a dialog box that prompts the user to specify a new path and file name. Places the new path and file name into a string. This function would typically be used to interactively create a file to export data.

INPUT

pathAndDefaultName (String): the default directory and defaulted to file name

fullFileName (String): the result of the put file function is placed into this string. This must be a String symbol.

OUTPUT

None

EXAMPLES

```
aString = putFileName ("HD:Todays Data:Sales", myPathName, aVolumeRef-
erence);
aString = putFileName ("TheData:Sales", myPathName, aVolumeReference);
```

CHAPTER 26
PROBLEM SOLVER FUNCTIONS

■ integAdapt

FUNCTION

[answer, error, numEval] = integAdapt (integFct, lowerLimit, upperLimit, absError, relError, maxEval);

PURPOSE

Compute the integral:

$$\int_a^b f(x)\,dx$$

using the best available general purpose adaptive integration code.

INPUT

integFct (Function): the function to be integrated

lowerLimit (Real Scalar): the lower numerical limit (a) of integration

upperLimit (Real Scalar): the upper numerical limit (b) of integration

absError (Real Scalar): the absolute accuracy requested

relError (Real Scalar): the relative accuracy requested

maxEval (Integer Scalar): upper bound on the number of integrand evaluations

OUTPUT

answer (Real Scalar): the numerically computed value of the integral

error (Real Scalar): computed absolute error estimate, should equal or exceed the absolute value of 1 - result

numEval (Integer Scalar): the actual number of integrand evaluations used

EXAMPLE

Compute the integral:

$$\int_0^1 \frac{\ln(x)}{\sqrt{x}}\,dx$$

whose exact answer is -4.

Example Script:

```
project answer, error, numEval;

function integFct(x)
   return x^(-1/2)*ln(x);
end function;

lowerLimit = 0.0;
upperLimit = 1.0;
absError = 1.0e-6;
relError = 1.0e-6;
maxEval = 25000;

[ answer, error, numEval ] = integAdapt (integFct,
   lowerLimit, upperLimit, absError, relError, maxEval );
//  Expected = -4;
```

ALGORITHM AND COMMENTS

This is an algorithm based on the most robust and sophisticated integration routine available in QUADPACK, a modern subroutine package for automatic integration. Called QAGS, it is a general purpose adaptive quadrature program. More precisely, QAGS is an integrator based on the combination of globally adaptive interval subdivision and extrapolation that will eliminate several kinds of integrand singularity effects.This program is rather complicated, hence only a superficial explanation will be provided with the algorithm outline.

In general, if we let:

integral = exact value of the integral and answer = result from the integrator,

we try to satisfy the accuracy requirement:

$$| \text{integral} - \text{answer} | \leq \text{tol} = \max \{ \text{abserr}, \text{relerr}|\text{integral}| \} ,$$

while not exceeding neval = the maximum number of integrand evaluations.

Local integral approximations over subdivisions of [a,b] are performed using a 21-point Kronrod rule. Local error estimates are computed using this rule along with a 10-point Gaussian rule. Let: sq = sum of local integral approximations over a partition on [a,b] and se = sum of local error estimates on the same partition. Define level = when smallest interval of the partition of [a,b] has length $| b - a | / 2^{level}$; then a small interval is one of this length and a large interval is one that is simply larger than a small one. A new level is introduced whenever the bisection of a small interval within the current level is required.

Define: sbe = sum of the error estimates over all large intervals

exq = integral approximation obtained by extrapolation

exe = error estimate with respect to exq

extol = max { abserr, relerr|exq| }

Algorithm Outline:

Initialization: entered values: a, b, abserr, relerr, tol, neval
compute: sq, se, tol = max { abserr, relerr|sq| }
set: sbe = se, exe = MAXNO, extol = tol, level = 0
While se > tol
and exe > extol
and interval selected for subdivision is not too small
and maximum number of subdivisions has not been reached
and no roundoff error detected
and continued calculation should yield improved values
select interval with largest error estimate for subdivision
if no new level is introduced
subdivide current interval and update sq, se, sbe, tol
else
while sbe > extol
and no roundoff error detected over large intervals
subdivide the large interval with largest error
and update sq, se, sbe, tol
extrapolate and update exq, exe, extol, sbe (= se)
level = level + 1
If exe/|exq| > se/|sq|
set answer = sq, abserr = se
else
test for divergence of the integral and set answer = exq, abserr = exe.
Return.

REFERENCES

This program is a modified and translated version of the QAGS routine contained in: Piessens, R., de Doncker-Kapenga, E., Uberhuber, C.W., and Kahaner, D.K., *QUADPACK*, Springer-Verlag, New York, 1983, pgs. 147 - 155.

■ integAlgLogSingular

FUNCTION

[answer, error, numEval] = integAlgLogSingular (integFct, lowerLimit, upperLimit, absError, relError, limit, weightFctFlag, alpha, beta);

PURPOSE

Compute the integral:

$$\int_a^b f(x)\, w(x)\, dx$$

where w(x) is one of the following types of weight functions:

type 1: $w(x) = (x-a)^{\alpha} * (b-x)^{\beta}$

type 2: $w(x) = (x-a)^{\alpha} * (b-x)^{\beta} * \ln(x-a)$

type 3: $w(x) = (x-a)^{\alpha} * (b-x)^{\beta} * \ln(b-x)$

type 4: $w(x) = (x-a)^{\alpha} * (b-x)^{\beta} * \ln(x-a) * \ln(b-x)$;

INPUT

integFct (Function): the function f(x) in the integrand

lowerLimit (Real Scalar): the lower numerical limit (a) of integration

upperLimit (Real Scalar): the upper numerical limit (b) of integration

absError (Real Scalar): the absolute accuracy requested

relError (Real Scalar): the relative accuracy requested

limit (Integer Scalar): upper bound on the number of subdivisions of (a, b)

weightFctFlag (Integer Scalar): flag that indicates which type of singular weighted integral; see the listing above for the flag values

alpha (Integer Scalar): a parameter indicating the power of (x-a) in w(x)

beta (Integer Scalar): a parameter indicating the power of (b-x) in w(x)

OUTPUT

answer (Real Scalar): the numerically computed value of the integral

error (Real Scalar): computed absolute error estimate, should equal or exceed the absolute value of 1 - result

numEval (Integer Scalar): the actual number of integrand evaluations used

EXAMPLE

Compute the integral:

$$\int_0^1 \frac{1}{(1 + (\ln(x))^2)^2}$$

whose exact answer is $(Ci(1)\sin(1) + (\pi/2 - Si(1))\cos(1))/\pi \approx -0.1892752$.

Example Script:

```
project answer, error, numEval, integFct;

function integFct(x)
    return 1/(1+(ln(x))^2)^2;
end function;

local lowerLimit, upperLimit, absError, relError, limit, weightFctFlag,
   alpha, beta;

lowerLimit = 0.0;
upperLimit = 1.0;
absError = 1.0e-6;
relError = 1.0e-3;
limit = 500;
weightFctFlag = 2;
alpha = 0;
beta = 0;

[answer, error, numEval] = integAlgLogSingular(integFct, lowerLimit,
upperLimit, absError, relError, limit, weightFctFlag, alpha, beta);
//  Expected =    -0.189274744041429
```

ALGORITHM AND COMMENTS

A globally adaptive subdivision method is utilized for evaluating these integrals which uses a modified Clenshaw-Curtis integration on those subintervals that contain a or b. The globally adaptive algorithm can be roughly outlined as follows:

1. Initialization. Set: result = Approx. quadrature of f over [a, b]

absolute error = Error estimate of result

tolerance = max {abserr, relerr(|result|)}

(Note: the approximate quadrature is performed using one of the 2k+1 point Kronrod rules (k = 7, 10, 15, 20, 25, 30); the error estimate is performed by comparing a k-point Gaussian quadrature with these Kronrod quadratures in norm; the tolerance is, of course, determined by the user-selected values of the error tolerances abserr and relerr)

2. while (absolute error > tolerance) and (the interval with the largest error estimate is bounded far enough from zero) and (maximum number of subdivisions has not been attained) and (roundoff error is acceptable) do:

subdivide the interval with the largest error estimate

update the values of result, absolute error, and tolerance

Return.

In integAlgLogSing, one starts by bisecting the starting interval [a, b] and then applies the modified Clenshaw-Curtis integration on each subinterval; in all of the further steps, the modified Clenshaw-Curtis integration method is applied on those subintervals that contain a or b and lower Gauss-Kronrod rules (7 and 15

point rules) are applied to the remaining subintervals.

REFERENCES

This routine is a program based on the algorithms described in: 1) Davis, P.J. and Rabinowitz, P., *Methods of Numerical Integration*, 2nd ed., Academic Press, 1984, and 2) Piessens, R., de Doncker-Kapenga, E., Uberhuber, C.W., and Kahaner, D.K., *QUADPACK*, Springer-Verlag, NewYork, 1983.

■ integBkpts

FUNCTION

[answer, error, numEval] = integBkpts(integFct, lowerLimit, upperLimit, absError, relError, maxEval, bkptsVector);

PURPOSE

Compute the integral:

$$\int_a^b f(x)\,dx$$

where f(x) over the interval (a, b) has break points, i.e., points $p_1 < p_2 < ... < p_n$ where local difficulties of f(x) may occur, such as singularities and discontinuities.

INPUT

integFct (Function): the function f(x) to be integrated

lowerLimit (Real Scalar): the lower numerical limit (a) of integration

upperLimit (Real Scalar): the upper numerical limit (b) of integration

absError (Real Scalar): the absolute accuracy requested

relError (Real Scalar): the relative accuracy requested

maxEval (Integer Scalar): an upper bound on the number of integrand evaluations

bkptsVector (Real Vector): vector of user-provided interior break points

OUTPUT

answer (Real Scalar): the numerically computed value of the integral

error (Real Scalar): computed absolute error estimate, should equal or exceed the absolute value of 1 - result

numEval (Integer Scalar): the actual number of integrand evaluations used

EXAMPLE

Compute the integral:

$$\int_0^3 x^3 \ln\left(\left|(x^2-1)\ (x^2-2)\right|\right) dx$$

whose exact answer is: 61ln(2) + 77ln(7)/4 - 27 ≈ 52.740748.

Example Script:

```
project answer, error, numEval, integFct;

function integFct(x)
   return (x^3*ln(abs((x^2-1)*(x^2-2))));
end function;

local lowerLimit, upperLimit, absError, relError, maxEval;
lowerLimit = 0.0;
upperLimit = 3.0;
absError = 0.0;
relError = 1.0e-2;
maxEval = 25000;
bkptsVector = {1, 1.414213562};
[answer, error, numEval] = integBkpts(integFct, lowerLimit,
   upperLimit, absError, relError, maxEval, bkptsVector);

//  Expected =            52.7413831482566
```

ALGORITHM AND COMMENTS

This algorithm operates similarly to the algorithm for integAdapt. However, there are some slight differences. Assuming that n break points partition [a, b], where a < b, i.e., $a < p_1 < p_2 < ... < p_n < b$, and defining $p_0 = a$, $p_{n+1} = b$, the first integral approximation is determined by partitioning [a, b] and taking the sum of the contributions from all of the subintervals. From this point on, each subdivision is then a bisection. Thus, we see that the definition of level, as used in the integAdapt routine, has been altered. Furthermore, the definitions of small and large subintervals have also changed. Given the process at the level l (≥ 0), the smallest subinterval in the partition of $[p_k, p_{k+1}]$ then has the length $(p_{k+1} - p_k)*2^{-l}$. Subintervals of this length are defined to be small; the other ones are then defined to be large.

REFERENCES

This routine is a program based on the algorithm given for the QAGP routine contained in: Piessens, R., de Doncker-Kapenga, E., Uberhuber, C.W., and Kahaner, D.K., *QUADPACK*, Springer-Verlag, NewYork, 1983, pgs. 130-136.

■ integFourier

FUNCTION

[answer, error, numEval] = integFourier(integFct, lowerLimit, absError, limit, weightFctFlag, alpha);

PURPOSE

Compute the integral:

$$\int_{a}^{\infty} f(x)\, w(x)\, dx$$

where w(x) is one of the two Fourier integral weight functions:

<sinwx>: w(x) = sin(αx); <coswx>: w(x) = cos(αx)

INPUT

integFct (Function): the function f(x) to be integrated

lowerLimit (Real Scalar): the lower numerical limit (a) of integration

absError (Real Scalar): the absolute accuracy requested

limit (Integer Scalar): upper bound on the number of subdivisions of (a, b)

weightFctFlag (Integer Scalar): flag that indicates which type of Fourier integral; see the listing above for the flag values

alpha (Real Scalar): a parameter (α) in the weight function w(x)

OUTPUT

answer (Real Scalar): the numerically computed value of the integral

error (Real Scalar): computed absolute error estimate, should equal or exceed the absolute value of 1 - result

numEval (Integer Scalar): the actual number of integrand evaluations used

EXAMPLE

Compute the integral:

$$\int_{0}^{\infty} \frac{\cos\left(\frac{\pi x}{2}\right)}{\sqrt{x}} dx$$

whose exact answer is 1.

Example Script:

```
project answer, error, numEval, integFct;
```

```
function integFct(x)
    return (1/sqrt(x));
end function;

local lowerLimit, absError, limit, weightFctFlag, alpha;
lowerLimit = 0.0000001;
absError = 1.0e-3;
limit = 500;
weightFctFlag = <coswx>;
alpha = 1.5707963268;
[answer, error, numEval] = integFourier(integFct, lowerLimit,
absError, limit, weightFctFlag, alpha);

//  Expected =          0.999997292771344
```

ALGORITHM AND COMMENTS

integFourier applies a specially designed adaptive integration method for oscillatory integrands over successive intervals I_k = [a + (k-1)*length, a + k*length], k = 1, 2, ..., where the intervals I_k have the constant lengths:

$$\text{length} = \frac{(2|\text{alpha}| + 1)\,\pi}{|\text{alpha}|}$$

where (|alpha|) represents the largest integer that is ≤ |alpha|. Because the length is equal to an odd number of half periods, the contributions to the integral alternate in sign over successive intervals when f(x) ≥ 0 and decreases monotonically over [a, ∞). Acceleration of the convergence of the series of contributions to the integral is accomplished by applying the ε-algorithm; consult the papers referred to in the QUADPACK reference below for a discussion of this algorithm. The reader is to be warned that the ε-algorithm is poorly documented and he needs to test the validity of the claims made in the references.

The user-specified absolute error tolerance abserr is used to prescribe the accuracy requirement:

tolerance on kth iteration = d_k * abserr, where $d_k = (1\text{-}e)*e^{k-1}$, k = 1, 2, ..., and e = 0.9.

This requirement is relaxed when difficulties arise within some of the subintervals, in which case the tolerance is reset to the value:

tolerance on kth iteration = d_k * max{abserr, max of error estimate over all I_k , where $1 \le k \le i\text{-}1$}.

REFERENCES

This routine is a program based on the algorithm QAWF, contained in: Piessens, R., de Doncker-Kapenga, E., Uberhuber, C.W., and Kahaner, D.K., *QUADPACK*, Springer-Verlag, NewYork, 1983.

■ integGauss

FUNCTION

[answer, error, numEval] = integGauss(integFct, lowerLimit, upperLimit, absError, relError);

PURPOSE

Compute the integral:

$$\int_a^b f(x)\,dx$$

using a rapid nonadaptive Gaussian integration routine for smooth functions.

INPUT

integFct (Function): the function f(x) to be integrated

lowerLimit (Real Scalar): the lower numerical limit (a) of integration

upperLimit (Real Scalar): the upper numerical limit (b) of integration

absError (Real Scalar): the absolute accuracy requested

relError (Real Scalar): the relative accuracy requested

OUTPUT

answer (Real Scalar): the numerically computed value of the integral

error (Real Scalar): computed absolute error estimate, should equal or exceed the absolute value of 1 - result

numEval (Integer Scalar): the actual number of integrand evaluations used

EXAMPLE

Compute the integral:

$$\int_0^1 \frac{1}{(1+x^2)}\,dx$$

whose exact answer is $\pi/4 \approx 0.78539816339$.

Example Script:

```
project answer, error, numEval, integFct;
function integFct(x)
    return (1/(1+x^2));
end function;
local lowerLimit, upperLimit, absError, relError;
```

```
lowerLimit = 0;
upperLimit = 1;
absError = 1.0e-6;
relError = 1.0e-6;
[answer, error, numEval] = integGauss(integFct, lowerLimit,
upperLimit, absError, relError);
//  Expected =          0.785398163397448
```

ALGORITHM AND COMMENTS

This program is based on the algorithm given for the non-adaptive automatic integration program contained in QUADPACK called QNG. It consists of a sequence of quadrature rules that have an increasingly greater degree of algebraic precision. There is a fixed number of integrand evaluations allowed in this program.

The program begins by evaluating the integral using a 10-point Gaussian rule, then constructs a nested sequence of three Patterson rules for that rule. Each of these rules uses the abscissas of the previous rule (the number of abscissas of each rule = 2 * previous number of abscissas + 1) and error estimates are computed for every rule. Instead of attempting to reduce the absolute error tolerance abserr, this algorithm computes the next quadrature sum in the sequence. Consequently, the final result is given by applying the 21-point Kronrod rule obtained by the optimal addition of abscissas to the 10 point rule, or by applying a 43-point rule obtained by the optimal addition of abscissas to the 21 point Kronrod rule, or by applying a 87-point rule obtained by the optimal addition of abscissas to the 43 point rule.

Define: q(m) = result obtained from m-point Kronrod rule

e(m) = error estimate for the m-point Kronrod rule

n = number of abscissas

Algorithm Outline:

Initialization: entered values: a, b, abserr, relerr, tol

compute: answer = q(21), abserr = e(21),

tol = max { abserr, relerr|answer| }

set: n = 21

While abserr > tol and n < 87

answer = q(2n+1) and abserr = e(2n+1)

update tol

n = 2n+1

Return.

REFERENCES

This routine is a program based on the algorithm given for the QNG routine contained in: Piessens, R., de Doncker-Kapenga, E., Uberhuber, C.W., and Kahaner, D.K., *QUADPACK*, Springer-Verlag, NewYork, 1983, pgs. 130-136.

■ integInfinite

FUNCTION

[answer, error, numEval] = integInfinite(integFct, bound, absError, relError, maxEval, typeFlag);

PURPOSE

Compute one of the three types of integrals:

$$\int_{b}^{\infty} f(x)\,dx \qquad \int_{-\infty}^{b} f(x)\,dx \qquad \int_{-\infty}^{\infty} f(x)\,dx$$

INPUT

integFct (Function): the function to be integrated

bound (Real Scalar): b, the finite bound of integration

absError (Real Scalar): the absolute accuracy requested

relError (Real Scalar): the relative accuracy requested

maxEval (Integer Scalar): upper bound on the number of integrand evaluations

typeFlag (Integer Scalar): flag that indicates which type of infinite integral;
typeFlag = <bnd2inf>: integral from b to ∞; typeFlag = <inf2bnd>: integral from -∞ to b; and typeFlag = <inf2inf>: integral from -∞ to ∞

OUTPUT

answer (Real Scalar): the numerically computed value of the integral

error (Real Scalar): computed absolute error estimate, should equal or exceed the absolute value of 1 - result

numEval (Integer Scalar): the actual number of integrand evaluations used

EXAMPLE

Compute the integral:

$$\int_{0}^{\infty} \frac{\ln(x)}{(1+100x^2)}\,dx$$

whose exact answer is - ln(10)/20 ≈ -0.3616892.

Example Script:

```
project answer, error, numEval, integFct;

function integFct(x)
```

```
    return (ln(x)/(1+100*x^2));
end function;

local bound, absError, relError, maxEval, typeFlag;
bound = 0.0;
absError = 0.0;
relError = 1.0e-3;
maxEval = 25000;
typeFlag = <BND2INF>;

[answer, error, numEval] = integInfinite(integFct, bound,
absError, relError, maxEval, typeFlag);

//  Expected =       -0.361689218612702
```

ALGORITHM AND COMMENTS

To compute the integrals:

$$\int_{b}^{\infty} f(x)\,dx \qquad \int_{-\infty}^{b} f(x)\,dx$$

the infinite range of integration (for each of these cases) is transformed to integrating over the range (0, 1], i.e.,

$$\int_{b}^{\infty} f(x)\,dx = \int_{0}^{1} \frac{f\left(b + \frac{1-t}{t}\right)}{t^2}dt$$

for the integral from -∞ to ∞, we use:

$$\int_{-\infty}^{\infty} f(x)\,dx = \int_{0}^{1} \frac{f\left(\frac{1-t}{t}\right) + f\left(\frac{t-1}{t}\right)}{t^2}dt$$

Integration over these intervals then proceeds as in integAdapt, but the underlying use of the quadrature rules is different. This is because the use of quadrature rules of higher degree gain little improvement in estimating the result due to the likelihood of singularities near the point x = 0 in the transformed integrand.

REFERENCES

1) Davis, P.J. and Rabinowitz, P., *Methods of Numerical Integration*, 2nd ed., Academic Press, 1984, and 2) the QAGI routine contained in: Piessens, R., de Doncker-Kapenga, E., Uberhuber, C.W., and Kahaner, D.K.,

QUADPACK, Springer-Verlag, NewYork, 1983, pgs.130-136.

■ integMultiple

FUNCTION

[answer, error] = integMultiple (integFct, lowerLimitFct, upperLimitFct, orderOfInteg, tolerance);

PURPOSE

Multiple Integration: Compute the multi-dimensional integral:

$$\int_{a_1}^{b_1} f_1(x_1) \int_{a_2 x_1}^{b_2(x_1)} f_2(x_1, x_2) \ldots \int_{a_n(x_1, x_2, \ldots x_n)}^{b_n(x_1, x_2, \ldots x_n)} f_n(x_1, x_2, \ldots, x_n) \ \ldots \ dx_n dx_{n-1} \ldots dx_1$$

using an adaptive Gaussian method. Note: there is no Problem Solver Graphical Interface for this method in version 1.0 of HiQ.

INPUT

integFct (Function): the function f(i,x) which returns $f(x_1,\ldots,x_i)$ for each i=1,…,n

lowerLimitFct (Function): the lower limit functions a(i, x[1], ... , x[i - 1]) , i = 1, ... , n

upperLimitFct (Function): the upper limit functions b(i, x[1], ... , x[i - 1]) , i = 1, ... , n

orderOfInteg (Integer Scalar): the dimension n of the integral

tolerance (Real Scalar): the error tolerance used to test convergence

OUTPUT

answer (Real Scalar): the numerically computed value of the integral

error (Real Scalar): an estimate of the error in the answer

EXAMPLE

Compute the integral:

$$\int_{-1}^{1} \int_{-\sqrt{1-x_1^2}}^{\sqrt{1-x_1^2}} \int_{-\sqrt{1-x_1^2-x_2^2}}^{\sqrt{1-x_1^2-x_2^2}} \frac{1}{\sqrt{x_1^2 + x_2^2 - (x_3 - k)^2}} dx_3 dx_2 dx_1$$

for various values of the constant k. The exact result is:

$$\pi\left(2+\left(\frac{1}{k}-k\right)\ln\left(\left|\frac{1+k}{1-k}\right|\right)\right)$$

For k = 2, this value is: 1.10609686434478.

Example Script:

```
project answer, error, integFct, lowerLimitFct, upperLimitFct;

local orderOfInteg, tolerance;

function integFct(i,x)
  select i from
    case 1:
      return 1;
    case 2:
      return 1;
    case 3:
      k=2;
      return 1/(x[1]^2+x[2]^2+(x[3]-k)^2);
  end select;
end function;

function lowerLimitFct(i,x)
  select i from
    case 1:
      return -1;
    case 2:
      return -sqrt(1-x[1]^2);
    case 3:
      return -sqrt(1-x[1]^2-x[2]^2);
  end select;
end function;

function upperLimitFct(i,x)
  select i from
    case 1:
      return 1;
    case 2:
      return sqrt(1-x[1]^2);
    case 3:
      return sqrt(1-x[1]^2-x[2]^2);
  end select;
end function;

orderOfInteg = 3;
tolerance = 1.0e-2;

[anser, error] = integMultiple ( integFct, lowerLimitFct,
upperLimitFct, orderOfInteg, tolerance );
```

```
// Expected:            1.1043023327681
```

ALGORITHM AND COMMENTS

This is an original adaptive integration program that uses error evaluation integral adjustment formulas. It provides a high degree of confidence in its performance on singular integrals and other multiple integrals that are usually difficult to integrate.

Essentially this algorithm performs n-dimensional integration via automatic selection of different Gaussian integration techniques over n-dimensional domains.

The original domain of integration is first bisected in each direction. As a result, one obtains a cluster of 2^n curvilinear hypercubes. For each hypercube of the cluster, two integral approximations are obtained, I2 and I4, using 2-point and 4-point Gaussian rules, respectively. An error estimate is then computed as:

$$e = \frac{|I4 - I2|}{C}$$

where the evaluation coefficient C is a function of the dimension n and the number of Gaussian points used.

The implemented algorithm assumes that, for n-dimensional problems, the error decreases as C/p^2, where p is the number of Gauss points. This assumption is similar to Atkinson's result for one-dimensional problems

$$|\text{error}| < \left(\frac{0.42}{p^2}\right)\max_{-1 \le x \le 1}|f''(x)|$$

Note that this formula does not guarantee that the error of the result I4 or I2 is smaller than e. If the value of e is less than $E/2^n$, where E is the admissible error for the cluster, then the hypercube calculation is done. Otherwise, the procedure is repeated recursively for the hypercube, which becomes a new cluster as a result of bisecting in each direction again. The admissible error for the new cluster is $E/2^n$. The number of recursive bisections is limited by the maximum number of intervals, which is 2^{31}. Refer to reference (4) below for further explanation and justification of the above error estimation process.

The single most important fact in using this code is to appreciate that the tolerance parameter is not an absolute error tolerance or rigorous error bound, but a tool for computing almost any kind of integral, if used correctly. It provides an error that must be satisfied locally within each hypercube. More specifically, the value of tolerance should never be very small; usually about 1.0e-2 will be sufficient for obtaining results that are much more accurate than this parameter may suggest.

REFERENCES

1) Freeman, R.D., "MULTINT," Algorithm 32, Collections of Automatic Computing Machinery (CACM), Vol. 1, 1980; 2) Davis, P.J. and Rabinowitz, P., *Methods of Numerical Integration*, 2nd ed., Academic Press, 1984; 3) Abramowitz, M. and Stegun, I., eds., *Handbook of Mathematical Functions*, Applied Mathematics Series 55, US National Bureau of Standards, 1964; 4) Atkins, K.E., *Introduction to Numerical Analysis*, Wiley & Sons, 1978, p. 240 - refers to the reference: Stroud, A.H. and Secrest, D., *Gaussian Quadrature Formulas*, Prentice-Hall, 1966.

■ integOscillate

FUNCTION

[answer, error, numEval] = integOscillate(integFct, lowerLimit, upperLimit, absError, relError, limit, weightFctFlag, alpha);

PURPOSE

Compute one of the two types of integrals with oscillatory integrands:

$$\int_a^b g(x)\,dx$$

where g(x) is of the form f(x)*w(x), and w(x) is one of the two Fourier integral weight functions:

<sinwx>: w(x) = sin(αx); <coswx>: w(x) = cos(αx)

INPUT

integFct (Function): the function to be integrated

lowerLimit (Real Scalar): the lower numerical limit (a) of integration

upperLimit (Real Scalar): the upper numerical limit (b) of integration

absError (Real Scalar): the absolute accuracy requested

relError (Real Scalar): the relative accuracy requested

limit (Integer Scalar): an upper bound related to the number of subdivisions

weightFctFlag (Integer Scalar): flag that indicates which type of Fourier integral; see the listing above for the flag values

alpha (Real Scalar): the oscillation parameter (argument α in sin(αx) or cos (αx))

OUTPUT

answer (Real Scalar): the numerically computed value of the integral

error (Real Scalar): computed absolute error estimate, should equal or exceed the absolute value of 1 - result

numEval (Integer Scalar): the actual number of integrand evaluations used

EXAMPLE

Compute the integral:

$$\int_0^1 \log(x)\,\sin(10\pi x)\,dx$$

whose exact answer is - (γ + log(10π) - Ci(10π))/(10$^{\pi}$) ≈ -0.1281316.

Example Script:

```
project answer, error, numEval, integFct;

function integFct(x)
    return (ln(x));
end function;

local lowerLimit, upperLimit, limit, alpha, absError, relError;
lowerLimit = 0;
upperLimit = 1;
limit = 500;
absError = 1.0e-6;
relError = 1.0e-6;
weightFctFlag = <sinosc>;
alpha = 10*<pi>;

[answer, error, numEval] = integOscillate(integFct, lowerLimit,
 upperLimit, absError, relError, limit, weightFctFlag, alpha);

   //  Expected =        -0.128136848399157
```

ALGORITHM AND COMMENTS

This program is similar to the integrator contained in QUADPACK called QAWO, designed for integrating oscillatory integrands of the type cos(αx)f(x) or sin(αx)f(x) over a finite inteval [a, b]. The algorithm used is an extrapolation method that is a modification of that used in the Adaptive Method.

If the condition Lα > 4 is satisfied over a subinterval of length L = 2^{-l}* abs(b - a), where l is the level as described in the Adaptive algorithm, then a modified 25 - point Clenshaw - Curtis method is used for the integration over the subinterval. An error estimate is computed from this approximation and from an integral approximation of lower degree, in this case the use of a 13 - point formula is suggested by QUADPACK. Our version uses a choice of two lower formulas, whichever returns a better estimate. If the condition Lα > 4 is not satisfied, we follow QUADPACK's algortihm and use the 7/15 - point Gauss - Kronrod integration formulas.

REFERENCES

1) Davis, P.J. and Rabinowitz, P., *Methods of Numerical Integration*, 2nd ed., Academic Press, 1984, and 2) the QAGI routine contained in: Piessens, R., de Doncker-Kapenga, E., Uberhuber, C.W., and Kahaner, D.K., *QUADPACK*, Springer-Verlag, NewYork, 1983, pgs.130-136.

■ intEqnFredholm

FUNCTION

[solnPoints, meshPoints] = intEqnFredholm(KIntEqns, gIntEqns, leftEndPoint, rightEndPoint, numOfEqns, numUnifMeshPoints, tolerance, maxIterations, defCorrFlag);

PURPOSE

Compute the solution **f**(t) of a system of m Fredholm integral equations of the second kind:

$$\mathbf{f}(t) - \int_a^b \mathbf{K}(t, s, \mathbf{f})\, ds = \mathbf{g}(t)$$

or, in expanded form,

$$f_1(t) - \int_a^b K_1(t, s, f_1(s), ..., f_m(s))\, ds = g_1(t)$$

$$...$$

$$...$$

$$f_m(t) - \int_a^b K_m(t, s, f_1(s), ..., f_m(s))\, ds = g_m(t)$$

for (known) continuously differentiable kernel functions, $K_1(t, s, \mathbf{f})$, ... , $K_m(t, s, \mathbf{f})$, and known right hand sides **g**(t), where $-\infty < a \le s, t \le b < \infty$.

INPUT

KIntEqns (Function): the kernels K_1, ... , K_m of the given system of integral equations

gIntEqns (Function): the right hand side g_1, ... , g_m of the given system of integral equations

leftEndPoint (Real Scalar): the left endpoint a of the interval on which the integral equations are defined

rightEndPoint (Real Scalar): the right endpoint b of the interval on which the integral equations are defined

numOfEqns (Integer Scalar): the number of integral equations

numUnifMeshPoints (Integer Scalar): the number of equally spaced points in the mesh on [a,b]

tolerance (Real Scalar): a positive tolerance for testing convergence of the solution.

maxIterations (Integer Scalar): maximum allowable iterations for computing the solution

defCorrFlag (Integer Scalar): an integer flag indicates that whether Richardson's extrapolation is desired (= 1) or not (= 0)

OUTPUT

solnPoints (Real Matrix): the m x n solution matrix containing the computed solution of the m given equations at the n points t_j, ... , t_n, i.e.,

$$F_{ij} = f_i(t_j)$$

meshPoints (Real Vector): the n-dimensional vector $(t_1, \ldots, t_n)^T$, containing the equally spaced mesh points on [a,b], i.e., $t_i = a+(i-1)h$ for i = 1, ... , n with h=(b-a)/(n-1)

EXAMPLE

Solve the Fredholm integral equation of the second kind:

$$f(t) + \int_0^1 \left[t\ s\ f^2(s)\right] ds = \sin(t) + t\frac{3 - 2\sin(2) - \cos(2)}{8}$$

The exact solution of this system is: f(t) = sin(t).

Example Script:

```
project KIntEqns, gIntEqns;

local leftEndPoint, rightEndPoint, numOfEqns, numUnifMeshPoints,
tolerance, maxIterations, defCorrFlag;

//----    Compute solution without deferred correction  -----
//

leftEndPoint = 0;
rightEndPoint = 1;
numOfEqns = 1;
numUnifMeshPoints = 11;
tolerance = 1.0e-11;
maxIterations = 125;
defCorrFlag = 0;


[solnPoints1, meshPoints] = intEqnFredholm(KIntEqns, gIntEqns, leftEnd-
Point, rightEndPoint, numOfEqns, numUnifMeshPoints, tolerance, maxIter-
ations,
defCorrFlag);
//-----    Compute solution with  deferred correction  ------
//
// defCorrFlag = 1;

//[solnPoints2, meshPoints] = intEqnFredholm(KIntEqns,
//gIntEqns,  leftEndPoint, rightEndPoint, numOfEqns,
//numUnifMeshPoints, tolerance, maxIterations, defCorrFlag);
function KIntEqns(i,t,s,f)
  select i from
```

```
    case 1:
           return (-(t*s*f[1]^2));
  end select;
end function;
function gIntEqns(i,t)
  select i from
    case 1:
           return (sin(t) + t*(3 - 2*sin(2) - cos(2))/8.0);
  end select;
end function;
```

ALGORITHM AND COMMENTS

The Nystrom method we use discretizes and subsequently solves the given system of integral equations by approximating the integrals using the standard trapezoidal rule in the following manner.

Let $a = t_1 < ... < t_n = b$ be n equally spaced points on [a,b], i.e., $t_i = a+(i-1)h$ and $h = (b-a)/(n-1)$, then replace the given system of integral equations by:

$$f_1(t_1) - \frac{h}{2}\sum_{i=1}^{n-1}[K_1(t_1, t_i, f_1(t_i), ..., f_m(t_i)) + K_1(t_1, t_{i+1}, f_1(t_{i+1}), ..., f_m(t_{i+1}))] = g_1(t_1)$$

...

$$f_m(t_1) - \frac{h}{2}\sum_{i=1}^{n-1}[K_m(t_1, t_i, f_1(t_i), ..., f_m(t_i)) + K_m(t_1, t_{i+1}, f_1(t_{i+1}), ..., f_m(t_{i+1}))] = g_m(t_1)$$

$$f_1(t_2) - \frac{h}{2}\sum_{i=1}^{n-1}[K_1(t_2, t_i, f_1(t_i), ..., f_m(t_i)) + K_1(t_2, t_{i+1}, f_1(t_{i+1}), ..., f_m(t_{i+1}))] = g_1(t_2)$$

...

$$f_m(t_2) - \frac{h}{2}\sum_{i=1}^{n-1}[K_m(t_2, t_i, f_1(t_i), ..., f_m(t_i)) + K_m(t_2, t_{i+1}, f_1(t_{i+1}), ..., f_m(t_{i+1}))] = g_m(t_2)$$

...

...

$$f_1(t_n) - \frac{h}{2}\sum_{i=1}^{n-1} [K_1(t_n, t_i, f_1(t_i), ..., f_m(t_i)) + K_1(t_n, t_{i+1}, f_1(t_{i+1}), ..., f_m(t_{i+1}))] = g_1(t_n)$$

...

$$f_m(t_n) - \frac{h}{2}\sum_{i=1}^{n-1} [K_m(t_n, t_i, f_1(t_i), ..., f_m(t_i)) + K_m(t_n, t_{i+1}, f_1(t_{i+1}), ..., f_m(t_{i+1}))] = g_m(t_n)$$

which is a system of mn equations with mn unknowns $f_i(t_j)$, for $1 \le i \le m$, $1 \le j \le n$.

The unknowns $f_i(t_j)$ are computed using the Newton method (for solving nonlinear system of equations) with an automatically chosen initial guess $g_i(t_j)$.

Comments :

(1) The given kernels K_1, ... , K_m must be "smooth" (in the sense of their continuous differentiability) in order to apply the method successfully. The accuracy of the solution computed by this algorithm is determined by the smoothness of the kernel and the step size h. Theoretically, if K_1, ... , K_m are all twice continuously differentiable with respect to all their arguments, the Nystrom method using the trapezoidal rule will have an (asymptotically) $O(h^2)$ convergence rate.

(2) The $O(h^2)$ accuracy of the solution may be improved to result in an $O(h^4)$ approximation by computing the solution at 2n-1 equally spaced points on [a,b], followed by Richardson's extrapolation procedure. The function will automatically perform such a process when the input parameter defCorrFlag = 1. Note that since Richardson's extrapolation procedure is based on an asymptotic discretization error analysis, it is valid only for "small" step sizes and kernels usually not containing narrow high peaks (or deep ditches). Otherwise, the results from the Richardson's extrapolation may not be any better than the original n-point solution from the Nystrom method. Since the computation of the approximation at 2n-1 points requires solving a system of (2n-1)m equations in (2n-1) unknowns, it can be rather time consuming for certain nonlinear integral equations (for the definition of linear integral equations, see (5) below).

(3) This method we use has been developed based on the assumption that the given system of integral equations has a unique solution. The choice of an initial guess $f_i(t_j)=g_i(t_j)$ for the Newton method results from the constructive proof of existence and uniqueness of the solution using successive approximations. If there does not exist a solution for the given system of equations, the Newton method will fail to converge and no approximation will be produced. In the other case, when the given system of integral equations has more than one solution, the Newton method will usually converge to one of the solutions, which, however, may not be the one the user expects.

(4) The Jacobian matrix used in the Newton method is computed automatically by using the central difference approximation formula.

(5) When $K_i(t,s,f_1(s), ... , f_m(s)) = K_{i,1}(t,s)\, f_1(s) + ... + K_{i,m}(t,s)\, f_m(s)$ for all i = 1, ... , m, the system of the integral equations is called linear. It is easy to see that after discretization, the approximation of $f_i(t_j)$ is the solution of a system of mn linear equations. Consequently, the Newton method will take no more than two iterations to obtain the solution (it is possible to take a second iteration in order to test convergence).

(6) The accuracy, tolerance, used to determine the convergence of Newton method should not be larger than O(h) in order to avoid the unnecessary loss of accuracy in the computed solution of the system of integral equations. On the other hand, a value of tolerance that is too small compared with h will not increase the accuracy of the solution for the original integral equations, it will only result in spending an unnecessary amount of computational effort.

(7) The default values for the error tolerance, tolerance, and maximum number of iterations, maxIterations, for the Newton method should usually be set to the square root of μ and a value between 10 and 100, respectively, where μ is the machine precision.

REFERENCES

Delves, L.M. and Mohamed, J.L., *Computational Methods for Integral Equations*, Cambridge University Press, Cambridge, 1985, pp. 85-88, 99-100.

■ intEqnVolt1

FUNCTION

[solnPoints, meshPoints] = intEqnVolt1(KIntEqns, gIntEqns, leftEndPoint, rightEndPoint, numOfEqns, numUnifMeshPoints, tolerance, maxIterations, defCorrFlag, intEqnVector);

PURPOSE

Compute the solution **f**(t) of a system of m Volterra integral equations of the first kind:

$$\int_a^t \mathbf{K}(t, s, \mathbf{f})\, ds = \mathbf{g}(t)$$

or, in expanded form,

$$\int_a^t K_1(t, s, f_1(s), ..., f_m(s))\, ds = g_1(t)$$

$$...$$

$$...$$

$$\int_a^t K_m(t, s, f_1(s), ..., f_m(s))\, ds = g_m(t)$$

for (known) continuously differentiable kernel functions, $K_1(t, s, \mathbf{f})$, ... , $K_m(t, s, \mathbf{f})$, and known right hand sides **g**(t), where $-\infty < a \le s \le t \le b < \infty$.

INPUT

KIntEqns (Function): the kernels $K_1, \ldots, K_m$ of the given system of integral equations

gIntEqns (Function): the right hand side $g_1, \ldots, g_m$ of the given system of integral equations

leftEndPoint (Real Scalar): the left endpoint a of the interval on which the integral equations are defined

rightEndPoint (Real Scalar): the right endpoint b of the interval on which the integral equations are defined

numOfEqns (Integer Scalar): the number of integral equations

numUnifMeshPoints (Integer Scalar): the number of equally spaced points in the mesh on [a,b]

tolerance (Real Scalar): a positive tolerance for testing convergence of the solution.

maxIterations (Integer Scalar): maximum allowable iterations for computing the solution

defCorrFlag (Integer Scalar): an integer flag indicates that whether Richardson's extrapolation is desired (= 1) or not (= 0)

intEqnVector (Real Vector): the m-dimensional vector containing the initial guess for the first midpoint solution of $f_1, \ldots, f_m$

OUTPUT

solnPoints (Real Matrix): the m x (n-1) solution matrix containing the computed solution of the m given equations at the n points $t_i, \ldots, t_{n-1}$, i.e.,

$$F_{ij} = f_i(t_j)$$

meshPoints (Real Vector): the (n-1)-dimensional vector $(t_1, \ldots, t_{n-1})^T$, containing the equally spaced mesh points on [a,b], i.e., $t_i = a+(i-1/2)h$ for i = 1, ... , n-1 with h=(b-a)/(n-1)

EXAMPLE

Solve the Volterra integral equations of the first kind:

$$\int_0^t e^{(s-t)} f_1(s) + f_2(s)\, ds = 1 - e^{-t} + \frac{t^2}{2}$$

$$\int_0^t \frac{(t-s) f_2^2(s)}{1 + f_1^2(s)} ds = \frac{t^4}{24}$$

The exact solution of this system is: $f_1(t) = 1$, $f_2(t) = t$.

Example Script:

```
project KIntEqns, gIntEqns;

local leftEndPoint, rightEndPoint, numOfEqns, numUnifMeshPoints,
tolerance, maxIterations, defCorrFlag;
```

```
//----    Compute solution without deferred correction  -----
//

leftEndPoint = 0;
rightEndPoint = 1;
numOfEqns = 2;
numUnifMeshPoints = 11;
tolerance = 1.0e-11;
maxIterations = 12;
defCorrFlag = 0;
intEqnVector = {0.5, 0.5};

[solnPoints1, meshPoints] = intEqnVolt1(KIntEqns, gIntEqns, leftEnd-
Point, rightEndPoint, numOfEqns, numUnifMeshPoints, tolerance, maxIter-
ations,
defCorrFlag, intEqnVector);

//-----    Compute solution with  deferred correction  ------
//
defCorrFlag = 1;
[solnPoints2, meshPoints] = intEqnVolt1(KIntEqns, gIntEqns, leftEnd-
Point, rightEndPoint, numOfEqns, numUnifMeshPoints, tolerance, maxIter-
ations,
defCorrFlag, intEqnVector);

function KIntEqns(i,t,s,f)
  select i from
    case 1:
           return (exp(s-t)*f[1]*f[1] + f[2]);
    case 2:
           return ((t-s)*f[2]*f[2] /(1 + f[1]* f[1]));
  end select;
end function;

function gIntEqns(i,t)
  select i from
    case 1:
           return (1 - exp(-t)+t*t/2.0);
    case 2:
           return (t*t*t*t/24.0);
  end select;
end function;
```

ALGORITHM AND COMMENTS

The midpoint method discretizes a system of Volterra integral equations of the first kind by approximating the integrals using the standard midpoint rule and subsequently computing the solution in the following manner.

Let $a+h/2 = t_1 < ... < t_{n-1} = b-h/2$ be n-1 equally spaced midpoints on [a,b], i.e.,

t_i = a+(i-1/2)h and h = (b-a)/(n-1); then the original system of integral equations is replaced by:

$$h\sum_{j=1}^{i} K_1\left(t_i+\frac{h}{2}, t_j, f_1(t_j), ..., f_m(t_j)\right) = g_1\left(t_i+\frac{h}{2}\right)$$

...

...

$$h\sum_{j=1}^{i} K_m\left(t_i+\frac{h}{2}, t_j, f_1(t_j), ..., f_m(t_j)\right) = g_m\left(t_i+\frac{h}{2}\right)$$

which is a system of m equations in m unknowns $f_1(t_j)$, ... , $f_m(t_j)$ for each j=1, ... , n-1.

The unknowns $f_i(t_j)$ are computed successively for each t_j using the Newton method (for solving nonlinear systems of equations) with an initially entered guess $f_1(t_1)$, ... , $f_m(t_1)$ and an automatically chosen initial guess $f_i(t_{j-1})$, i.e., the computed solution at t_{j-1}, for j=2, ... , n-1.

Comments :

(1) The given kernels K_1,...,K_m must be smooth (in the sense of their continuous differentiability) in order to apply the method successfully. The accuracy of the solution computed by this algorithm is determined by the smoothness of the kernel and the step size h. Theoretically, if K_1, ... , K_m are all twice continuously differentiable with respect to all of their arguments, the midpoint method will have an (asymptotically) $O(h^2)$ convergence rate.

(2) The $O(h^2)$ accuracy of the solution may be improved to an $O(h^4)$ one by computing the solutions at the first 3n-1 of the 3n equally spaced midpoints on [a,b], followed by Richardson's extrapolation procedure. The intEqnVolt1 function will automatically perform such a process when the input parameter defCorrFlag = 1. Note that the computation of the approximation at 3n-1 points requires solving 3n-1 systems, each containing m equations in m unknowns. This can be rather time consuming for the case of nonlinear integral equations (for definition of linear integral equations, see (5) below).

(3) The method is developed based on the assumption that the provided system of integral equations has unique solution. However, even when the given system of integral equations has a unique solution, the discretized problem (resulting from replacing the integrals by the midpoint rule) may not have a unique solution. For example, in the case of a single linear integral equation (see (5) below for a definition) the discretized problem has a unique solution only if $K(t_i+h/2, t_i) \neq 0$ for all i=1, 2, ... , n-1. Moreover, even the discretized problem has a solution, the choice of an initial guess can still make the Newton method fail because of its non-global convergence property. In general, Volterra integral equations of the first kind are more difficult to solve than those of the second kind.

(4) The Jacobian matrix used in the Newton method is computed automatically by using the central difference approximation formula.

(5) When $K_i(t,s,f_1(s),...,f_m(s)) = K_{i,1}(t,s)\, f_1(s) + ... + K_{i,m}(t,s) f_m(s)$ for all i = 1, ... , m, the system of the integral equations is called linear. It is easy to see that, after discretization, the approximation of $f_i(t_j)$ is the

solution of the system of mn linear equations. Consequently the Newton method will take no more than two iterations to obtain the solution (it is possible to take a second iteration in order to test convergence).

(6) The accuracy, tolerance, used to determine the convergence of Newton method should not be larger than O(h) in order to avoid the unnecessary loss of accuracy in the computed solution of the system of integral equations. On the other hand, a value of tolerance that is too small compared with h will not increase the accuracy of the solution for the original integral equations, it will only result in spending an unnecessary amount of computational effort.

(7) The default values for the error tolerance, tolerance, and maximum number of iterations, maxIterations, for the Newton method should usually be set to the square root of μ and a value between 10 and 100, respectively, where μ is the machine precision.

REFERENCE

Linz, P., *Analytical and Numerical Methods for Volterra Equations*, SIAM, Philadelphia, 1985, pp. 144-145.

■ intEqnVolt2

FUNCTION

[solnPoints, meshPoints] = intEqnVolt2(KIntEqns, gIntEqns, leftEndPoint, rightEndPoint, numOfEqns, numUnifMeshPoints, tolerance, maxIterations, defCorrFlag);

PURPOSE

Compute the solution **f**(t) of a system of m Volterra integral equations of the second kind:

$$\mathbf{f}(t) - \int_a^t \mathbf{K}(t, s, \mathbf{f})\, ds = \mathbf{g}(t)$$

or, in expanded form,

$$f_1(t) - \int_a^t K_1(t, s, f_1(s), ..., f_m(s))\, ds = g_1(t)$$

$$...$$

$$...$$

$$f_m(t) - \int_a^t K_m(t, s, f_1(s), ..., f_m(s))\, ds = g_m(t)$$

for (known) continuously differentiable kernel functions, $K_1(t, s, \mathbf{f})$, ... , $K_m(t, s, \mathbf{f})$, and known right hand sides **g**(t), where $-\infty < a \le s \le t \le b < \infty$.

INPUT

KIntEqns (Function): the kernels K_1, ... , K_m of the given system of integral equations

gIntEqns (Function): the right hand side g_1, ... , g_m of the given system of integral equations

leftEndPoint (Real Scalar): the left endpoint a of the interval on which the integral equations are defined

rightEndPoint (Real Scalar): the right endpoint b of the interval on which the integral equations are defined

numOfEqns (Integer Scalar): the number of integral equations

numUnifMeshPoints (Integer Scalar): the number of equally spaced points in the mesh on [a,b]

tolerance (Real Scalar): a positive tolerance for testing convergence of the solution.

maxIterations (Integer Scalar): maximum allowable iterations for computing the solution

defCorrFlag (Integer Scalar): an integer flag indicates that whether Richardson's extrapolation is desired (= 1) or not (= 0)

OUTPUT

solnPoints (Real Matrix): the m x n solution matrix containing the computed solution of the m given equations at the n points t_i, ... , t_n, i.e.,

$$F_{ij} = f_i(t_j)$$

meshPoints (Real Vector): the n-dimensional vector $(t_1, \ldots, t_n)^T$, containing the equally spaced mesh points on [a,b], i.e., $t_i = a+(i-1)h$ for i = 1, ... , n with h=(b-a)/(n-1)

EXAMPLE

Solve the Volterra integral equation of the second kind:

$$f(t) - \int_{-1}^{1} \left(t - s - \frac{3}{2}\right)\sqrt{t-s}\, f(s)\, ds = e^t + \frac{(t+1)^{3/2}}{e}$$

The exact solution of this system is: $f(t) = e^t$.

Example Script:

```
project KIntEqns, gIntEqns;

local leftEndPoint, rightEndPoint, numOfEqns, numUnifMeshPoints,
tolerance, maxIterations, defCorrFlag;

//------    Compute solution without deferred correction  ---
//

leftEndPoint = -1;
rightEndPoint = 1;
```

```
numOfEqns = 1;
numUnifMeshPoints = 11;
tolerance = 1.0e-11;
maxIterations = 12;
defCorrFlag = 0;

   [solnPoints1, meshPoints] = intEqnVolt2(KIntEqns, gIntEqns, leftEnd-
Point, rightEndPoint, numOfEqns, numUnifMeshPoints, tolerance, maxIter-
ations, defCorrFlag);

//-----    Compute solution with  deferred correction  ------
//

defCorrFlag = 1;

   [solnPoints2, meshPoints] = intEqnVolt2(KIntEqns, gIntEqns, leftEnd-
Point, rightEndPoint, numOfEqns, numUnifMeshPoints, tolerance, maxIter-
ations, defCorrFlag);

function KIntEqns(i,t,s,f)
  select i from
    case 1:
     return ((t-s-1.5)*sqrt((t-s))*f[1]);
  end select;
end function;

function gIntEqns(i,t)
  select i from
    case 1:
     return (exp(t)+(t+1.0)^1.5/exp(1.0));
  end select;
end function;
```

ALGORITHM AND COMMENTS

The trapezoidal method discretizes a system of Volterra integral equations of the second kind by approximating the integrals using the standard trapezoidal rule and subsequently computing the solution in the following manner.

Let $a = t_1 < ... < t_n = b$ be n equally spaced points on [a,b], i.e., $t_i = a+(i-1)h$ and $h = (b-a)/(n-1)$, then the system of integral equations is replaced by:

$$f_1(t_i) - \frac{h}{2}\sum_{j=1}^{i-1} [K_1(t_i, t_j, f_1(t_j), ..., f_m(t_j)) + K_1(t_i, t_{i+1}, f_1(t_{j+1}), ..., f_m(t_{j+1}))] = g_1(t_i)$$

...

$$f_m(t_i) - \frac{h}{2}\sum_{j=1}^{i-1} [K_m(t_i, t_j, f_1(t_j), ..., f_m(t_j)) + K_m(t_i, t_{j+1}, f_1(t_{j+1}), ..., f_m(t_{j+1}))] = g_m(t_i)$$

which is a system of m equations in m unknowns $f_1(t_j)$, ... , $f_m(t_j)$ for each j=2, ... , n with known initial values $f_1(t_1) = g_1(t_1)$, ... , $f_m(t_1) = g_m(t_1)$.

The unknowns $f_i(t_j)$ are computed successively for each t_j using the Newton method (for solving nonlinear system of equations) with an automatically chosen initial guess $f_i(t_{j-1})$, i.e., the computed solution at t_{j-1}.

Comments :

(1) The given kernels K_1, ... , K_m must be smooth (in the sense of their continuous differentiability) in order to apply the method successfully. The accuracy of the solution computed by this algorithm is determined by the smoothness of the kernel and the step size h. Theoretically, if K_1, ... , K_m are all twice continuously differentiable with respect to all of their arguments, the trapezoidal method will have an (asymptotically) $O(h^2)$ convergence rate.

(2) The $O(h^2)$ accuracy of the solution may be improved to obtain an $O(h^4)$ approximation by computing the solution at 2n-1 equally spaced points on [a,b], followed by Richardson's extrapolation procedure. Note that the computation of the approximation at 2n-1 points requires solving 2n-1 systems, each containing m equations in m unknowns, so that it may result in a rather time consuming process in some cases - especially when m and n are both large.

(3) The continuously differentiable kernels of the given system of Volterra integral equations of the second kind guarantees the existence and uniqueness of the solution. For detailed information, see the reference listed below.

(4) The Jacobian matrix used in the Newton method is computed automatically by using the central difference approximation formula.

(5) When $K_i(t,s,f_1(s), ... , f_m(s)) = K_{i,1}(t,s)\, f_1(s) + ... + K_{i,m}(t,s)\, f_m(s)$ for all i=1, ... , m, the system of the integral equations is called linear. It is easy to see that after discretization, the approximation of $f_i(t_j)$, for each j, is the solution of a system of m linear equations. Consequently, the Newton method will take no more than two iterations to obtain the solution (it is possible to take a second iteration in order to test for convergence).

(6) The accuracy, tolerance, used to determine the convergence of Newton method should not be larger than O(h) in order to avoid the unnecessary loss of accuracy in the computed solution of the system of integral equations. On the other hand, a value of tolerance that is too small compared with h will not increase the accuracy of the solution for the original integral equations, it will only result in spending an unnecessary amount of computational effort.

(7) The default values for the error tolerance, tolerance, and maximum number of iterations, maxIterations,

for the Newton method should usually be set to the square root of μ and a value between 10 and 100, respectively, where μ is the machine precision.

REFERENCE

Linz, P., *Analytical and Numerical Methods for Volterra Equations*, SIAM, Philadelphia, 1985, pp. 52-53, 96.

■ odeBvpGenLinear

FUNCTION

[bvpMatrix, bvpVector] = odeBvpGenLinear (bvpFct, termFct, leftBoundConds, rightBoundConds, rightHandSide, start, finish, stepSize, maxStep, tolerance, shootAlgorithm, ivpType);

PURPOSE

Solve the boundary value problem for the nth-order linear system of differential equations:

$$\frac{d\mathbf{y}}{dx} = A(x)\,\mathbf{y} + \mathbf{q}(x) \qquad a < x < b$$

$$B_1\mathbf{y}(a) + B_r\mathbf{y}(b) = \beta$$

where $A(x) \varepsilon \mathbf{R}^{n \times n}$, $\mathbf{q}(x) \varepsilon \mathbf{R}^n$, $B_\mathbf{l}$, $B_\mathbf{r} \varepsilon \mathbf{R}^{n \times n}$, and $\beta \varepsilon \mathbf{R}^n$.

INPUT

bvpFct (Function): the coefficient matrix function A(x) of the boundary value problem (BVP)

guessFct (Function): the term vector function **q**(x)

leftBoundConds (Real Matrix): the left boundary condition matrix $B_\mathbf{l}$

rightBoundConds (Real Matrix): the right boundary condition matrix $B_\mathbf{r}$

rightHandSide (Real Vector): the right hand side vector **b**

start (Real Scalar): the left end of the interval (a, b)

finish (Real Scalar): the right end of the interval (a, b)

stepSize (Real Scalar): the output mesh stepsize

maxStep (Real Scalar): the maximum length of the shooting stepsize

tolerance (Real Scalar): the absolute error tolerance

shootAlgorithm (Integer Scalar): the type of shooting algorithm: Simple or Marching

ivpType (Integer Scalar): the initial value problem solver type

OUTPUT

bvpMatrix (Real Matrix): the solution matrix whose columns consist of each component of **y** at each independent variable mesh value

bvpVector (Real Vector): the independent variable mesh vector, each component of which is an output mesh step

EXAMPLES

Consider two problems, each described by the one dimensional ordinary differential equation:

(p(x) y')' - q(x) y = f(x).

(1) For a string that is stretched in the transverse direction, we have p = tension = constant, q = 0, f = (orthogonal) force distribution along the string, and we solve for y = transverse displacement at x. (2) For a longitudinally stretched elastic bar, we have p = ES, where E is Young's modulus and S is the cross-section area of the bar, q = 0, f = force along the bar, and y = displacement of a point along the bar.

We thus have the linear system:

y[1]' = y[2]

y[2]' = (q/p) y[1] - (p'/p) y[2] + f/p

subject to the boundary conditions:

y[1] = 0 at x = 0 and y[1] = 0 at x = 1.

Example Script:

```
// One-Dimensional String and Bar Displacement Equation
//
// We have a system of the type:
//
//  y' = bvpFct*y + termFct,
//  leftBoundConds*y(0) + rightBoundConds*y(1) =
//  rightHandSide

local mode, leftBoundConds, rightBoundConds, rightHandSide;

// Next two functions may be entered by the user.
// They are physical parameters.

// 1. distribution of force:
function force(t)
   x = t; return x*x;
end function;

// 2. either identical to 1, or ES:
function p(t)
   x = 1.0;
   return x;
end function;

// Prepare the main arguments:

function bvpFct(t)
  x = t;
  dstep = 0.000000001;
```

```
   A[1,1] = 0.0; // coeff. of the first eq.
   A[1,2] = 1.0;// coeff. of the first eq.
   A[2,1] = 0.0;// second eq. : q=0;
// next is -p'/p:
  A[2,2] = - (p(x + dstep) - p(x))/(dstep*p(x));
  return A;
end function;

function termFct(t)
  x = t;
   F[1] = 0.0;// RHS of the first eq.
   F[2] =   force(x)/p(x); // RHS of the second eq.
  return F;
end function;

leftBoundConds  = {1.0, 0.0; 0.0, 0.0};
rightBoundConds = {0.0, 0.0; 1.0, 0.0};
rightHandSide  =  {0.0; 0.0 };

// Prepare additional arguments:

start = 0.0;        // left end point
finish = 1.0;       // right end point
ivpType = <IV_BULR>;// Bulirsch-Stoer extrapolation code

// Two types of algorithms are available, SIMPLE & MARCH:

mode = getNumber("Enter 0 (default) for SIMPLE algorithm,
1 for MARCHing techniques", "0");

if mode = 1 then
   shootAlgorithm = <MARCH>;
else
   shootAlgorithm = <SIMPLE>;
endif;

// Enter the output mesh step:

stepSize = getNumber("Enter the output mesh step. Default is 0.1",
"0.1");

// Enter the accuracy:

digits = getNumber("Enter the number of digits for absolute tolerance
(default is 6, that means tolerance = 1.e-6)", "6");

tolerance = 10.0^(-digits);

// Enter the upper bound of shooting step:
```

```
maxStep = getNumber("Enter the max length of shooting step (the more
shootings - the better the result) Default is 1.0 (one shooting)",
"1.0");

// launch the ODE BVP Solver (and time the execution):

[bvpMatrix, bvpVector] =

odeBvpGenLinear (bvpFct, termFct, leftBoundConds, rightBoundConds,
rightHandSide, start, finish, stepSize, maxStep, tolerance, shootAlgo-
rithm, ivpType);

// PLOT the results:

StringGraph = new2DGraph("Stretched String");
StringPlot = new2DDataPlot("Force = x*x", bvpVector,
   bvpMatrix[*,1]);
addPlot(StringGraph, StringPlot);
```

ALGORITHM AND COMMENTS

This program uses the multiple shooting method to transform the given boundary problem:

$$\frac{d\mathbf{y}}{dx} = A(x)\,\mathbf{y} + \mathbf{q}(x) \qquad a < x < b$$

$$B_1\mathbf{y}(a) + B_r\mathbf{y}(b) = \beta$$

to a system of linear algebraic equations for parameters **s**[i], i ≥ 1. The solution on the interval (x[i], x[i + 1]) is assumed to be of the form:

$$\mathbf{y}(x) = \mathbf{Y}_i(x)\,\mathbf{s}[i] + \mathbf{v}[i](x)$$

where **Y**(x) is the matrix of fundamental solutions and **v**[i] is the inhomogeneous (particular) solution. The n columns of **Y**(x) and the vector **v**(x) can be computed as solutions of n + 1 initial value problems; initial conditions are imposed on $\mathbf{Y}_i$ and **v**[i] at each x[i]. The user has a choice of IVP solvers to use: a Bulirsch - Stoer extrapolation code (the default choice) or a variable step Runge - Kutta routine. The choice of algorithms for solving the above BVP includes either a simple shooting method or the use of a marching technique that reduces the dimension of the superposition parameter s[•] for the cases where the rank of either $B_{\mathbf{l}}$ or $B_{\mathbf{r}}$ is less than n.

Because of the length of details to explain our implementation, we refer the reader to the reference listed below. We have been faithful to the algorithmic development provided in the reference, but have used our own versions of IVP solvers and nonlinear system solving routines to implement the algorithms.

REFERENCE

Ascher, U.M., Mattheij, R.M.M., and Russell, R.D., *Numerical Solutions of Boundary Value Problems for Ordinary Differential Equations*, Prentice - Hall, 1988.

■ odeBvpGenNonlinear

FUNCTION

[bvpMatrix, bvpVector] = odeBvpGenNonLinear(bvpFct, NonLinBoundCondFct, guessFct, dimension, start, finish, stepSize, maxStep, tolerance, maxIterations, BndCndType);

PURPOSE

Solve the boundary value problem for the nonlinear system of differential equations:

$$\frac{d\mathbf{y}}{dx} = f(x, \mathbf{y}) \qquad a < x < b$$

$$g(\mathbf{y}(a), \mathbf{y}(b)) = 0$$

where **y**, **f**, and **g** ε $\mathbf{R}^n$.

INPUT

bvpFct (Function): the first-order system **f**(x, **y**) of differential equations

NonLinBoundCondFct (Function): the system **g**(**y**(a), **y**(b)) of nonlinear boundary condition functions

guessFct (Function): the initial guess for the solution **y**

dimension (Integer Scalar): the dimension n of the system

start (Real Scalar): the left end of the interval (a, b)

finish (Real Scalar): the right end of the interval (a, b)

stepSize (Real Scalar): the output mesh stepsize

maxStep (Real Scalar): the maximum length of the shooting stepsize

tolerance (Real Scalar): the absolute error tolerance

maxIterations (Integer Scalar): the maximum number of iterations

BndCndType (Integer Scalar): the type (linear or nonlinear) of boundary conditions

OUTPUT

bvpMatrix (Real Matrix): the solution matrix whose columns consist of each component of **y** at each independent variable mesh value

bvpVector (Real Vector): the independent variable mesh vector, each component of which is an output mesh step

EXAMPLES

This example solves a reduced Navier-Stokes equation that arises in describing the problem of fluid injection through one side of a long vertical channel. The reduced Navier-Stokes equation to be solved is:

$$f''' - R[(f')^2 - f f''] + RA = 0,$$

subject to the boundary conditions:

$$f(0) = f'(0) = 0, f(1) = 1, f'(1) = 0.$$

A is a constant that may be eliminated by adding another differential equation and R is the Reynolds number whose value is chosen by the user. We may convert this problem to the first order system:

$$x[1]' = x[2]$$

$$x[2]' = x[3]$$

$$x[3]' = R(x[2]^2 - x[1]\,x[3] - x[4])$$

$$x[4]' = 0$$

with the boundary conditions x[1] = x[2] = 0 at t = 0 and t = 1.

Example Script:

```
// Flow in a Channel Problem

project FctCalls, Time, R, bvpMatrix, bvpVector;

// the right hand side of the ODE system is given by:
//        x' = bvpFct(t, f)
// where R is the Reynolds number.

function bvpFct(t,x)
   project FCalls,  R;
   FCalls = FCalls + 1;
   if (FCalls > 1000000) then
      error ("Too many function calls");
   endif;

   dydx[1] = x[2];
   dydx[2] = x[3];
   dydx[3] = R * ( x[2]*x[2] - x[1]*x[3] - x[4] );
   dydx[4] = 0.0;

   return dydx;
end function;

// the nonlinear boundary condition function is:
// NonLinBoundCondFct(x0, x1) = 0 ( x0 = x[1] at t0 ,
//  x1 = x[1] at t1)

function NonLinBoundCondFct(x0, x1)
   BC[1] = x0[1];
   BC[2] = x0[2];
   BC[3] = x1[1] - 1.0;
   BC[4] = x1[2];
   return BC;
end function;

// the guess of the solution: at first (R=16) is trivial,
// then utilize the previous solution, and interpolate it.
```

```
// The initial guess function is:

function guessFct(t)
   project  R;

   if R <= 16 then
      gs[1] = 0.0;
      gs[2] = 0.0;
      gs[3] = 0.0;
      gs[4] = 0.0;
   else
      if R <= 64.0 then  z = 20.0;
      else
         if R <= 128 then  z = 29.0;
         else
            if R <= 256 then  z = 40.0;
            else
               if R <= 512 then z = 56.0;
               else
                  if R <= 1024 then z = 79.0;
                  else
                     if R <= 2048 then z = 111.0;
                     end if;
                  end if;
               end if;
            end if;
         end if;
      end if;
   end if;

// else if (L <= 4096) then z = 157.0;
// else if (L <= 6000) then z = 189.0;
// else  z = 200.0;

   gs[1] = t;
   gs[2] = 1.5 - 6*(t-0.5)*(t-0.5);
   gs[3] = 1.0/(10.0*t + 1/z) - 0.4 - 2.3*t;
   gs[4] = 2.5;

   return gs;
end function;

local
dimension, // dimension of the system
start,       // starting point of the interval
finish,   // finishing point  of the interval
stepSize,    // output mesh step (8 mesh intervals)
maxStep,  // max length of shooting interval
tolerance,// tolerance parameter
```

```
maxIterations,// limitation on the number of iterations
BndCndType ;// indicate the type of Boundary Conditions

dimension = 4;
start = 0.0;
finish = 1;
stepSize = .125;
maxStep = .0625;
tolerance = 1.e-7;
maxIterations = 64;
BndCndType = <NlBvsLIN_BC>;

R = getNumber("Enter the output mesh step. Default is 512.0", "512.0");
// Set R = 64 for a fairly short run time!

FCalls = 0;
time=timer(<zero>);
time=timer(<start>);

[bvpMatrix, bvpVector] =

odeBvpGenNonLinear(bvpFct, NonLinBoundCondFct, guessFct,
dimension, start, finish, stepSize, maxStep, tolerance,
maxIterations, BndCndType);
```

ALGORITHM AND COMMENTS

This program uses an algorithm whose major iterative loop consists of performing pseudo-Newton iterations. If the interval (a, b) is divided at the points x[i], i = 1, ... , N, where N is the number of points in the mesh, we may seek a solution **y**(x) which can be determined as a solution of an initial value problem on the interval (x[i], x[i + 1]) with the initial conditions:

$$y(x[i]) = s[i], \qquad i = 1, ..., N$$

The s[i], i ≥ 1, are called shooting parameters. In order to have a continuous solution and to satisfy the boundary conditions, the following set of N nonlinear equations have to be solved:

$$F(s) = 0$$

formed from the nonlinear boundary condition functions **g** at the shooting parameter values.

For each major iteration, we perform a Newton update of **s**:

$$s_{new} = s - \left(\frac{1}{J}\right)F(s)$$

where **J** is an approximation of the Jacobian of the function **F(s)**. To compute this value of **J** and to find the value of **F**, a number of IVPs need to be solved.

The aggregate function **F** is defined by the formulas:

$$F_i = \mathbf{s}[i + 1] - \mathbf{y}[i](x[i + 1], \mathbf{s}[i]),\ \ 1 \le i < N,$$

$$F_N = \mathbf{g}(\mathbf{s}[1], y_N(x[N+1], s[N+1])) .$$

The first N - 1 conditions are continuity conditions; the last arises from the boundary conditions. Except for the last row, the ith row of the n x n Jacobian matrix for F consists of the diagonal element $-Y_i(x_{i+1})$ and I in the i + 1 column, where $Y_i(x)$ is the n x n fundamental solution given by:

$$Y_i'(x) = A(x)Y_i ,\ \ x_i < x < x_{i+1} ,$$

$$Y_i(x_i) = I , 1 \le i \le N,$$

and I is the unit matrix. In the last row, we have B_a in the first column and $B_b Y_N(b)$ in the last column. Here, Y_i is (D**y**[i]/Ds)(x[i+1],s[i]), B_a = D(**g**)/Dy_l ,
B_b = D(**g**)/Dy_r . Omitting the index i for convenience, from ODE theory we know that Y = D**y**/Ds satisfies the equation:

$$Y' = (D(\text{bvpFct})/D\mathbf{y})(x, \mathbf{y}(x, s))Y.$$

We can approximately solve for the fundamental matrix Y by solving the initial value problem:

$$y_1' = \text{bvpFct}(x, y_1),\ y_1(x[i]) = s_1 = s + \text{eps}(0, 0, ..., 1, 0, ..., 0)$$

and use the fact that: $(y_1 - y)' = \text{bvpFct}(y_1) - \text{bvpFct}(y) \approx D(\text{bvpFct})/Dy*(y_1 - y)$. These computations are performed in the shooting routine.

To find $\delta = \mathbf{s}_{new} - \mathbf{s}$ in the Newton update, we either solve the specially banded linear system directly with or some other approach specific to this system. In our algorithm we use a specific approach that utilizes the method of parameter condensation. See the reference listed below.

The termination conditions for the major loop is:

1) Function tolerance is achieved;

2) if the s-step is smaller than the s-tolerance, then we have arrived at a fixed point and a message is returned indicating this;

3) if the iterations are divergent, the initial solution guess was probably bad - if there exists a solution - a message is returned for this as well.

As with the Linear BVP algorithm, we have been faithful to the algorithmic development provided in the reference, but have used our own versions of IVP solvers and nonlinear system solving routines to implement the algorithms.

REFERENCE

Ascher, U.M., Mattheij, R.M.M., and Russell, R.D., *Numerical Solutions of Boundary Value Problems for Ordinary Differential Equations,* Prentice - Hall, 1988.

■ odeIvpRKF

FUNCTION

[ivpMatrix,ivpVector] = odeIvpRKF(ivpEqns, initVector, start, finish, stepSize, relError, absError);

PURPOSE

Solve the initial value problem for the nth-order system of differential equations:

$$\frac{dx_1}{dt} = f_1(t, x_1, x_2, ..., x_n)$$
$$\frac{dx_2}{dt} = f_2(t, x_1, x_2, ..., x_n)$$
$$...$$
$$\frac{dx_n}{dt} = f_n(t, x_1, x_2, ..., x_n)$$

subject to the initial conditions: $x_1 = x_{10}$, $x_2 = x_{20}$, ... , $x_n = x_{n0}$ at $t = t_0$.

INPUT

ivpEqns (Function): the system of differential equations to be solved

initVector (Real Vector): the initial state vector (or initial solution estimate)

start (Real Scalar): the initial value of the independent variable t

finish (Real Scalar): the final value of the independent variable t

stepSize (Real Scalar): the time steps at which output (state vector) is displayed

relError (Real Scalar): the relative error estimate

absError (Real Scalar): the absolute error estimate

OUTPUT

ivpMatrix (Real Matrix): the matrix of solutions of the initial value problem (whose columns are the solution of the IVP at the output step values of the independent variable t)

ivpVector (Real Vector): the vector of independent variable step values

EXAMPLE

Solve the system of differential equations:

$$\frac{dx_1}{dt} = 2x_1 - \alpha x_1 x_2$$
$$\frac{dx_2}{dt} = -x_2 + \alpha x_1 x_2$$

subject to the initial conditions: $x_1(0) = 300$ and $x_2(0) = 150$. This system describes a classical ("Predator - Prey") model first proposed by Volterra to describe a simple ecosystem involving the interaction of two bio-

logical species, one the predator and the other the prey (food for the predator).

Problem Script:

```
project ivpEqns, initVector, ivpMatrix;

local start, finish, stepSize, relError, absError, alpha;
start = 0;
finish = 10;
stepSize = 0.5;
relError = 1.e-9;
absError = 0.0;
initVector = {300,150};
alpha = 0.01;

Function ivpEqns(t, x)
   alpha = 0.01;
   dxdt[1] = 2*x[1]-alpha*x[1]*x[2];
   dxdt[2] = -x[2]+alpha*x[1]*x[2];
   return dxdt;
end function;

[ivpMatrix,ivpVector] = odeIvpRKF(ivpEqns, initVector,
start, finish, stepSize, relError, absError);
```

ALGORITHM AND COMMENTS

This program is a popular implementation of Runge-Kutta formulas developed by E. Fehlberg in 1970. The essential idea behind the method is as follows. Two Runge-Kutta rules of different orders are computed; the first one, of the fifth order, involves six function evaluations $k_1, \ldots, k_6$ defined by:

$$k_i = h_n^* f\left(y_n + \sum_{j=1}^{i-1} \beta_{ij} k_i, t_n + \alpha_i h_n \right)$$

where i = 1, ... , 6. The value of y at step n+1 is then the weighted combination:

$$y_{n+1} = y_n + \sum_{i=1}^{6} \gamma_i k_i$$

The parameters b_{ij} , a_i , and g_i are computed from the Runge-Kutta order conditions that arise from matching the Taylor series expansion of the k_i with the Taylor series expansion of the exact local solution at t_{n+1}. There are numerous possible choices for these 27 parameters. Fehlberg was able to determine coefficients that started with a term including h_n^6 , i.e., produced a fifth order method. Although this 5th ordered method required six function evaluations, Fehlberg was able to make further use of these evaluations by finding a set of coefficients g_i^* such that four of these k's produce another, fourth order method:

$$y^*_{n+1} = y_n + \sum_{i=1}^{6} \gamma^*_i k_i$$

which enables the error estimate:

$$\sum_{i=1}^{6} (\gamma_i - \gamma^*_i)\, k_i$$

to be used for step size control. A complete FORTRAN code listing is provided in the References section.

REFERENCES

This program is a modified translation of a well tested version of RKF45, a Fehlberg 4th/5th order Runge-Kutta method written by H.A. Watts and L.F. Shampine at Sandia Laboratories. FORTRAN source is provided in: Forsythe, G.E., Malcolm, M.A., and Moler, C.B., *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, N.J., 1977, pgs. 135 - 147.

■ odeIvpSmooth

FUNCTION

[ivpMatrix,ivpVector] = odeIvpSmooth(ivpEqns, initVector, start, finish, stepSize, tolerance, seqType);

PURPOSE

Solve the initial value problem for the nth-order system of differential equations:

$$\frac{dx_1}{dt} = f_1(t, x_1, x_2, ..., x_n)$$
$$\frac{dx_2}{dt} = f_2(t, x_1, x_2, ..., x_n)$$
$$...$$
$$\frac{dx_n}{dt} = f_n(t, x_1, x_2, ..., x_n)$$

subject to the initial conditions: $x_1 = x_{10}$, $x_2 = x_{20}$, ... , $x_n = x_{n0}$ at $t = t_0$.

INPUT

ivpEqns (Function): the system of differential equations to be solved

initVector (Real Vector): the initial state vector (or initial solution estimate)

start (Real Scalar): the initial value of the independent variable t

finish (Real Scalar): the final value of the independent variable t

stepSize (Real Scalar): the length of output step

tolerance (Real Scalar): the estimated error tolerance

seqType (Integer Scalar): type of extrapolating sequence used in the algorithm (<romberg> or <bulrsqn>).

OUTPUT

ivpMatrix (Real Matrix): the matrix of solutions of the initial value problem (whose columns are the solution of the IVP at the output step values of the independent variable t)

ivpVector (Real Vector): the vector of independent variable step values

EXAMPLE

Solve the system of differential equations:

$$\frac{dx_1}{dt} = x_3$$

$$\frac{dx_2}{dt} = x_4$$

$$\frac{dx_3}{dt} = x_1 + 2x_4 - \frac{m_2(x_1 + m_1)}{((x_1 + m_1)^2 + x_2^2)^{3/2}} - \frac{m_1(x_1 - m_2)}{((x_1 - m_2)^2 + x_2^2)^{3/2}}$$

$$\frac{dx_4}{dt} = x_2 - 2x_3 - \frac{m_2 x_2}{((x_1 + m_1)^2 + x_2^2)^{3/2}} - \frac{m_1 x_2}{((x_1 - m_2)^2 + x_2^2)^{3/2}}$$

subject to the initial conditions: $x_1(0) = 0.994$, $x_2(0) = 0$, $x_3(0) = 0$, $x_4(0) = -2.00158510637908252240$. This system describes the motion of a satellite about the Earth-Moon system in a special orbit - the so-called "Arenstorf Orbit" - a restricted three-body orbit problem.

Problem Script:

```
project ivpEqns, initVector, ivpMatrix;

local start, finish, stepSize, tolerance, seqType;
start = 0;
finish = 17.06521656015796255889;
```

```
stepSize = 1;
tolerance = 1.e-10;
seqType = 1;
initVector = {0.994,0,0,-2.00158510637908};

Function ivpEqns(t, x)
   m1 = 0.012277471;
   m2 = 1.0-m1;
   D1 = ((x[1]+m1)^2+x[2]^2)^1.5;
   D2 = ((x[1]-m2)^2+x[2]^2)^1.5;
   dxdt[1] = x[3];
   dxdt[2] = x[4];
   dxdt[3] = x[1]+2*x[4]-m2*(x[1]+m1)/D1-m1*(x[1]-m2)/D2;
   dxdt[4] = x[2]-2*x[3]-m2*x[2]/D1-m1*x[2]/D2;
   return dxdt;
end function;

[ivpMatrix,ivpVector] = odeIvpSmooth(ivpEqns, initVector,
start, finish, stepSize, tolerance, seqType);
```

ALGORITHM AND COMMENTS

This code implements the extrapolation Bulirsch-Stoer-Gragg algorithm. Stepsize control and order selection are used. The accuracy of the computed solution is estimated very conservatively. Thus for relatively simple problems the results may well be more precise then required. This ODE IVP solver works well for really restrictive tolerances. It will be more effective then odeIvpRKF for the order of tolerance ≤ 1.e-8. It is also recommended if error estimation is of high importance. For the easier problem when tolerances ≥ 1.e-6 odeIvpRKF will be the first reasonable choice. The output of the function consists of the solution (matrix) and of the independent mesh points (vector):

ivpMatrix[m,j] = $y_j(t_m)$, ivpVector[m] = t_m,

where

$t_{m+1} - t_m \le$ stepSize, m = 1, 2, ….

To solve the system $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$, $\mathbf{y}(\mathbf{x}_0) = \mathbf{y}_0$, begin by selecting an initial step size h and initial order k. The following pseudocode illustrates the major iteration loop:

i = 1;

while (tolerance is achieved)

{

1. Compute y(x + h) using N=seq(i) subdivisions.

2. Extrapolate the results.

3. Monitor the results.

4. Correspondingly :

- reject the step, if i = k+1 and tolerance is not achieved,

or i ≤ k but there is no hope on former convergence;

- accept the results, if tolerance is achieved, and set

new step h and new k := k-1,k,k+1; -

continue with i = i + 1

}

These steps are described below.

1. At step 1 the modified midpoint rule of order 2 is used to compute solutions using n = sqn(i) subdivisions of the interval [x, x + h]. sqn[i] is the extrapolating sequence. Two choices are evident. One is (sqnType = <romberg>)

sqn[ ] = 2, 4, 8, 16, 32, sqn[i] = 2^i .

The other choice is (sqnType = <bulrsqn>) suggested by Bulirsch

sqn[ ] = 2, 4, 6, 8, 12, 16, 24, 32, sqn[i] = 2 sqn[i-2].

Some other sqn[ ] are possible, with only theoretical requirement that the Toeplitz condition ([3], p.508) sqn[i] / sqn[i+1] ≤ ß < 1 be fulfilled.

The computation by itself consists of the following steps

$$h_1 = \frac{h}{n}, \quad h_2 = 2h_1, \quad y_0 = y(x), \quad y(x+h) = y_n$$

$$y_1 = y_0 + h_1^* f(x_0, y_0)$$

$$y_{k+1} = y_{k-1} + h_2^* f(x_k, y_k), \qquad k = 1, ..., n-1$$

$$y_n = 0.5\,(y_n + y_{n-1} h_1^* f(x_n, y_n))$$

2. Generally speaking we computed some approximations to y(x+h) using different steps

$$h_k = \frac{h}{n_k} \to 0$$

Our goal is to guess the function F(0) = y(x + h), if $F(h_k)$ are known. Two options are available: to do the polynomial or rational interpolation. The polynomial extrapolation is based on the Aitken-Nevill algorithm, where the extrapolation table T is constructed:

$$T_{1,1}$$

$$T_{2,1}, T_{2,2}$$

...

$$T_{k,1}, T_{k,2}, ..., T_{k,k}$$

Here (n_k = seq[k])

$$T_{k,j} = T_{k,j-1} + \frac{T_{k,j-1} - T_{k-1,j-1}}{\frac{n_k^2}{n_{k-j+1}^2} - 1}$$

3. Step 3 is described in [2], p. 228 - 231. Its pseudo code is

if (tolerance is achieved)
{
Accept the results.
Recompute order k and step h.
i = 1;
}
else if (we hope for later convergence)
{
Compute the next extrapolation line i = i+1;
}

else
{
diminish the step
i = 1;
}

Now we need to go into some details. Let us introduce some notations:

$$\delta_{k,j} = \left| y(x+h) - T_{k,j} \right|$$

$$err_{k,j} = \left| T_{k,j} - T_{k,j+1} \right|$$

$$err_k - err_{k,k-1}$$

where some kind of (scaling) norm is used in $|\bullet|$.
It is well known that

$$\delta_{k,\,j} \approx e_j \theta_{kj} h^{2j+1}$$

$$\theta_{kj} = \frac{1}{n_k^2 n_{k-1}^2 \ldots n_{k-j+1}^2} \tag{1}$$

We will assume that

$$\delta_{k,\,k-1} \ll \delta_{k,\,k} \tag{2}$$

Then it easily derived that

$$\begin{aligned} err_k = err_{k,\,k-1} &= \delta_{k,\,k-1} \bmod (\delta_{k,\,k}) \\ &= e_{k-1} \frac{1}{n_k^2 \ldots n_2^2} h^{2k-1} \end{aligned} \tag{3}$$

Also

$$\varepsilon_k = \delta_{k,\,k} \approx e_k \frac{1}{n_k^2 \ldots n_1^2} h^{2k+1} \tag{4}$$

The hypothesis (2) is equivalent to the assumption that

$$\frac{e_k}{n_1^2} h^2 \ll e_{k-1} \tag{5}$$

It will always assumed to be fulfilled. The method of step-size control in [2] is based on estimation of the left hand side of (3). Then $T_{k,k}$ is taken as the line approximation to the solution if

$$err_k \le tolr \tag{6}$$

Where tolr is the tolerance. Formula (3) has two consequences:

3. 1. The first is that we are able to recompute the current step:

$$h_{k,\,new} = c_2 \left(c_1 \frac{tolr}{err_k} \right)^{\frac{1}{2k-1}} h \tag{7}$$

Some safety factors c_1, c_2 are included in the algorithm.

3. 2. The other is that we have some reasons to estimate the possibility of convergence at the next line k in case (6) is not satisfied.

$$err_{k+1} \approx e_k \frac{1}{n_{k+1}^2 \ldots n_2^2} h^{2k+1} = \frac{e_k h^2}{e_{k-1} n_1^2 n_{k+1}^2} \frac{n_1^2}{} e_{k-1} \frac{1}{n_k^2 \ldots n_2^2} h^{2k-1} \Rightarrow err_{k+1} \ll \frac{n_1^2}{n_{k+1}^2} err_k \text{ (by(5))} \quad (8a)$$

So we may hope for next line convergence if

$$\left(\frac{n_1}{n_{k+1}}\right)^2 err_k \le tolr \left(\frac{n_1}{n_{k+1}}\right)^2 err_k \le tolr \quad (8)$$

Both estimates (6) and (8) prove to be very reliable, even though they are too conservative and higher accuracy is achieved in typical problems.

When the tolerance is achieved a new order and step can be chosen. The bigger k is, the larger step h we can use. But they will require larger amount of work. Thus we choose new order k to minimize work per step:

$$k_{new} = \text{argmin}\left\{\frac{A_q}{h_{q,new}}\right\} \qquad q = k-1, k, k+1$$

where A_k is the number of function calls needed for k-order computation. When the new k is equal to k+1 the only basis available to predict the new h is empirical. We assume that work should not increase, so

$$h_{k+1,new} = h_{k,new} \frac{A_{k+1}}{A_k} \quad (9)$$

REFERENCES

(1) Stoer, J. and Bulirsch, R., *Introduction to Numerical Analysis,* Springer-Verlag, New York, 1980, chapter 7. (2) Hairer, E., et. al., *Solving Ordinary Differential Equations I,,* Springer - Verlag, New York, 1987, section II.9. (3) Deuflhard, P., Recent Progress in Extrapolation Metods for ODE, SIAM Review,v.27,no.4,-Dec.1985,p.505-535. (4) Fatunla, S.O., *Numerical Methods for Initial Value Problems in Ordinary Differential Equations,* Academic Press, San Diego, 1988, chapter 7;

■ odeIvpSmoothNEq

FUNCTION

[ivpMatrix,ivpVector] = odeIvpSmoothNEq(ivpEqns, initVector, start, finish, firstStep,stepSize, tolerance, seqType);

PURPOSE

Solve the initial value problem for the nth-order system of differential equations:

$$\frac{dx_1}{dt} = f_1(t, x_1, x_2, ..., x_n)$$
$$\frac{dx_2}{dt} = f_2(t, x_1, x_2, ..., x_n)$$
$$...$$
$$\frac{dx_n}{dt} = f_n(t, x_1, x_2, ..., x_n)$$

subject to the initial conditions: $x_1 = x_{10}$, $x_2 = x_{20}$, ... , $x_n = x_{n0}$ at $t = t_0$.

INPUT

ivpEqns (Function): the system of differential equations to be solved

initVector (Real Vector): the initial state vector (or initial solution estimate)

start (Real Scalar): the initial value of the independent variable t

finish (Real Scalar): the final value of the independent variable t

firstStep(Real Scalar);initial step size to be used

stepSize (Real Scalar): the maximum length of output step

tolerance (Real Scalar): the estimated error tolerance

seqType (Integer Scalar): type of extrapolating sequence used in the algorithm (<romberg> or <bulrsqn>).

OUTPUT

ivpMatrix (Real Matrix): the matrix of solutions of the initial value problem (whose columns are the solution of the IVP at the output step values of the independent variable t)

ivpVector (Real Vector): the vector of independent variable step values

EXAMPLE

Solve the system of differential equations:

$$\frac{dx_1}{dt} = x_3$$

$$\frac{dx_2}{dt} = x_4$$

$$\frac{dx_3}{dt} = x_1 + 2x_4 - \frac{m_2(x_1 + m_1)}{((x_1 + m_1)^2 + x_2^2)^{3/2}} - \frac{m_1(x_1 - m_2)}{((x_1 - m_2)^2 + x_2^2)^{3/2}}$$

$$\frac{dx_4}{dt} = x_2 - 2x_3 - \frac{m_2 x_2}{((x_1 + m_1)^2 + x_2^2)^{3/2}} - \frac{m_1 x_2}{((x_1 - m_2)^2 + x_2^2)^{3/2}}$$

subject to the initial conditions: $x_1(0) = 0.994$, $x_2(0) = 0$, $x_3(0) = 0$, $x_4(0) = -2.00158510637908252240$. This system describes the motion of a satellite about the Earth-Moon system in a special orbit - the so-called "Arenstorf Orbit" - a restricted three-body orbit problem.

Problem Script:

```
project ivpEqns, initVector, ivpMatrix;

local start, finish, stepSize, tolerance, seqType;
start = 0;
finish = 17.06521656015796255889;
firstStep = 0.05;
stepSize = 1;
tolerance = 1.e-10;
seqType = 1;
initVector = {0.994,0,0,-2.00158510637908};

Function ivpEqns(t, x)
   m1 = 0.012277471;
   m2 = 1.0-m1;
   D1 = ((x[1]+m1)^2+x[2]^2)^1.5;
   D2 = ((x[1]-m2)^2+x[2]^2)^1.5;
   dxdt[1] = x[3];
   dxdt[2] = x[4];
   dxdt[3] = x[1]+2*x[4]-m2*(x[1]+m1)/D1-m1*(x[1]-m2)/D2;
   dxdt[4] = x[2]-2*x[3]-m2*x[2]/D1-m1*x[2]/D2;
   return dxdt;
end function;

[ivpMatrix,ivpVector] = odeIvpSmoothNEq(ivpEqns,
initVector, start, finish, firstStep, stepSize, tolerance,
seqType);
```

ALGORITHM AND COMMENTS

odeIvpSmoothNEq uses the same algorithm as odeIvpSmooth except that the stepsize is variable, which significantly improves performance.

REFERENCES

(1) Stoer, J. and Bulirsch, R., *Introduction to Numerical Analysis*, Springer-Verlag, New York, 1980, chapter 7. (2) Hairer, E., et. al., *Solving Ordinary Differential Equations I*,, Springer - Verlag, New York, 1987, section II.9. (3) Deuflhard, P., Recent Progress in Extrapolation Metods for ODE, SIAM Review,v.27,no.4,-Dec.1985,p.505-535. (4) Fatunla, S.O., *Numerical Methods for Initial Value Problems in Ordinary Differential Equations*, Academic Press, San Diego, 1988, chapter 7;

■ odeIvpStiff

FUNCTION

[ivpMatrix,ivpVector] = odeIvpStiff(ivpEqns, initVector, start, finish, firstStep, stepSize, tolerance, solverType);

PURPOSE

Solve the initial value problem for the nth-order system of differential equations:

$$\frac{dx_1}{dt} = f_1(t, x_1, x_2, ..., x_n)$$
$$\frac{dx_2}{dt} = f_2(t, x_1, x_2, ..., x_n)$$
$$...$$
$$\frac{dx_n}{dt} = f_n(t, x_1, x_2, ..., x_n)$$

subject to the initial conditions: $x_1 = x_{10}$, $x_2 = x_{20}$, ... , $x_n = x_{n0}$ at $t = t_0$.

INPUT

ivpEqns (Function): the system of differential equations to be solved

initVector (Real Vector): the initial state vector (or initial solution estimate)

start (Real Scalar): the initial value of the independent variable t

finish (Real Scalar): the final value of the independent variable t

firstStep (Real Scalar): the initial stepsize to be used

stepSize (Real Scalar): the length of output step

tolerance (Real Scalar): the absolute error tolerance

solverType (Integer Scalar): type of nonlinear system solver used in the algorithm (<newton>, <brent>, or <quasiNewton>).

OUTPUT

ivpMatrix (Real Matrix): the matrix of solutions of the initial value problem (whose columns are the solution of the IVP at the output step values of the independent variable t)

ivpVector (Real Vector): the vector of independent variable step values

EXAMPLE

Solve the system of differential equations:

$$\frac{dx_1}{dt} = -0.04x_1 + 0.01x_2x_3$$

$$\frac{dx_2}{dt} = 400x_1 - 100x_2x_3 - 3000x_2^2$$

$$\frac{dx_3}{dt} = 30x_2^2$$

subject to the initial conditions: $x_1(0) = 1.0$, $x_2(0) = 0$, $x_3(0) = 0$. This system of "Reaction-Rate Equations" describes the concentrations of the reactants in a chemical reaction system as a function of the time from the start of the reaction.

Example Script:

```
project ivpEqns, initVector, ivpMatrix;
local start, finish, firstStep, stepSize, tolerance, solverType;
tolerance = 1.e-6;
start = 0;
finish = 40;
firstStep = 1;
stepSize = 1;
solverType = <newton>;
initVector = {1,0,0};

Function ivpEqns(t, x)
   dxdt[1] = -0.04*x[1]+0.01*x[2]*x[3];
   dxdt[2] = 400*x[1]-100*x[2]*x[3]-3000*x[2]*x[2];
   dxdt[3] = 30*x[2]*x[2];
   return dxdt;
end function;

[ivpMatrix,ivpVector] = odeIvpStiff(ivpEqns, initVector, start,finish,
firstStep, stepSize, tolerance, solverType);
```

ALGORITHM AND COMMENTS

This program is based upon the cyclic composite method due to Tendler, Bickart, and Picel for solving non-linear stiff systems of ordinary differential equations. It contains multi-step integration schemes up to the 7th order and utilizes step size and order control. odeIvpStiff is effective for stiff ODE systems when other ODE IVP solvers are not effective or fail to provide a result. For other situations it will be less effective. There are numerous applications of stiff systems in such fields as chemical kinetics. The output of the function consists of the solution (matrix) and of the independent mesh points (vector):

$$\text{ivpMatrix}[m, j] = y_j(t_m), \text{ivpVector}[m] = t_m$$

where

$$t_{m+1} - t_m \le \text{stepSize}, m = 1, 2, ...$$

The idea of the algorithm is to compute the solution by the multistep implicit method, utilizing different difference schemes. The cycles of three schemes are realized for all orders between one and seven. The implicit approach guarantees stability, but the non-linear equation has to be solved at each step. Step-order control is applied to optimize the performance and to control the local accuracy. After the major cycle the new order and new step are computed either to go ahead with the optimal parameters, or to refine unsatisfactory current results.

This is the outline of the algorithm where x_0, y_0 is the result computed at the last point:

While the end point is not reached the next 4 steps are performed:

1. For current p (1<=p<=7) and h (order and step), l = 3 multistep schemes are applied cyclically:

$$\sum_{j=-p+i}^{i} \alpha_{ij} y_j - h \sum_{j=1}^{i} \beta_{ij} f(x_j, y_j) = 0, \qquad i = 1, ..., 1 \tag{1}$$

We assume that previous needed y_i are already constructed. The equation (1) is implicit with respect to the last y_i. Thus a solver of nonlinear equations is needed. The nonlinear equation has a form

$$F(y_i) \equiv \lambda y_i - \mu f(x_i, y_i) + \gamma = 0 \tag{2}$$

where

$$\lambda = \alpha_{ii} \qquad \mu = h\beta_{ii} \qquad \gamma = \sum_{j=-p+i}^{i-1} \alpha_{ij} y_j - h \sum_{j=1}^{i-1} \beta_{ij} f(x_j, y_j)$$

The schemes are constructed so that p order of approximation is guaranteed. The examples of implicit rules for p =1,2 are

$$y_i - y_{i-1} - hf(x_i, y_i) = 0 \qquad \text{for } p = 1$$

$$3y_i - 4y_{i-1} + y_{i-2} - 2hf(x_i, y_i) = 0 \qquad \text{for } p = 2$$

2. For $i \geq 2$ we do error control by estimating the backward differences of (p+1)-order with respect to the given local tolerance. If i = 2 the convergence test has a form

$$\Delta^{p+1} y_i \leq Err_{i,p} tolr$$

where value Err is tabulated in [1], Table V, p. 353, for different p, and for i = 2, 3, 4, and backward difference operator

$$\Delta^{p+1} = (\Delta)^{p+1} \qquad \Delta y_k = y_k - y_{k-1}$$

It is possible to monitor the statistics concerning rejections to avoid suspiciously big steps. If rejection happens, the major iteration J will be memorized and some restrictions on the length of steps will be imposed on the next steps J+1, ... , J+J1.

3. The new step $h_{new} = \eta * h$ is calculated. We require the new step h to be some multiple of an integer fraction of the old h. If we ended cyclic loop successfully, the order p may be changed by 1 to p_{new}. If the step is rejected, order does not change: $p_{new} = p$. In the case of multiple rejections, we are especially conservative. The factor stepFactor is computed by the formula

$$\eta = \min\left\{\tau(p, p_{new}), const * \left(Err_{2, p_{new}} * z\right)^{\frac{1}{(p_{new}+1)}}\right\}$$

where

$$z = \frac{tolr}{\left\|\Delta^{p_{new}+1} y_{i_{last}}\right\|}$$

$$i_{last} = \begin{cases} 0, \text{ if step was rejected} \\ 1, \text{ if step was accepted} \end{cases}$$

$\tau(p, p1)$ are tabulated (see [1], Table VI, p. 354) and are limited by the requirement to have enough points "at the back", const is a safe factor.

4. If the cyclic loop ended successfully we output intermediate mesh point's results, if needed, and prepare for the next big iteration

$$x_0 = x_0 + lh \qquad p = p_{new} \qquad h = h_{new}$$

If the loop was terminated by error control, we will start with the same x_0, but with the smaller h. To proceed with the next big iteration we need now to have computed points for $j \geq -p + i_{last}$, that are used in (1). If the step h has been changed, these previous grid points are not available, and we have to recompute them. We do this by means of Hermite interpolation

REFERENCES

(1) Tendler, J.M., Bickart, T.A., and Picel, Z., "A Stiffly Stable Integration Process Using Cyclic Composite Methods", ACM Trans. of Math. Soft., v. 4, No. 4, Dec. 1978, pp. 339-368; (2) Enright, W.H., Hull, T.E., and Lindberg, B., "Comparing Numerical Methods for Stiff Systems of ODEs", BIT 15, 1975, pp. 10-48; (3) Enright, W.H. and Hull,T.E., "Comparing Numerical Methods for the Solution of Stiff Systems of ODEs Arising in Chemistry", in Lapidus L., Sciesser W.E. (editors), Numerical Methods for Differential Systems, 1976. (4) E. Hairer, G.Wanner, (HW) Solving Differential Equations II. Stiff and Differential-Algebraic Problems, Springer-Verlag,1991

■ odeIvpStiffNEq

FUNCTION

[ivpMatrix,ivpVector] = odeIvpStiffNEq(ivpEqns, initVector, start, finish, firstStep, stepSize, tolerance, solverType);

PURPOSE

Solve the initial value problem for the nth-order system of differential equations:

$$\frac{dx_1}{dt} = f_1(t, x_1, x_2, ..., x_n)$$
$$\frac{dx_2}{dt} = f_2(t, x_1, x_2, ..., x_n)$$
$$...$$
$$\frac{dx_n}{dt} = f_n(t, x_1, x_2, ..., x_n)$$

subject to the initial conditions: $x_1 = x_{10}$, $x_2 = x_{20}$, ... , $x_n = x_{n0}$ at $t = t_0$.

INPUT

ivpEqns (Function): the system of differential equations to be solved

initVector (Real Vector): the initial state vector (or initial solution estimate)

start (Real Scalar): the initial value of the independent variable t

finish (Real Scalar): the final value of the independent variable t

firstStep (Real Scalar): the initial stepsize to be used

stepSize (Real Scalar): the length of output step

tolerance (Real Scalar): the absolute error tolerance

solverType (Integer Scalar): type of extrapolating sequence used in the algorithm (<newton>, <brent>, or <quasiNewton>).

OUTPUT

ivpMatrix (Real Matrix): the matrix of solutions of the initial value problem (whose columns are the solution of the IVP at the output step values of the independent variable t)

ivpVector (Real Vector): the vector of independent variable step values

EXAMPLE

Solve the system of differential equations:

$$\frac{dx_1}{dt} = -0.04x_1 + 0.01x_2x_3$$

$$\frac{dx_2}{dt} = 400x_1 - 100x_2x_3 - 3000x_2^2$$

$$\frac{dx_3}{dt} = 30x_2^2$$

subject to the initial conditions: $x_1(0) = 1.0$, $x_2(0) = 0$, $x_3(0) = 0$. This system of "Reaction-Rate Equations" describes the concentrations of the reactants in a chemical reaction system as a function of the time from the start of the reaction.

Example Script:

```
project ivpEqns, initVector, ivpMatrix;
local start, finish, firstStep, stepSize, tolerance, solverType;
tolerance = 1.e-6;
start = 0;
finish = 40;
firstStep = 1;
stepSize = 1;
solverType = <newton>;
initVector = {1,0,0};

Function ivpEqns(t, x)
   dxdt[1] = -0.04*x[1]+0.01*x[2]*x[3];
   dxdt[2] = 400*x[1]-100*x[2]*x[3]-3000*x[2]*x[2];
   dxdt[3] = 30*x[2]*x[2];
   return dxdt;
end function;
```

```
[ivpMatrix,ivpVector] = odeIvpStiffNEq(ivpEqns, initVector, start,fin-
ish, firstStep, stepSize, tolerance, solverType);
```

ALGORITHM AND COMMENTS

odeIvpStiffNEq uses the same algorithm as odeIvpStiff except that the stepsize is variable, which significantly improves performance.

REFERENCES

(1) Tendler, J.M., Bickart, T.A., and Picel, Z., "A Stiffly Stable Integration Process Using Cyclic Composite Methods", ACM Trans. of Math. Soft., v. 4, No. 4, Dec. 1978, pp. 339-368; (2) Enright, W.H., Hull, T.E., and Lindberg, B., "Comparing Numerical Methods for Stiff Systems of ODEs", BIT 15, 1975, pp. 10-48; (3) Enright, W.H. and Hull,T.E., "Comparing Numerical Methods for the Solution of Stiff Systems of ODEs Arising in Chemistry", in Lapidus L., Sciesser W.E. (editors), Numerical Methods for Differential Systems, 1976. (4) E. Hairer, G.Wanner, (HW) Solving Differential Equations II. Stiff and Differential-Algebraic Problems, Springer-Verlag,1991

■ optBFGS

FUNCTION

[minValue, optVector] = optBFGS(optFct, dimension, initVector, tolerance, maxIterations, updateType);

PURPOSE

Determine a local minimum value of a function $f(\mathbf{x}) = f(x_1, x_2, \ldots, x_n)$, $\mathbf{x} \in \mathbf{R}^n$.

INPUT

optFct (String): the function f(x[1], x[2], ... , x[n]) to be minimized.

dimension (Integer Scalar): the dimension of the domain of the objective function f(x[1], x[2], ... , x[n])

initVector (Real Vector): the initial estimate vector

tolerance (Real Scalar): the error tolerance to determine convergence

maxIterations (Integer Scalar): the maximum number of iterations to be attempted

updateType (Integer Scalar): a flag indicating the kind of update formula to be used

OUTPUT

minValue (Real Scalar): the minimum value of the objective function
f(x[1], x[2], ... , x[n])

optVector (Real Vector): the optimizing vector corresponding to the minimum of f

EXAMPLE

Determine the minimum of the function:

$$(x_1 - 2)^4 + (x_1 - 2)^2 x_2^2 + (x_2 + 1)^2$$

and its location. The exact minimum value is zero; the optimizing vector is (2, -1).

Example Script:

```
project optFct;

local tolerance, maxIterations, dimension, updateType;
tolerance = 1.0e-10;
maxIterations = 200;
dimension = 2;
updateType = -1;
initVector = {1,1};

function optFct(x)
    return ((x[1]-2)^4+(x[1]-2)^2*x[2]^2+(x[2]+1)^2);
end function;

[minValue,optVector] = optBFGS(optFct, dimension, initVector,
 tolerance, maxIterations, updateType);
```

ALGORITHM AND COMMENTS

This unconstrained minimization method combines a Quasi-Newton method with the Broyden-Fletcher-Goldfarb-Shano (BFGS) rank-two update formula. Briefly, on the k-th iterative step, we have:

$$x_{k+1} = x_k - \lambda_k H_k g_k$$

where H_k is the inverse of Hessian matrix J_k of function f computed at x_k, or its analog, and $g_k = \nabla f(x_k)$. (In the classical Newton method for solving nonlinear systems J_k would be the Jacobian of f at x_k). In the Quasi-Newton method, instead of recomputing the Hessian (or its finite difference approximation) the "weaker" secant condition,

$$g_{k+1} - g_k = J_{k+1}(x_{k+1} - x_k)$$

is used to recompute the analog J_k of the Hessian. Letting: $y_k = g_{k+1} - g_k$ and $s_k = x_{k+1} - x_k$, we can write:

$$y_k = J_{k+1} s_k \qquad s_k = H_{k+1} y_k$$

Broyden established that among different classes of matrices there exists a J_{k+1} (or H_{k+1}) which is closest to J_k (or to H_k) in the Frobenius norm. This is the famous one-rank Broyden's update of J_k:

$$J_{k+1} = J_k + \frac{(y_k - J_k s_k)\, s_k^T}{\|s_k\|^2}$$

(the H_{k+1} closest to H_k is known as the "Bad Broyden's update"). For the class of positive symmetric matrices, the following formula was independently established by Broyden, Fletcher, Goldfarb, and Shanno to be:

$$J_{k+1} = J_k + \frac{y_k y_k^T}{\langle y_k, s_k \rangle} - \frac{J_k s_k s_k^T J_k}{\langle s_k, J_k s_k \rangle}$$

This is the BFGS two-rank update formula, < , > is the inner product. It may be obtained from the updating of the factor L_k in the Cholesky decomposition of J_k: $L_k L_k^T = J_k$. We choose the class of positive symmetric matrices, since it is natural to deal with positive J, if the Hessian satisfies the sufficiency conditions. If not, this approach at least prevents a singular J in the classical Newton's method. Since we need H rather than J, we prefer to give a direct formula for H_{k+1}. It may be derived from the above using Sherman-Morrison-Woodbury formula; it is:

$$H_{k+1} = H_k + \left(1 + \frac{\langle y_k, H_k y_k \rangle}{\langle y_k, s_k \rangle}\right)\frac{s_k s_k^T}{\langle y_k, s_k \rangle} - \frac{s_k y_k^T H_k + H_k y_k s_k^T}{\langle y_k, s_k \rangle}$$

for generalizations, consult reference (1).

In order to find the appropriate value of λ_k, we perform a linear search. We require that f(x) strictly decrease from step to step to ensure convergence. In older algorithms, exact minimization along a line was attempted; these have been found to be less effective. Instead of trying to minimize $F(\lambda) = f(x_k - \lambda s_k)$, we prefer to take big steps that will at least improve F a little bit. We impose the condition:

$$F(\lambda) < F(0) - \alpha\lambda\langle g_k, s_k \rangle$$

where $\alpha < 0.5$ (usually $\alpha = 0.0001$). If (1) is not satisfied, a better guess for λ is supplied by using quadratic interpolation and the minimum of this function is used for the next guess. See reference (2) for further details.

The algorithm terminates if the iterations are divergent or a fixed point is reached, if the gradient of f(x) in norm is less than the prescribed tolerance, or when the maximum number of iterations has been exceeded.

REFERENCES

1) Scales, L.E., *Introduction to Non-Linear Optimization*, Springer-Verlag, 1985, pp. 89 - 91; 2) Dennis, Jr., J.E. and Schnabel, R.B., *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, 1983.

■ optConGradient

FUNCTION

[minValue,optVector,optIterations] = optConGradient(optFct, initVector, tolerance, maxIterations);

PURPOSE

Determine a local maximum or minimum value of a function $f(\mathbf{x}) = f(x_1, x_2, \ldots, x_n)$, $\mathbf{x} \in \mathbf{R}^n$.

INPUT

optFct (String): the function f(x[1], x[2], ... , x[n]) to be minimized

n (Integer Scalar): the dimension of the domain of the objective function f(x[1], x[2], ... , x[n])

initVector (Real Vector): the initial estimate vector

tolerance (Real Scalar): the error tolerance to determine convergence

maxIterations (Integer Scalar): the maximum number of iterations to be attempted

OUTPUT

minValue(Real Scalar): the minimum value of the objective function f(x[1], x[2], ... , x[n])

optVector (Real Vector): the optimizing vector corresponding to the minimum of f

optIterations (Integer Scalar): the number of iterations required to obtain the minimum

EXAMPLE

Determine the minimum of the function:

$$100\,(x_2 - x_1^2)^2 + (1 - x_1)^2$$

and its location. The exact value of the minimum value is zero;

the optimizing vector is (1, 1).

Example Script:

```
project optFct;
local tolerance, maxIterations;
tolerance = 1.0e-10;
maxIterations = 200;
initVector = {-1.2,1};

function optFct(x)
    return (100*(x[2]-x[1]^2)^2 + (1-x[1])^2);
end function;

[minValue,optVector,optIterations] = optConGradient(optFct, initVec-
tor,
         tolerance, maxIterations);
```

ALGORITHM AND COMMENTS

The Conjugate Gradient method combines the quadratic approximation of f(**x**):

$$f(x) \approx f(x_0) + \sum_i \frac{\partial}{\partial x_i} f \delta x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2}{\partial x_i \partial x_j} f \delta x_i \delta x_j = f(x_0) + \nabla f(x_0) \cdot \delta x + \frac{1}{2} \delta x^T H(x_0) \delta x$$

where $\delta \mathbf{x} = \mathbf{x} - \mathbf{x}_0$, with the method of conjugate directions (two direction vectors $\mathbf{s}_i$ and $\mathbf{s}_j$ are conjugate if $\mathbf{s}_i^T H \mathbf{s}_j = 0$, where H is the Hessian matrix defined above) to pick efficient search directions $\mathbf{s}_k$ for the iterative line searches:

$$x_k = x_{k-1} + \alpha_k s_k, \qquad k = 1, 2, ...$$

required in the method. Besides the construction of mutually conjugate directions, the key to the method is not having to compute the Hessian matrix H.

The method's two fundamental theorems are:

1) For a n x n symmetric, positive definite H and an initial vector $\mathbf{g}_0$, define the vector sequences:

$$g_{k+1} = g_k - \beta_k H h_0 h_{k+1} = g_{k+1} + \gamma_k h_k \qquad k = 0, 1, 2, ...$$

where β_k and γ_k are chosen to make $\mathbf{g}_{k+1}$ orthogonal to its predecessor, $\mathbf{g}_{k+1} \cdot \mathbf{g}_k = 0$, and $\mathbf{h}_{k+1}$ conjugate to its predecessor, $\mathbf{h}_{k+1}^T H \mathbf{g}_k = 0$. Then this procedure produces **g**'s and **h**'s all mutually orthogonal and mutually conjugate, respectively.

2) If $\mathbf{g}_k = -\nabla f(\mathbf{x}_k)$, where f is given by the quadratic form above, and **g** and **h** are the vectors contained in the sequences in 1), then moving along the direction $\mathbf{h}_i$ from $\mathbf{x}_k$ to the local minimum of f located at $\mathbf{x}_{k+1}$ and setting $\mathbf{g}_{k+1} = -\nabla f(\mathbf{x}_{k+1})$ produces the same vector as given by the sequence above without the construction of H.

The algorithm based on these results follows.

Algorithm Description:

Given f(**x**), dimension n;

Enter **x**0, tolerance, maxIterations;

{Note: in this description, **g** is the positive gradient: $\mathbf{g}_m = +\nabla f(\mathbf{x}_m)$}

Set stop = false;

k = 0;

repeat

for l = 0 to n-1 do

Set m = nk + l;

if l = 0 then

Set $b_m = 0$;

else

$b_m = \Delta\mathbf{g}_{m-1}^T \mathbf{g}_m / \mathbf{g}_{m-1}^T \mathbf{g}_{m-1}$; {Polak - Ribiere formula}

endif;

Set $\mathbf{s}_m = -\mathbf{g}_m + b_m\mathbf{s}_{m-1}$;

Compute a_m by linear search;

Set $\mathbf{x}_{m+1} = \mathbf{x}_m + a_m\mathbf{s}_m$;

if $\| \mathbf{g}_{m+1} \| <$ tolerance then

stop = true;

endif;

end for;

if $k \geq$ maxIterations then

stop = true;

endif;

until stop;

There is published evidence that the Polak - Ribiere choice gives considerably better results on certain classes of problems. It is suggested in reference (3) that this is due to an inherent ability for the Polak - Ribiere function to reset itself automatically to the steepest descent direction for those cases when $\mathbf{p}_k$ becomes nearly orthogonal to $-\mathbf{g}_k$.

The key advantage to the Conjugate Gradient method is its economy of storage; it requires only the storage of four n-dimensional vectors; other gradient methods require the storage of an n x n matrix or its lower triangle.

REFERENCES

This program implements the Polak variation of the Fletcher-Reeves conjugate gradient method provided in reference (1) and discussed further in (2): 1) Polak, E., *Computational Methods in Optimization*, Academic Press, New York, N.Y., 1971, section 2.3; 2) Jacobs, D.A.H., *The State of the Art in Numerical Analysis*, Academic Press, London, 1977, chapter 3; 3) Scales, L.E., *Introduction to Non-Linear Optimization*, Springer-Verlag, 1985, pp. 73 - 84.

■ optLinProg

FUNCTION

[optValue, optVector] = optLinProg(m1constraint, m2constraint, m3constraint, simpMatrix);

PURPOSE

Using the Simplex method, solve the following linear programming problem in standard form:

optimize

$$f(x) = c_1x + c_2x + ... + c_nx$$

subject to the constraints:

$$Ax = b \geq 0$$
$$x \geq 0$$

where A is an m x n matrix, m < n, and **b** is a nonnegative vector of dimension m.

INPUT

m1constraint (Integer Scalar): the number of ≤ inequality constraints

m2constraint (Integer Scalar): the number of ≥ inequality constraints

m3constraint (Integer Scalar): the number of equality constraints

simpMatrix (Real Matrix): the (N+1) x (M+1) dimensional matrix of Simplex Tableau coefficients

OUTPUT

optValue (Real Scalar): the minimum value of the linear objective function f

optVector (Real Vector): the optimizing vector corresponding to the minimum of f

EXAMPLE

Maximize the function f(**x**) = x_4 - x_5 subject to the constraints:

$$2x_2 - x_3 - x_4 + x_5 \geq 0$$
$$-2x_1 + 2x_3 - x_4 + x_5 \geq 0$$
$$x_1 - 2x_2 - x_4 + x_5 \geq 0$$
$$x_1 + x_2 + x_3 = 1$$

The exact value of the maximum is 0.0 at the optimizing vector (0.4, 0.2, 0.4, 0.0, 0.0).

Example Script:

```
project optValue, optVector, simpMatrix;

local m1constraint, m2constraint, m3constraint;
m1constraint = 0;
m2constraint = 3;
m3constraint = 1;

simpMatrix = {0, 0, 0, 0, 1, -1;
              0, 0, -2, 1, 1, -1;
              0, 2, 0, -2, 1, -1;
              0, -1, 2, 0, 1, -1;
              1, -1, -1, -1, 0, 0};
```

```
//  The first row of simpMatrix contains the coefficients of
//  the objective function; the first column contains the
//  right hand side constants in the constraints; the
//  rest of the columns (the next four rows underneath
//  the first) contain the negatives of the coefficients
//  in the inequality and equality constraints

[optValue, optVector] = optLinProg(m1constraint, m2constraint, m3con-
straint, simpMatrix);
```

ALGORITHM AND COMMENTS

This is a well established routine for solving linear programming problems using the simplex method. A standard linear programming problem is of the form:

maximize: $f = a_{11} + a_{12}x_1 + a_{13}x_2 + ... + a_{1,n+1}x_n$

subject to the constraints:

$a_{i2}x_1 + a_{i3}x_2 + ... + a_{i,n+1}x_n \leq a_{i1}$, $i = 2, ... , m_1 + 1$

$a_{i2}x_1 + a_{i3}x_2 + ... + a_{i,n+1}x_n \geq a_{i1}$, $i = m_1 + 2, ... , m_1 + m_2 + 1$

$a_{i2}x_1 + a_{i3}x_2 + ... + a_{i,n+1}x_n = a_{i1}$, $i = m_1 + m_2 + 2, ... , m_1 + m_2 + m_3 + 1$

$x_1 > 0,\ x_2 > 0,\ ...\ ,\ x_n > 0.$

Note that there are m+n constraint inequalities, n of them are non-negativity constraints (also called primary constraints) on the variables $x_1, x_2, ... ,x_n$ and the remaining are $m = m_1 + m_2 + m_3$ constraint inequalities.

A simplex tableau A of dimensions (m+1) x (n+1) is formed from the coefficients a(1,1), ... , a(1,n+1), a(2,1), ... , a(m+1,n+1) that converts the original problem above into one that is in *restricted normal form*, i.e., one that contains only equality constraints, each of which contains at least one variable with a positive coefficient that appears uniquely in that constraint. Any linear programming problem can be put into such a form. The conversion to restricted normal form requires the introduction of $m_1 + m_2$ slack variables (ignored upon output from the program) that convert the inequality constraints to equality constraints and a set of m artificial slack variables introduced into each of the m non-primary constraints. The solution of the linear programming problem with an auxiliary objective function formed from the introduction of the artificial slack variables then provides an initial feasible basic solution to the problem.

For the entry of the Simplex tableau in optLinProg(), we use the following format:

a) the first row in the tableau contains the coefficients of the objective function

b) the first column in the tableau contains the right hand side constants in the constraints

c) the rest of the columns in the tableau contain the negatives of the coefficients in the inequality and equality constraints, respectively.

The simplex method given below is then applied with the original objective function and corresponding equality constraints in restricted normal form. We have:

Simplex Algorithm Outline:

It is assumed the LP problem is in restricted normal form:

(1) maximize: $f = \mathbf{a}_{0.}{}^{T}\mathbf{x}$, subject to the constraints: $\mathbf{Ax} = \mathbf{a}_{.0}$, $\mathbf{x} \geq \mathbf{0}$,

where: $\mathbf{x}^{T}$ and $\mathbf{a}_{0.}{}^{T}$ are m+n component row vectors and **A** is the m x (m+n) simplex tableau. The coefficients a_{j0} are assumed to be non-negative.

Step 2. Construct the complete tableau. The system of equations in (1) are solved with respect to the slack variables $x_{n+1}, x_{n+2}, \ldots, x_{n+m}$, the value a_{00} of the objective function at this basic solution is computed, and the coefficients (valuation coefficients) α_{0i} of the original variables $x_1, x_2, \ldots, x_n$ in the objective function are evaluated. The complete tableau then consists of the a_{i0} , i = 1, ... , m, column plus the n+m columns of x variables along the top. The m+1 rows consist of the coefficients of the slack variable equations and the row of objective function coefficients. (See the reference below for the tables and other details)

Step 3. Check for optimality. Determine whether the second column of the tableau is optimal by evaluating the valuation coefficients α_{0i} for each variable x_i not in the basis. If all valuation coefficients α_{0i} are positive, the optimal solution has been reached; if all are strictly positive except for some zero values, then a number of equivalent optimal solutions exist. If one or more of the valuation coefficients α_{0i} are negative, the solution cannot be optimal and one or more of the iterations contained in the following steps must be performed.

Step 4. Locate the variable x_e entering the basis. Search for a variable with an index i with the properties: a) $x_i = 0$ in the tableau; b) the ith valuation coefficient is less than zero; c) at least one of the a_{ij} , j = 1, ... , m, in the x_i column is greater than zero. Choose that variable x_i corresponding to $\max_i (-\alpha_{0i})$. The column corresponding to x_e is called the pivot column.

Step 5. Determine the variable x_f leaving the basis. Determine the index f such that the quotient a_{j0}/a_{je} (formed for all j with $a_{je} > 0$) assumes its smallest value. The row corresponding to x_f is called the pivot row. a_{fe} is the pivot element.

Step 6. Transformation of the tableau. Perform the following elementary matrix operations to form a new tableau with transformed pivot element $a'_{fe} = 1$ and all other elements in the pivot column equal to zero: 1) divide all the elements of the pivot row by the pivot element a_{fe}; 2) subtract the appropriate multiples of the new pivot row from the remaining m-1 rows of the tableau to obtain zeros in the other elements of the pivot column; 3) interchange the columns of x_e and x_f and replace x_f in the lead column by x_e.

Step 7. Repeat steps 2 through 6 until step 2 indicates an optimum has been achieved.

REFERENCE

This simplex routine is algorithmically based on the program provided in: Kunzi, H. P., Tzschach, H. G., and Zehnder, C.A., *Numerical Methods of Mathematical Optimization*, Academic Press, 1971, chapter 1 and pgs. 84 - 91 and 115 - 118.

■ optNelderMead

FUNCTION

[minValue,optVector,optIterations] = optNelderMead(optFct, initVector, tolerance);

PURPOSE

Determine a local maximum or minimum value of a function $f(\mathbf{x}) = f(x_1, x_2, \ldots, x_n)$, $\mathbf{x} \varepsilon \mathbf{R}^n$.

INPUT

optFct (String): the function f(x[1], x[2], ... , x[n]) to be minimized

n (Integer Scalar): the dimension of the domain of the objective function f(x[1], x[2], ... , x[n])

initVector (Real Vector): the initial estimate vector

tolerance (Real Scalar): the error tolerance to determine convergence

OUTPUT

minValue(Real Scalar): the minimum value of the objective function f(x[1], x[2], ... , x[n])

optVector (Real Vector): the optimizing vector corresponding to the minimum of f

optIterations (Integer Scalar): the number of iterations required to obtain the minimum

EXAMPLE

Determine the minimum of the function:

$$x_1^2 + 2x_2^2 + x_3^2 - 2x_1x_2 + 2x_1 - \frac{5}{2}x_2 - x_3 + 3$$

and its location. The exact value of the minimum is 1.6875; the optimizing vector is (-3/4, 1/4, 1/2).

Example Script:

```
project optFct;

local tolerance;
tolerance = 1.0e-10;
initVector = {0,0,0};

function optFct(x)
      return (x[1]^2 + 2*x[2]^2 + x[3]^2 - 2*x[1]*x[2] + 2*x[1]
           - 2.5*x[2] - x[3] + 3);
end function;

[minValue,optVector,optIterations] = optNelderMead(optFct, initVector,
       tolerance);
```

ALGORITHM AND COMMENTS

The Downhill Nelder-Mead simplex search method is a procedure based on the heuristic concepts of the general minimization problem. Its strengths are that it requires no derivative information; only function evaluations.

A n-dimensional simplex is a structure consisting of n+1 points (called vertices - not necessarily all in the same plane) and all their interconnecting line segments. The Nelder-Mead simplex method implemented here is an iterative procedure which starts with a chosen initial point, say $P_0 = (p^0_1, \ldots, p^0_n)^T$, in R^n and then auto-

matically creates a set of n vertices $P_1, \ldots, P_n$ for the initial simplex, where:

$$P_i = P_0 + \mu \mathbf{e}_i$$

with $\mu = \max(1, \max |p^0_k|)/10$ and $\mathbf{e}_i$, the ith unit vector in R^n.

In each iteration, the method first tries to move the highest point of the simplex, i.e., the vertex where the function has the largest value, to a point having the lowest function value along the line passing through the centroid of the simplex excluding the highest point. This step may be done by one or more of four operations - reflection, expansion, reduction and contraction (see the Comments below for definitions). When it is possible to find a point, called P, which has a lower function value in this manner, the simplex is updated by replacing the highest point with P, and the iteration is repeated. On the other hand, when it fails to find such a point P, the method will perform a contraction on the simplex (i.e., for each vertex) through its lowest point (i.e., the vertex of the simplex with the lowest function value) before starting the next iteration.

The method converges when the highest point and lowest point differ in values by no more than a preassigned tolerance.

Comments :

1) The four operations: reflection, expansion, reduction and contraction, for a given

point X through the fixed point B in R^n, are defined as follows:

a) Reflection:

$$X_r = B + \alpha (B-X), \qquad \alpha \geq 1.$$

b) Expansion:

$$X_e = B + (\Delta-1) (B-X), \qquad \Delta > 1.$$

c) Contraction:

$$X_c = B -(1-\beta) (B-X), \qquad 0 < \beta \leq 1.$$

d) Reduction:

$$X_d = B - \gamma (B-X), \qquad 0 < \gamma \leq 1.$$

In our implementation of the Nelder-Mead method, we have chosen $\alpha = \gamma = 1$, $\beta=0.5$, and $\Delta=2$.

2) Since the underlying idea of the simplex method is somewhat heuristic, even when convergence occurs it is still possible to converge at a point which is not a minimum. If, for example, the n+1 vertices of the simplex are all in a one n-dimensional plane, the simplex can only move in n-1 dimensions in the n-dimensional space and may not be able to proceed towards the minimum. To reduce the possibility of such undesired convergence cases, we test the function value at the "supposed" (i.e., computed) minimum by comparing it to a set of nearby points generated by moving the "supposed" minimum along an axial direction by a "small" distance. If any function value is found lower than that of the "supposed" minimum, then the Nelder-Mead procedure is restarted.

3) The weakness of the Nelder-Mead simplex method is that the information provided by only function evaluations may not be used as effectively as those methods involving derivative evaluations. Consequently, it may take an unexpectedly large number of function evaluations to locate a solution within the preassigned tolerance. It is strongly suggested to not use the simplex method for higher dimensional (say $n > 5$) problems with high accuracy.

REFERENCE

Nash, J.C., *Compact Numerical Methods for Computers*, Adam Hilger, 1979, pp. 141-149.

■ optNonLinCon

FUNCTION

[minValue,optVector] = optNonLinCon(optFct, eqlConFct, ineqlConFct, dimension, eqlConFctDim, ineqlConFctDim, initVector, tolerance, maxIterations, maxFctCalls, optAlgType);

PURPOSE

Using an augmented Lagrange multiplier method, solve the general constrained optimization problem:

minimize

$$f(\mathbf{x}), \mathbf{x} = (x_1, x_2, ..., x_n)$$

subject to the constraints

$$\mathbf{h}(\mathbf{x}) = \mathbf{0} \qquad \mathbf{h} = (h_1, h_2, ..., h_n)$$
$$\mathbf{g}(\mathbf{x}) \leq \mathbf{0} \qquad \mathbf{g} = (g_1, g_2, ..., g_n)$$

INPUT

optFct (String): the function f(x[1], x[2], ... , x[n]) to be minimized

eqlConFunction (String): the equality constraint vector function
h = (h[1, **x**], h[2, **x**], ... , h[k, **x**])

ineqlConFunction (String): the inequality constraint vector function
g = (g[1, **x**], g[2, **x**], ... , g[l, **x**])

dimension (Integer Scalar): the dimension of the domain of the objective function f(x[1], x[2], ... , x[n])

ineqlConFctDim (Integer Scalar): the dimension, k, of the equality constraint function h = (h[1, x], h[2, x], ... , h[k, x])

eqlConFctDim (Integer Scalar): the dimension, l, of the equality constraint function g = (g[1, x], g[2, x], ... , g[l, x])

initVector (Real Vector): the initial estimate vector

tolerance (Real Scalar): the error tolerance to determine convergence

maxIterations (Integer Scalar): the maximum number of iterations to be attempted

maxFctCalls (Integer Scalar): the maximum number of function calls to be allowed

optAlgType (Integer Scalar): the type of unconstrained algorithm to be used

OUTPUT

minValue (Real Scalar): the minimum value of the objective function f(x[1], x[2], ... , x[n])

optVector (Real Vector): the optimizing vector corresponding to the minimum of f

EXAMPLE

Determine the minimum of the function:

$$(x_1 - 2)^2 + (x_2 - 1)^2$$

subject to the constraints:

$$h(x) = x_1 - 2x_2 + 1 = 0$$

$$g(x) = \frac{x_1^2}{4} + x_2^2 - 1 \leq 0$$

IMSL™ provides this example with the minimum value of 1.393 at the optimizing vector (0.8228, 0.9114).

Example Script:

```
project optFct, eqlConFct, ineqlConFct;

local dimension, eqlConFctDim, ineqlConFctDim, tolerance,
   maxIterations, maxFctCalls, optAlgType;

dimension = 2;
eqlConFctDim = 1;
ineqlConFctDim = 1;
tolerance = 1.0e-8;
maxIterations = 128;
maxFctCalls = 1024*128;
optAlgType = -2;
initVector = {2.0,2.0};

function optFct(x)
   return((x[1]-2)^2+(x[2]-1)^2);
end function;

function eqlConFct(x)
   h[1] = x[1]-2*x[2]+1;
   return(h);
end function;
function ineqlConFct(x)
   g[1] = (x[1]^2)/4.0+x[2]^2-1;
   return(g);
end function;

[minValue,optVector] = optNonLinCon(optFct, eqlConFct, ineqlConFct,
```

```
dimension, eqlConFctDim, ineqlConFctDim, initVector, tolerance,
maxIterations, maxFctCalls, optAlgType);
```

ALGORITHM AND COMMENTS

We solve the general constrained optimization problem:

minimize

$$f(\mathbf{x}), \mathbf{x} = (x_1, x_2, ..., x_n)$$

subject to the constraints

$$\mathbf{h}(\mathbf{x}) = \mathbf{0} \qquad \mathbf{h} = (h_1, h_2, ..., h_k)$$
$$\mathbf{g}(\mathbf{x}) \le \mathbf{0} \qquad \mathbf{g} = (g_1, g_2, ..., g_l)$$

using the Augmented Lagrangian Method (ALM). This involves trying to sequentially minimize a Lagrangian function of a special type. It is constructed by utilizing the Kuhn-Tucker necessary conditions and the feasibility requirements of the problem. Lagrangian multipliers in the Lagrangian are recomputed sequentially.

This method is the most advanced outgrowth of the penalty function developments of the past two decades.

The Augmented Lagrangian is defined by the formula:

$$L(x, \lambda, \rho) = f(\mathbf{x}) + \langle \lambda_h, \mathbf{h} \rangle + \rho \langle \mathbf{h}, \mathbf{h} \rangle + \langle \lambda_g, \mathbf{G} \rangle + \rho \langle \mathbf{G}, \mathbf{G} \rangle \qquad (1)$$

where: $\mathbf{h}(\mathbf{x})$ and $\mathbf{g}(\mathbf{x})$ are the equality and inequality constraint vectors, $\lambda=(\lambda_h,\lambda_g)$, λ_h and λ_g are the Lagrangian multiplier vectors (of dimension k and l, respectively), $\mathbf{G} = \mathbf{G}(\mathbf{x}) = \max \{ \mathbf{g}, -\lambda_g/2\rho \}$, ρ is a bounded weight (typically used in the exterior penalty method), and < , > indicates the natural inner product. Another name for the Augmented Lagrangian that reflects the history of the subject is Powell-Hestenes-Rockafellar penalty function. Rockafellar suggested the somewhat mysterious formula for **G**. It can be derived by adding slack variables **z** to the inequality constraints for **g** providing the equality constraints:

$$g(x) + \langle z, z \rangle = 0$$

thereby expanding the domain (size of the design variable space) but returning the problem to one for equality constraints (which the ALM was originally designed for). The minimization in **z** can be performed explicitly. The solution of the associated unconstrained minimization problem must be performed carefully because the Augmented Lagrangian has discontinuous 2nd derivatives at
$\mathbf{g}(\mathbf{x}) = -\lambda_g/2\rho$. We provide the user the choice of using either the Conjugate Gradient method or the BFGS method for the unconstrained minimization.

Theoretical considerations provide the following update formulas for the Lagrangian multipliers:

$$\lambda_g^{p+1} = \lambda_g^p + 2r_p G^p \qquad G_i^p = \max\left\{ g_i(x), \frac{-\lambda_{g,i}^p}{2\rho_p} \right\}, \qquad i = 1, ..., l$$
$$\lambda_h^{p+1} = \lambda_h^p + 2\rho_p h(x) \tag{2}$$

where ρ_p is increased as in the exterior penalty method, but a finite upper bound limits its value. The unconstrained minimization problem for the Augmented Lagrangian determines the values for ρ_p, λ_h and λ_g and the problem is repeated iteratively until convergence occurs.

We start with the values $\lambda^0 = 0$, $\rho_p = 1$. The algorithm halts if the tolerance for the gradient of L and the constraints has been achieved (convergence occurs) or if the maximum number of iterations has been exceeded.

Algorithm Outline:

Enter: x_0, f- and x-tolerances, maxIterations;

Set $\lambda^0 = 0$, $\rho_p = 1$, p = 0;

while p < maxIterations do

Do unconstrained minimization of the Augmented Lagrangian L(x, λ^p, ρ_p) given by formula (1);

If convergence occurs then exit from the loop;

Compute λ^{p+1} from λ^p using the formula (2);

Put $\rho_p = \min(\gamma \rho_p, \rho_{max})$, where $\gamma, \rho_{max} > 1$ are algorithm's constants;

end while;

The attractive features of this algorithm are:

1. This method is relatively insensitive to the values of ρ_p; thus we need not increase its value indefinitely.

2. One can accelerate the process through updating the Lagrangian multipliers.

3. One can obtain the exact vectors (to precision) x such that g(x) = 0 and h(x) = 0.

4. One may start with either a feasible or infeasible starting point.

5. The value of the nonzero Lagrangian multipliers in $\lambda_{g,i}$ automatically identify the active constraint set at the optimum.

REFERENCES

1) Vanderplaats, G.N., *Numerical Optimization Techniques for Engineering Design*, McGraw-Hill, 1984, Chap. 5, pp. 140-148; 2) Scales, L.E., *Introduction to Non-Linear Optimization*, Springer-Verlag, 1985, pp. 194-214; 3) Rockafellar, R.T., "A Dual Approach to Solving Nonlinear Programming Problems by Unconstrained Optimization," Math. Programming, **5**, 1973, pp. 354 - 373; 4) Fletcher, R., "Methods related to Lagrangian Functions," in: Gill, P.E. and Murray, W. (eds.), *Numerical Methods for Constrained Optimization*, Academic Press, 1974.

■ rootAnalytic

FUNCTION

[rootsVector, estimatedError] = rootAnalytic(analyticFct, radius, nOfPoints);

PURPOSE

Find all roots z of the complex analytic function f(z):

$$f(z) = 0$$

that lie inside the circle |z| < r.

INPUT

analyticFct(Function): the complex analytic function f(z)

radius(Real Scalar): the radius r of the circle

nOfPoints(Integer Scalar): specifies N the algorithmic parameter - the number of points that are used in the trapezoid rule (must be ≥ 8)

OUTPUT

rootsVector(Complex Vector): the vector of computed roots z

estimatedError (Real Vector): the estimated error in roots' evaluation

EXAMPLE

Find the roots of f(z) = (z^2 + 1)*sin(z) in { z: |z| < 4.0 }

```
project  nOfPoints, radius, estimError, roots;

radius = 4.0;
nOfPoints= 32;

[roots, estimError] = rootAnalytic(analyticFct, radius, nOfPoints);

function analyticFct(z)
    w = (z^2 + 1)* sin(z);
return (w);
```

ALGORITHM AND COMMENTS

The algorithm is based on the computation of the order and of the coefficients of a polynomial with the same roots as function f(z). Then the polynomial root solver is applied.

Let

$$P(z) = \sum_{0}^{n} a_k z^k \qquad a_n = 1$$

be a polynomial. If all its zeros lie inside D than after the multiplication by $P'(z)/(P(z)z^m)$, and integration, we have

$$m a_m = \sum_{m}^{n-1} a_k L_{k-m} \qquad m = 0, 1, ..., n-1 \tag{1}$$

where L_k is defined by the contour integral over ∂D :

$$L_k = \frac{1}{2\pi i} \int_{\partial D} \frac{P'(z)}{P(z)} z^k dz$$

If polynomial P(z) has the same zeros in D as the analytic function f(z) than the difference f'(z)/f(z) - P'(z)/P(z) is holomorphic in D+∂D. Hence its contour integral over ∂D is zero, and thus in evaluation of L_k we may use f(z) instead of P(z). If these coefficients are found we can find zeros of P(z) by standard means used for finding roots of polynomials. To compute L_k we can do the difference approximation of f'(z). But there is a better way to do this. Let D = { z: |z| < r }, F(t) = f(z), z = r exp(it). Then

$$L_k = \frac{1}{2\pi i} \int_{\partial D} \frac{f'(z)}{f(z)} z^k dz = \frac{-i}{2\pi} r^k \int_{0}^{2\pi} \frac{F'(t)}{F(t)} e^{ikt} dt$$

Therefore, L_k/r^k is a Fourier coefficient of the function

$$-i\frac{F'(t)}{F(t)} = \sum_{-\infty}^{\infty} \frac{L_k}{r^k} e^{-ikt} \qquad \text{for } 0 \le t < 2\pi$$

We will integrate this formula over [t - 2π/N, t] and use the existence of a primitive function ($k \neq 0$)

$$-g(t) \equiv \left(\ln \left[\frac{F(t)}{F\left(t - \frac{2\pi}{N}\right)} \right] = \sum_{k \neq 0} \frac{L_k}{k r^k} \left(e^{\frac{i2k\pi}{N}} - 1 \right) e^{-ikt} + \frac{2\pi i}{N} L[0] \right)$$

Finally after integration with respect to t, $0 \le t < 2\pi$ ($k \neq 0$)

$$n = L_0 = \frac{N}{2\pi i}\frac{1}{2\pi}\int_0^{2\pi} g(t)\,dt$$

$$\frac{L_k}{r^k} = \frac{k}{e^{i2k\pi/N} - 1}\frac{1}{2\pi}\int_0^{2\pi} g(t)\,e^{-ikt}dt$$

Here n is the order of polynomial P(z). Now the trapezoid rule yields

$$L_0 = \frac{1}{2\pi i}\sum_1^N g(t_j) \qquad t_j = \frac{2\pi j}{N}$$

$$\frac{L_k}{r^k} = \frac{k}{e^{i2k\pi/N} - 1}\sum_1^N g(t_j)\,e^{-ikt_j}$$

Now coefficients of the polynomial P(z) can be found from the linear system (1).

REFERENCES

M.P. Carpentier and A.F. Dos Santos, Solution of Equations Involving Analytic Functions, Journ. of Computational Physics, 45, 210-220 (1982)

■ rootBisection

FUNCTION

[realRoot, iterations] = rootBisection(rootFct, leftEndPoint, rightEndPoint, tolerance);

PURPOSE

Determine a real root of a single variable real-valued function f(x) (i.e., solve the equation f(x) = 0 for x) in a specified closed interval [a, b].

INPUT

rootFct (Function): the function f(x) for which a root is to be computed

leftEndPoint (Real Scalar): a, the left endpoint of the interval [a, b] containing the desired root.

rightEndPoint (Real Scalar): b, the right endpoint of the interval [a, b] containing the desired root.

tolerance (Real Scalar): the error tolerance used to test the convergence of the algorithm.

OUTPUT

realRoot (Real Scalar): the estimated value of the desired root in [a, b].

iterations (Integer Scalar): the positive integer indicating the number of iterations taken to obtain the computed root.

EXAMPLE

Compute the roots of the function: exp(-x) - x

Script Example:

```
project rootFct;
local leftEndPoint, rightEndPoint, tolerance;
leftEndPoint = 0;
rightEndPoint = 3;
tolerance = 1.0e-8;

function rootFct(x)
    return (exp(-x)-x);
end function;

[realRoot,iterations] = rootBisection(rootFct, leftEndPoint, rightEnd-
Point, tolerance);

//  Expected: realRoot =  0.567143290410095
```

ALGORITHM AND COMMENTS

The modified Dekker's method is a globally convergent algorithm for computing a real root of a single variable real-valued function f(x) in a prescribed interval [a,b] within which f(a), f(b) have opposite signs. This iterative method combines the certainty of bisection with the ultimate speed of the secant and inverse quadratic interpolation procedures.

In each iteration, three points x_1, x_2 and x_3 are considered (with x_1=a, x_2=b, and x_3 = c to start). Normally,

1. x_1 is the previous iterate.
2. x_2 is the latest iterate and closest approximation to the root.
3. x_3 is the previous or an older iterate so that $f(x_2)$ and $f(x_3)$ have opposite signs.

The x_1, x_2 and x_3 are updated for the next iteration by either bisection or by interpolation. Inverse quadratic interpolation is used whenever x_1, x_2 and x_3 are distinct, and linear interpolation (i.e., the secant method) is used whenever they are not. If the point obtained by interpolation is inside the current interval and essentially not too close to the endpoints, it is chosen; otherwise the bisection point is chosen.

The method always brackets the desired root during iterations. It terminates whenever the length $|x_2 - x_3|$ has been reduced to satisfy

$$|x_2 - x_3| \leq tol + 4^{\mu}|x_2|$$

where tol is the specified error tolerance and μ is the machine precision. In such cases, x_2 is the desired root.

Comments :

1) The computed root, say r, returned by the modified Dekker's method described above is such that the function f(x) is guaranteed to change sign in the approximate interval [r-2ß, r+2ß], where ß = tol/2 + $2^{\mu}|r|$.

2) In our program, to carry out the implementation of modified Dekker's method, we choose tol =0 as the default value for the error tolerance.

REFERENCES

Forsythe, G.E., Malcolm, M.A. and Moler, C.B., *Computer Methods for Mathematical Computations*, Prentice-Hall, 1977, pp.161-165.

■ rootMuller

FUNCTION

compRoot = rootMuller(rootFct, startPoint, fTolerance, xTolerance, maxIterations);

PURPOSE

Compute a real or complex root of the function f(z), i.e., find a complex solution of the equation f(z) = 0.

INPUT

rootFct (Function): the function f(z) for which a root is to be computed

startPoint (Complex Scalar): the initial complex value $z_0 = (x_0, y_0)$ that begin the search iterations

fTolerance (Real Scalar): the function error tolerance

xTolerance (Real Scalar): the x error tolerance

maxIterations (Integer Scalar): the maximum number of iterations to be used in the Muller search algorithm

OUTPUT

compRoot (Complex Scalar): the estimated complex root of f(z) = 0.

EXAMPLE

Compute the roots of the function: $f = (x^2 - 4x + 13)(x^2 + 10)$

Script Example:

```
project rootFct;
local startPoint, fTolerance, xTolerance, maxIterations;
startPoint = (1,1);
fTolerance = 1.0e-8;
xTolerance = 1.0e-8;
maxIterations = 100;

function rootFct(x)
    return ((x^2-4*x+13)*(x^2+10));
```

```
end function;

[compRoot] = rootMuller(rootFct, startPoint, fTolerance, xTolerance,
        maxIterations);

//  Expected Results:
//   For:startPoint = (1,-1):     compRoot:   2 -3i
```

ALGORITHM AND COMMENTS

This algorithm finds the real or complex roots of a general complex function f(z) by utilizing an iteration formula formed from a three-point Lagrangian interpolation formula and direct interpolation. It is essentially a generalization of the Secant method.

Given three initial approximations x_1, x_2, and x_3, the next approximation is computed by finding the complex zero of the second-order interpolation polynomial: $f[x_3] + f[x_2, x_3](x - x_3) + f[x_1, x_2, x_3](x - x_2)(x - x_3) = 0$, where f[•] indicates a Newton divided difference formula. This equation is a quadratic equation whose roots may be calculated; when applying this formula iteratively, the root closest to the iterate x_i is the root of the smallest absolute value. In order to have a numerically stable procedure, the reciprocal of the standard quadratic formula is used, i.e.,

$$x_{i+1} = x_i - \frac{c_i}{b_i \pm \sqrt{b_i^2 - a_i c_i}}$$

with:

$$a_i = f[x_{i-2}, x_{i-1}, x_i]$$
$$b_i = 0.5\,(f[x_{i-1}, x_i] + f[x_{i-2}, x_{i-1}, x_i]\,(x_i - x_{i-1}))$$
$$c_i = f[x_i]$$

where the sign of the square root is chosen to maximize the magnitude of the denominator. If $a_i = 0$, a linear interpolation step is taken; if $a_i = b_i = 0$, the iteration is restarted with different values.

Two different stopping criteria are used, using both an x tolerance and a function tolerance parameter; if both are met, convergence has been achieved.

Algorithm Outline:

Enter: x tolerance xtol, function tolerance ftol, max number of iterations itmax;

Enter or generate three estimates x_0, x_1, x_2 of a zero ξ of f(x);

{ recommendation of the cited reference computes $\xi \pm 0.5$ from an initial ξ }

Determine $f(x_0)$, $f(x_1)$, $f(x_2)$;

Set: $h_2 = x_2 - x_1$; $h_1 = x_1 - x_0$; $f[x_2, x_1] = (f(x_2) - f(x_1))/h_2$;
$f[x_1, x_0] = (f(x_1) - f(x_0))/h_1$;

i = 2;

repeat

$f[x_i, x_{i-1}, x_{i-2}] = (f[x_i, x_{i-1}] - f[x_{i-1}, x_{i-2}])/(h_i + h_{i-1})$;

$\alpha_i = f[x_i, x_{i-1}] + h_i\, f[x_i, x_{i-1}, x_{i-2}]$;

$h_{i+1} = -2\, f(x_i)/(\alpha_i \pm \mathrm{sqrt}(\alpha_i^2 - 4\, f(x_i)\, f[x_i, x_{i-1}, x_{i-2}])$;

{sign choice maximizes denominator}

$x_{i+1} = x_i + h_{i+1}$;

determine $f(x_{i+1})$ and $f[x_{i+1}, x_i] = (f(x_{i+1}) - f(x_i))/h_{i+1}$;

$i = i + 1$;

end until ($|x_i - x_{i-1}| < xtol\ |x_i|$ or $|f(x_i)| < ftol$ or imax exceeded);

REFERENCES

1) Stoer, J. and Bulirsch, R., *Introduction to Numerical Analysis*, Springer-Verlag, 1983, pp. 294-295; 2) Conte, S. D. and de Boor, C., *Elementary Numerical Analysis - An Algorithmic Approach*, 3rd Edition, McGraw - Hill, New York, 1980, pgs. 121 - 122.

■ rootPolynomial

FUNCTION

compRoots = rootPolynomial(compCoefficients, initEstimate);

PURPOSE

Determine all of the real and complex roots of the general nth-degree complex polynomial:

$$(a_n + ib_n)\, x^n + (a_{n-1} + ib_{n-1})\, x^{n-1} + \ldots + (a_1 + ib_1)\, x^1 + (a_0 + ib_0)$$

INPUT

compCoefficients (Complex Vector): the vector $[(a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n)]$ of coefficients of the polynomial

initEstimate (Complex Scalar): an initial guess for a root of the polynomial

OUTPUT

compRoots (Complex Vector): a vector whose elements are the complex roots of the polynomial

EXAMPLE

Compute the roots of the polynomial:

$$(3 + 2i)\, x^5 - (4 - 2i)\, x^2 + (3 - i)$$

Example Script:

```
initEstimate = (1.0, 1.0);
compCoefficients = { (3,-1),(0,0),(-4,2),(0,0),(0,0),(3,2) };

compRoots = rootPolynomial(compCoefficients, initEstimate);

//  Results: compRoots:  5 rows
//    0.911149803941382 -0.568982359485567i
//   -0.188626521899724 +1.20684307472613i
//   -0.754462001049842 +0.0316117031013021i
//   -0.730985701477025 -0.857265402851867i
//    0.762924420485209 +0.187792984510005i
```

ALGORITHM AND COMMENTS

This program is a modified complex Newton method, developed by D.M. Russell, that finds the roots of a complex polynomial by combining Newton's method for finding real roots with a method for minimizing the real valued function
$g = |f(z)| = | f(x + iy) |^2$. Since Newton's method only converges locally (i.e., for a sufficiently small neighborhood of a root), a method is used to always guarantee that the initial estimate for the Newton's procedure lies in such a neighborhood.

The essential idea of the method is to locate the zero of the smallest magnitude of the polynomial and then deflate in a stable manner to locate all of the remaining zeros. The Newton formula is used to find search direction for the next iterate, rather than the iterate's value. This direction is usually a descent direction for the function g, resulting in a sequence of decreasing function magnitudes. When we are in a small enough neighborhood of a zero of f(z) to use Newton's method, we apply it. The termination of the process from either the search procedure or the Newton step occurs when either $\varepsilon |z_{n+1}| > |z_{n+1} - z_n| > 0$ or $F(z_{n+1}) = F(z_n) \leq \delta$, where ε is the machine epsilon and δ is a function of ε that overestimates the roundoff error in computing f(z) at the zero with the smallest modulus. In addition to using complex arithmetic, an efficient method for the derivative of a polynomial is used that is based on Horner's method for the implementation of this algorithm. See the reference for further details on the basic algorithm.

In general, the algorithm performs at least as well as the Jenkins-Traub method with respect to accuracy and is considerably faster for almost all complex polynomials.

REFERENCE

Ralston, A. and Rabinowitz, P., *A First Course in Numerical Analysis*, 2nd edition, McGraw-Hill, 1978, section 8.12.

■ sysBrent

FUNCTION

solnVector = sysBrent(eqns, maxIterations, xTolerance, fTolerance, initVector,
jacobianReUse);

PURPOSE

Solve the nonlinear system of equations **F**(**x**) = **0**:

$$f_1(x_1, x_2, ..., x_n) = 0$$
$$f_2(x_1, x_2, ..., x_n) = 0$$
$$...$$
$$f_n(x_1, x_2, ..., x_n) = 0$$

INPUT

eqns (Function): the system of equations to be solved

maxIterations (Integer Scalar): the maximum number of iterations to be used in the main Brent iteration loop

xTolerance (Real Scalar): the value of the x tolerance to be used; determines when the norm of the correction to the updated value of **x** is less than xTolerance

fTolerance (Real Scalar): the value of the absolute function tolerance to be used; determines when the norm of the system of equations, || **-F** ||, at the current iterate value of **x** is less than fTolerance

initVector (Real Vector): the initial approximation vector $\mathbf{x}_0 = (x_1, x_2, ... , x_n)^T$; an initial guess of the solution for **F**(**x**) = **0**

jacobianReUse (Integer Scalar): length of the loop using the computed Brent parameters ss and Q (see the algorithm description)

OUTPUT

solnVector (Real Vector): the numerical solution of the system of equations **F**(**x**) = **0**.

EXAMPLES

Solve the system of equations:

$$x^2 + y^2 = 1$$
$$y = x$$

describing the intersection of a unit circle with the line y = x.

This problem has two solutions: starting from (1,1) one expects to obtain the solution (sqrt(2)/2, sqrt(2)/2); starting from (-1, -1) the expected result is (- sqrt(2)/2, - sqrt(2)/2); of course, in general one does not know the solutions of the system in advance and, many times, whether there are any solutions and if there are solutions, how many there are.

Example Script:

```
project eqns, solnVector, initVector;

local maxIterations, xTolerance, fTolerance, jacobianReUse;
```

```
maxIterations = 100;
xTolerance = 1.0e-6;
fTolerance = 1.0e-6;
jacobianReUse = 1;
initVector = {-1,-1};

Function eqns(index, x)
    select (index) from
        case 1:
            return x[1]^2+x[2]^2 - (1);
        case 2:
            return x[2]-x[1] - (0);
    end select;
    return 0;
end function;
solnVector = sysBrent(eqns, maxIterations, xTolerance,
 fTolerance, initVector, jacobianReUse);

// Results:initVector:  2 columns
//        -1 -1

//        solnVector:  2 rows
//      -0.707107066181216
//      -0.707107066181216
```

ALGORITHM AND COMMENTS

The solution of the nonlinear system $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is returned by this routine using the iterative algorithm due to Richard P. Brent. This algorithm utilizes the secant approximation to the Jacobian for providing the pseudo-Newton step update. The main purposes for using this algorithm are to avoid the computation of the Jacobian $D\mathbf{F}/D\mathbf{x}$ and to minimize the number of function calls necessary for convergence.

Algorithm Outline:

This algorithm consists of the initializations and the loop of major iterations. The length of this loop cannot exceed maxIterations. In the beginning, we start from $\mathbf{y}(1) = \mathbf{x}_0$ and Q[1] = I. The major iteration consists of three parts: the Brent step loop, the loop for reusing computed information, and the new step preparations.

Instead of using Newton's step:

$$\mathbf{F}(\mathbf{x}_0) + \left(\frac{D\mathbf{F}}{D\mathbf{x}}\right)\mathbf{p} = \mathbf{0} \qquad \mathbf{x}_{new} = \mathbf{x}_0 + \mathbf{p}$$

each coordinate is improved subsequently by:

$$F(j, x_0) + \left\langle \frac{DF(j, x)}{Dx}, p_k \right\rangle \qquad x_{new} = x_0 + p_k, \qquad j \le k$$

We try to preserve what was achieved in previous steps; these steps are the Brent loop. Let k be the loop vari-

able, $y_{k+1} = y_k + p_k$, and $A = (a_1, a_2, ..., a_n) \sim DF/Dx$, i.e., $a_j = DF(j, x)/Dx$.
At the kth step of the main Brent loop iteration we find y_{k+1} closest to y_k such that:

$$F(j, y_k) + \langle a_k, y_{k+1} - y_j \rangle, \qquad 1 \le j \le k$$

Rewriting $y_{k+1} - y_j = p_k - (y_k - y_j)$, we obtain:

$$\langle a_j, p_k \rangle = 0, \qquad 1 \le j < k \tag{1}$$

$$\langle a_k, p_k \rangle = -F(k, y_k) \tag{2}$$

Let

$$\begin{bmatrix} a_1^T \\ \dots \\ a_k^T \end{bmatrix} Q_{k+1} = \begin{bmatrix} s_1 & 0 & 0 & 0 \\ \dots & s_2 & 0 & 0 \\ \dots & \dots & s_k & 0 \end{bmatrix}$$

Since

$$\langle a_j, p_k \rangle = a_j^T Q_{k+1} Q_{k+1}^T p_k$$

we have from (1),(2) and from the minimum distance condition that the first k - 1 coordinates of $Q_{k+1}^T p_k$ are zero, that its kth coordinate is $(F(k, y_k)/s_k)\, e_k$, and that,

$$Q_{k+1}^T p_k = -F(k, y(k))\, ss_k e_k$$

where $ss_k = 1/s_k$, e_k is the vector with only one nonzero kth coordinate equal to 1.
This matrix structure is convenient, since at each step we can seek Q_{k+1} in the form:

$$Q_{k+1} = Q_k * U_1 \tag{3}$$

where

$$U_1 = \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix}$$

Here I is the identity matrix, and

$$\dim I = (k-1) \times (k-1) \qquad b^T U = (\dots, s_k, 0, 0) \qquad b = a_k$$

Since only k, k+1, ... , n components of vector b are important , and since a_k is the gradient of the function F_k

and therefore it can be approximated by the corresponding finite difference, we can reduce the dimension. Thus we will use the n - k + 1 dimensional projection b_k of b with elements 1/h*[$F(k,y_k+h\ Q_k\ e_j) - F(k,y_k)$], j = k, ..., n, and will find the Householder's rotation U corresponding to b_k.

We then compute $Q_{k+1} = Q_k*U_1$ (see (3)), put $ss_k = (+/-)\ 1/|b_k|$, and have:

$$p_k = -ss_k\ F(k,y_k)\ Q_{k+1}\ e_k.$$

In summary, the algorithm is:

0. Initialize: $y_1 = x_0$, Q_1 = I,hstep, n = dimension
1. Do main iterations until either the function or the difference between two subsequent iterations are sufficiently small (steps 2-9)
2. For k = 1, ... , n do steps 3-6.
3. Compute vector b_k , $\dim(b_k) = n - k + 1$, with components $1/hstep*[\ F(k,y_k+h\ Q_k\ e_j) - F(k,y_k)\ ]$, j =k, ..., n,
4. Find Householder's rotation U corresponding to the vector b_k
5. Compute $Q_{k+1} = Q_k*U_1$ (see (3)) and $ss_k = (+/-)\ 1/|b_k|$,
6. Let
$$y_{k+1} = y_k - ss_k\ F(k,y_k)\ Q_{k+1}\ e_k$$
7. Do the additional loop in j =1, ... , jacobianReUse (step 8)
8. For k = 1,..., n reuse Q_{n+1} from 5 and compute:
$$y_{k+1} = y_k - ss_k\ F(k,y_k)\ Q_{n+1}\ e_k$$
9. Recompute parameters for the next major iteration:

$y_1 = y_{n+1}$, $Q_1 = Q_{n+1}$, hstep = $-ss_1*F(1, y_1)$ (or set it to be sufficiently small)

General Recommendations for the use of sysBrent are:

1. Usually fTolerance is a more appropriate restriction from the user point of view than xTolerance is, but care should be taken in its choice because an appropriate value depends on the scaling of the functions;

2. If the solution does not exist, xTolerance may provide a good measure for when to terminate the process;

3. jacobianReUse > 0 gives an opportunity (especially for large dimensions) to reduce the number of function calls but may cause divergence;

4. Since loss of precision is possible in the evaluation of function F, it is necessary to impose a restriction on the size of hstep in the calculation of the gradient of F; this is done internally in the program;

5. If divergence occurs, j-iterations will be suppressed (jacobianReUse = 0) ; and

6. Large dimensions will impose heavy requirements on the memory; on the order of n^3, where n is the dimension of the system

REFERENCES

1) Brent, R.P., *Some Efficient Algorithms for Solving Systems of Nonlinear Equations*, SIAM J. Numerical

Analysis, No 2, Apr. 1973, p.327-345; 2) More, J.J., Cosnard, M.Y., *Numerical Solution of Nonlinear Equations*, ACM Trans.of Math. Soft., V.5, No.1, March 1979, p. 64-85.

■ sysNewton

FUNCTION

solnVector = sysNewton(eqns, maxIterations, xTolerance, fTolerance, initVector);

PURPOSE

Solve the nonlinear system of equations **F**(**x**) = **0**:

$$\begin{aligned} f_1(x_1, x_2, ..., x_n) &= 0 \\ f_2(x_1, x_2, ..., x_n) &= 0 \\ &... \\ f_n(x_1, x_2, ..., x_n) &= 0 \end{aligned}$$

INPUT

eqns (Function): the system of equations to be solved

maxIterations (Integer Scalar): the maximum number of iterations to be used in the Newton iteration procedure

xTolerance (Real Scalar): the value of the x tolerance to be used; determines when the norm of the correction to the updated value of **x** is less than xTolerance

fTolerance (Real Scalar): the value of the function tolerance to be used; determines when the norm of the system of equations, $\| -\mathbf{F} \|$, at the current iterate value of **x** is less than fTolerance

initVector (Real Vector): the initial approximation vector $\mathbf{x}_0 = (x_1, x_2, ... , x_n)^T$; an initial guess of the solution for **F**(**x**) = **0**.

OUTPUT

solnVector (Real Vector): the numerical solution of the system of equations **F**(**x**) = **0**.

EXAMPLES

Solve the system of equations:

$$\begin{aligned} & x_1 - e^{(x_7/10)} = 0 \\ & x_1 x_3 + x_2 x_4 + x_5 x_6 x_7 - 4 = 0 \\ & x_1 - (x_3 x_4)^2 \cos(x_7) + x_3 x_5 - 3 x_6 = 0 \\ & (x_1 + x_2)^4 + (x_3 + x_4)^3 + (x_5 + x_6)^2 + \sin[\pi \sin(x_7)] = 0 \end{aligned}$$

$x_1^2 + x_7 - x_2 - x_3 = 0$

$x_2^3 - x_3 + 3(x_4 - x_6) = 0$

$x_3 - 2(x_4 + x_5) - 9.8\,x_7 = 0$

The expected solution is: $x^* = (1, -1, 2, -2, 3, -3, 0)^t$.

Typical Input: xTolerance = fTolerance = 10^{-6};

initialVector = $(2, -2, 2, -2, 2, -2, 0.4)^t$;

Example Script:

```
x_exact = {1; -1; 2; -2; 3; -3; 0};
maxIterations = 1000;
xTolerance = 1.0e-6;
fTolerance = 1.0e-6;
initVector = {2; -2; 2; -2; 2; -2; 0.4};

Function eqns(index, x)
    select (index) from
        case 1:
            return x[1]-exp(0.1*x[7]) - (0);
        case 2:
            return x[1]*x[3]+x[2]*x[4]+x[5]*x[6]*x[7]-4 - (0);
        case 3:
            return x[1]-(x[3]*x[4])^2*cos(x[7])+x[3]*x[5]
                -3*x[6] - (0);
        case 4:
            return (x[1]+x[2])^4+(x[3]+x[4])^3+(x[5]+x[6])^2
                sin(<pi>*sin(x[7])) - (0);
        case 5:
            return x[1]^2+x[7]-x[2]-x[3] - (0);
        case 6:
            return x[2]^3-x[3]+3*(x[4]-x[6]) - (0);
        case 7:
            return x[3]-2*(x[4]+x[5])-9.8*x[7] - (0);
    end select;
    return 0;
end function;

solnVector = sysNewton(eqns, maxIterations, xTolerance, fTolerance,
initVector);
```

ALGORITHM AND COMMENTS

This standard algorithm approximates solutions to nonlinear systems of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ by a functional iteration procedure that starts with the initial solution estimate $\mathbf{x}(0)$ and generates the sequence of problems $\mathbf{x}(k) = \mathbf{G}(\mathbf{x}(k-1))$, where $\mathbf{G}(\mathbf{x}) = \mathbf{x} - J(\mathbf{x})^{-1}\,\mathbf{F}(\mathbf{x})$. $J(\mathbf{x})$, the Jacobian matrix of the system, is computed using a second-order central difference formula. The iteration procedure is terminated when either the sum of the magnitudes of $F_i(\mathbf{x})$ is less than a specified function error tolerance or when the norm of the correction to $\mathbf{x}(k-1)$ is less than a given x error tolerance. It is a locally convergent algorithm that will give a quadratic con-

vergence rate for sufficiently accurate initial estimates.

Define: $\| \mathbf{x} \|$ = norm of the vector **x**
xtol = x error tolerance
ftol = function error tolerance
maxiter = maximum number of iterations

Algorithm Outline:
Initialization: entered values: Initial approximation $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$, xtol, ftol, maxiter
compute: n = number of equations = number of unknowns
Set k = 1
While k ≤ maxiter
    Calculate **F**(**x**(k)) and J(**x**(k))
    Compute ferror = $\| -\mathbf{F} \|$
    If ferror ≤ ftol
        Return
    else
        Solve the linear system of equations: J(**x**)**y** = - **F**(**x**)
        Update the value of **x**: **x** = **x** + **y**
        Compute xerror = $\| \mathbf{y} \|$
        If xerror ≤ xtol
            Return
        else
            k = k + 1
        end
    end
Return (Maximum number of iterations exceeded).

REFERENCE

Burden, R.L. and Faires, J.D., *Numerical Analysis*, 4th ed., PWS - KENT Publ., 1989

■ sysQuasiNewton

FUNCTION

solnVector = sysQuasiNewton(eqns, maxIterations, xTolerance, fTolerance, initVector);

PURPOSE

Solve the nonlinear system of equations **F**(**x**) = **0**:

$$f_1(x_1, x_2, ..., x_n) = 0$$
$$f_2(x_1, x_2, ..., x_n) = 0$$
$$...$$
$$f_n(x_1, x_2, ..., x_n) = 0$$

INPUT

eqns (Function): the system of equations to be solved

maxIterations (Integer Scalar): the maximum number of iterations to be used in the Broyden iterative procedure

xTolerance (Real Scalar): the value of the x tolerance to be used; determines when the norm of the correction to the updated value of **x** is less than xTolerance

fTolerance (Real Scalar): the value of the function tolerance to be used; determines when the norm of the system function, || **F** ||, at the current iterate value of **x** is less than fTolerance

initVector (Real Vector): the initial approximation vector $\mathbf{x}_0 = (x_1, x_2, ..., x_n)^T$; an initial guess of the solution for **F**(**x**) = **0**.

OUTPUT

solnVector (Real Vector): the numerical solution of the system of equations **F**(**x**) = **0**.

EXAMPLES

Solve the system of equations:

$$x^2 + y^2 = 1 \qquad y = x$$

describing the intersection of a unit circle with the line y = x.

This problem has two solutions: starting from (1,1) one expects to obtain the solution (sqrt(2)/2, sqrt(2)/2); starting from (-1, -1) the expected result is (- sqrt(2)/2, - sqrt(2)/2); of course, in general one does not know the solutions of the system in advance and, many times, whether there are any solutions and if there are solutions, how many there are.

Example Script:

```
project eqns, solnVector, initVector;

local maxIterations, xTolerance, fTolerance;

maxIterations = 100;
```

```
xTolerance = 1.0e-6;
fTolerance = 1.0e-6;
initVector = {1,1};

Function eqns(x)
y[1] = x[1]^2+x[2]^2-(1);
y[2] = x[2] - x[1];
    return y;
end function;

solnVector = sysQuasiNewton(eqns, maxIterations, xTolerance, fToler-
ance, initVector);

//  Results:
//initVector:  2 columns
//     1 1

//solnVector:  2 rows
//      0.707106781186548
//      0.707106781186548
```

ALGORITHM AND COMMENTS

This is a least-change secant update program that combines Broyden's method with the Sherman-Morrison matrix inversion formula. Although it is still a locally convergent method, it has two distinct advantages over Newton's method. One advantage is that this algorithm implements Broyden's method, which requires only n functional evaluations per iteration, compared with the n^2 + n evaluations required by Newton's method. Another advantage is that use of the Sherman -Morrison matrix inversion formula reduces the number of arithmetic calculations from $O(n^3)$ for Newton's method to $O(n^2)$. The disadvantages are that the algorithm now has only a superlinear convergent rate (see the references) and that the program is no longer self-correcting (i.e., round-off error is not generally corrected for successive iterations). In spite of these reductions in performance, the computational advantages of this algorithm over Newton's method can be very significant for large problems and problems that Newton's method cannot handle.

Given an initial approximate solution x_0 to F(x) = 0, the next iteration provides the same x_1 as in Newton's method. Denote by J_0 corresponding Jacobian of F(x) (or its finite difference approximation). To compute x(2) however, a generalized secant formula,

$$J_1 (x_1 - x_0) = F(x_1) - F(x_0)$$

is used. It was established by Broyden that J_1 closest to J_0 in Frobenius norm exists that is defined by the famous formula (one-rank Broyden's update)

$$J_1 = J_0 - \frac{(y - J_0 s)\, s^T}{\|s\|^2}$$

where $y = F(x_1) - F(x_0)$, and $s = x_1 - x_0$. One then computes the next iteration: $x_2 = x_1 - J_1^{-1} F(x_1)$. The method is then repeated to determine x_3 by substituting J_1 for J_0 and the pair (x_2, x_1) for the pair (x_1, x_0) in the above formulas. This is essentially Broyden's method.

Since we need the inverse of the matrix J_1 then J_1 itself the Sherman - Morrison formula

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}$$

is used in the above to compute J_i^{-1} directly from J_{i-1}^{-1} by putting

$$A = J_{i-1} \qquad u = \frac{y - J_{i-1}s}{\|s\|^2} \qquad v = s$$

Not only is there a reduction to a second ordered calculation for the inverse of J_i, but the formation of J_i, necessary for the linear system solver in Newton's method, is also bypassed.

REFERENCES

(1) Broyden, C.G., "A class of methods for solving nonlinear simultaneous equations", *Mathematics of Computation*, **19**, 1965, pgs. 577-593; (2) Dennis, J.E., Jr., and More, J.J., "Quasi-Newton methods, motivation and theory", *SIAM Review*, **19**, 1977, No. 1, pgs. 46-89; (3) Dennis, J.E. and Schnabel, R.B., *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, 1983, Chapter 8.

CHAPTER 27
BUSINESS FUNCTIONS

■ anFV

FUNCTION

a = anFV(f, i, n)

PURPOSE

Compute the annuity from the given future value, compound interest rate, and number of payment periods

INPUT

f (Real Scalar): the future value

i (Real Scalar): the compound interest rate

n (Integer Scalar): the number of payment periods

OUTPUT

a (Real Scalar): the computed annuity

EXAMPLES

```
//  Compute anFV(f,i,n) for f=1000, i=0.1 and n = 10

local f,i,n;
f = 1000;
i = 0.1;
n = 10;
a = anFV(f,i,n);

//  Result:
//     a:           62.7453948825116
```

ALGORITHM AND COMMENTS

The annuity a calculated from the given future value f, the compound interest rate i, and the number of payment periods n is given by

$$a = \frac{i}{(1+i)^n - 1} f$$

REFERENCE

Beyer, W.H., *CRC Standard Mathematical Tables*, 28th Ed., CRC Press, Inc., 1987, p.587

■ anPV

FUNCTION

a = anPV(p, i, n)

PURPOSE

Compute the annuity from the given present value, compound interest rate and number of payment periods

INPUT

p (Real Scalar): the present value

i (Real Scalar): the compound interest rate

n (Integer Scalar): the number of payment periods

OUTPUT

a (Real Scalar): the computed annuity

EXAMPLES

```
//  Compute anPV(p,i,n) for p=1000, i=0.1 and n = 10

local p,i,n;
p = 1000;
i = 0.1;
n = 10;
a = anPV(p,i,n);

//  Result:
//     a:           162.745394882512
```

ALGORITHM AND COMMENTS

The annuity a calculated from the given present value p, the compound interest rate i, and the number of payment periods n is given by

$$a = \frac{i(1+i)^n}{(1+i)^n - 1} p$$

REFERENCE

Beyer, W.H., *CRC Standard Mathematical Tables*, 28th Ed., CRC Press, Inc., 1987, p.587

■ fvAn

FUNCTION

f = fvAn(a, i, n)

PURPOSE

Compute the future value from the given annuity, compound interest rate, and number of payment periods

INPUT

a (Real Scalar): the annuity

i (Real Scalar): the compound interest rate

n (Integer Scalar): the number of payment periods

OUTPUT

f (Real Scalar): the computed future value

EXAMPLES

```
//  Compute fvAn(a,i,n) for a=100, i=0.1 and n = 10

local a,i,n;
a = 100;
i = 0.1;
n = 10;
f = fvAn(a,i,n);

//  Result:
//   f:               1593.7424601
```

ALGORITHM AND COMMENTS

The future value f calculated from the given annuity a, the compound interest rate i, and the number of payment periods n is given by

$$f = \frac{(1+i)^n - 1}{i} a$$

REFERENCE

Beyer, W.H., *CRC Standard Mathematical Tables*, 28th Ed., CRC Press, Inc., 1987, p.587

■ FVPV

FUNCTION

f = FVPV(p, i, n)

PURPOSE

Compute the future value from the given present value, compound interest rate, and number of payment periods

INPUT

p (Real Scalar): the present value

i (Real Scalar): the compound interest rate

n (Integer Scalar): the number of payment periods

OUTPUT

f (Real Scalar): the computed future value

EXAMPLES

```
//  Compute FVPV(p,i,n) for p=1000, i=0.1 and n = 10

local p,i,n;

p = 1000;
i = 0.1;
n = 10;
f = FVPV(p,i,n);

//  Result:
//    f:              2593.7424601
```

ALGORITHM AND COMMENTS

The future value f calculated from the given present value p, the compound interest rate i, and the number of payment periods n is given by

$$f = (1+i)^n p$$

REFERENCE

Beyer, W.H., *CRC Standard Mathematical Tables*, 28th Ed., CRC Press, Inc., 1987, p.587

■ pvAn

FUNCTION

p = pvAn(a, i, n)

PURPOSE

Compute the present value from the given annuity, compound interest rate, and number of payment periods

INPUT

a (Real Scalar): the annuity

i (Real Scalar): the compound interest rate

n (Integer Scalar): the number of payment periods

OUTPUT

p (Real Scalar): the computed present value

EXAMPLES

```
//  Compute pvAn(a,i,n) for a=100, i=0.1 and n = 10
local a,i,n;
a = 100;
i = 0.1;
n = 10;
p = pvAn(a,i,n);
//  Result:
//   p:           614.456710570468
```

ALGORITHM AND COMMENTS

The present value p calculated from the given annuity a, the compound interest rate i, and the number of periods n is given by

$$p = \frac{(1+i)^n - 1}{i(1+i)^n} a$$

REFERENCE

Beyer, W.H., *CRC Standard Mathematical Tables*, 28th Ed., CRC Press, Inc., 1987, p.587

■ PVFV

FUNCTION

p = PVFV(f, i, n)

PURPOSE

Compute the present value from the given future value, compound interest rate, and number of payment periods

INPUT

f (Real Scalar): the future value

i (Real Scalar): the compound interest rate

n (Integer Scalar): the number of payment periods

OUTPUT

p (Real Scalar): the computed present value

EXAMPLES

```
//  Compute PVFV(f,i,n) for f=1000, i=0.1 and n = 10
local f,i,n;
f = 1000;
i = 0.1;
n = 10;
p = PVFV(f,i,n);

//  Result:
//    p:           385.543289429532
```

ALGORITHM AND COMMENTS

The present value p calculated from the given future value f, the compound interest rate i, and the number of payment periods n is given by

$$p = \frac{f}{(1+i)^n}$$

REFERENCE

Beyer, W.H., *CRC Standard Mathematical Tables*, 28th Ed., CRC Press, Inc., 1987, p.587

APPENDIX A

CUSTOMER COMMUNICATION

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the Technical Support Form before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

National Instruments
Corporate Headquarters
6504 Bridge Point Parkway
Austin, TX 78730-5039
(512) 794-0100
Technical support fax: (800) 328-2203
(512) 794-5678

Branch Offices	Phone Number	Fax Number
Australia	03 879 9422	03 879 9179
Austria	0662 435986	0662 437010 19
Belgium	02 757 00 20	02 757 03 11
Denmark	45 76 26 00	45 76 71 11
Finland	90 527 2321	90 502 2930
France	1 48 65 33 00	1 48 65 19 07
Germany	089 7 14 50 93	089 7 14 60 35
Italy	02 48301892	02 48301915
Japan	03 3788 1921	03 3788 1923
Netherlands	01720 45761	01720 42140
Norway	03 846866	03 846860
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 20 51 51	056 20 51 55
U.K.	0635 523545 or 0800 289877 (in U.K. only)	0635 523154

HiQ Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___) _____ Phone (___) _____

Computer brand _____ Model _____ Processor _____

Operating system (include version number) _____

Clock speed _____MHz RAM _____MB Display adapter _____

Mouse ___yes ___no Other adapters installed _____

Hard disk capacity _____MB Brand _____

Boards installed (include revision and configuration) _____

_____

_____

Describe your application _____

_____

HiQ Version _____

_____

The problem is: _____

_____

_____

List any error messages: _____

_____

The following steps reproduce the problem:_____

_____

_____

_____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **HiQ® Reference Manual for Macintosh and Power Macintosh, version 2.1**
Edition Date: **August 1994**
Part Number: **320735B-01**
Please comment on the completeness, clarity, and organization of the manual.

_____

_____

_____

_____

If you find errors in the manual, please record the page numbers and describe the errors.

_____

_____

_____

_____

_____

Thank you for your help.

Name_____

Title _____

Company _____

Address_____

_____

_____

Phone (____) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway, MS 53-02
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
MS 53-02
(512) 794-5678

APPENDIX B
FUNCTION CROSS-REFERENCE LIST

CONTENTS

2D and 3D Graphs

addPlot: display a plot in a graph, 21-1
fitToWindow: fit a graph inside the Graph Editor window, 21-3
focusCamera: change the perspectivity of the view of a 3D graph, 21-4
getAxisFlag: query an attribute of the axis, 21-5
getAxisLimits: query the limits of an axis, 21-7
getAxisMinorTicks: query the number of minor ticks on an axis, 21-9
getAxisScale: query the scaling mode of an axis, 21-10
getAxisTitle: query the title of an axis, 21-12
getGraphDimension: query the dimension of a graph, 21-13
getGraphFlag: query an attribute of a graph, 21-14
getGraphPlotBackfaceMode: query the backface mode of a plot in a graph, 21-15
getGraphPlotContourPlane: query the coordinate plane along which the contours of a plot in a graph are to be made, 21-16
getGraphPlotCoordSystem: query the coordinate system of a plot in a graph, 21-17
getGraphPlotDimension: query the dimension of ah plot in a graph, 21-19
getGraphPlotDisplayFormat: query the display format of a plot in a graph, 21-20
getGraphPlotEdgeMode: query the edge mode of a plot in a graph, 21-22
getGraphPlotFillColor: query the color to fill facets of a plot in a graph, 21-23
getGraphPlotLineColor: query the color to draw edges of a plot in a graph, 21-24
getGraphPlotLineWidth: query the line width to draw edges of a plot in a graph, 21-25
getGraphPlotMarkerColor: query the color to draw vertices of a plot in a graph, 21-27

Animation

Business

Data Fitting

comCubicSpline: complete cubic spline interpolation for data with 1st order derivatives given at the end points, 20-7
evalBSplineInterp: evaluation of the k^{th} order b-spline interpolation for a vector using data with knots given, 20-10
evalComCubicSpline: evaluation of the complete cubic spline for a vector using data with 1st order derivatives given at the end points, 20-11
evalHermInterp: evaluation of the Hermite interpolation polynomial for a vector using data, 20-13
evalLagrInterp: evaluation of the Lagrange interpolation polynomial for a vector using data, 20-14
evalNatCubicSpline: evaluation of the natural cubic spline for a vector using data, 20-15
evalPiecePolyInterp: evaluation of the k^{th} degree piecewise polynomial interpolation for a vector using data, 20-16
evalRatInterp: evaluation of the rational interpolation function for a vector using data, 20-18
formLSFit: formulate the general least-squares problem of fitting data $x_1, \ldots, x_m$ and measurements $y_1, \ldots, y_m$ to a user-defined linear combination of basis functions, 20-19
formPolyFit: formulate the least-squares n^{th}-degree polynomial fit to a set of m points $x_1, \ldots, x_m$ with corresponding measurements $y_1, \ldots, y_n$, 20-21
genFit: solve the general nonlinear least-squares problem of fitting a prescribed function $F(a_1, \ldots, a_m, x_1, \ldots, x_n)$ with model parameters $a_1, \ldots, a_m$ at data values $x_1, \ldots, x_n$ using the Marquardt - Levenberg method, 20-23
hermInterp: Hermite interpolation polynomial for data, 20-27
lagrInterp: Lagrange interpolation polynomial for data, 20-28
lineFit: compute a least-squares line fit to a set of data points $x_1, \ldots, x_n$ with estimated function values $y_1, \ldots, y_n$, 21-29
natCubicSpline: natural cubic spline interpolation for data, 20-31
piecePolyBasis: piecewise polynomial Cardinal basis with knots given, 20-33
piecePolyInterp: n^{th} degree piecewise polynomial interpolation for data, 20-36
ratInterp: rational interpolation function for data, 20-38
SVDFit: compute the solution of a general linear least-squares problem using the singular value decomposition, 20-39

Derivative Formulas

biharmonic: finite difference biharmonic of a function, 8-1
derivative: derivative of a function, 8-2
finiteDiffMat1: finite difference approximation of d/dx over a defined mesh, 8-3
finiteDiffMat2: linear operator (matrix) that is a finite difference approximation of d^2/dx^2, 8-5
laplacian: finite difference Laplacian of a function, 8-8
numDerivative: n^{th} order derivative of the interpolating polynomial of numerical data, 8-9
partDerivative: mixed partial derivative of a function, 8-10
polyDerivative: k^{th} order derivative of a polynomial, 8-11

Eigenvalues and Eigenvectors

comEV: all the eigenvalues and eigenvectors of a complex matrix, 17-1
comEVal: all the eigenvalues of a complex matrix, 17-3
EV: all the eigenvalues and eigenvectors of a real matrix, 17-4
eVal: all the eigenvalues of a real matrix, 17-7
genEV: all the eigenvalues and eigenvectors of a generalized eigenvalue problem, 17-8

genEVal: all the eigenvalues of a generalized eigenvalue problem, 17-10
genIter: the closest eigenvalue to l and corresponding eigenvector for a generalized eigenvalue problem, 17-11
hermEV: all the eigenvalues and eigenvectors of a complex Hermitian matrix, 17-13
hermEVal: all the eigenvalues of a complex Hermitian matrix, 17-15
powerEV: the dominant eigenvalue and eigenvector of a real matrix, 17-16
symEV: all the eigenvalues and eigenvectors of a real symmetric matrix, 17-18
symEVal: all the eigenvalues of a real symmetric matrix, 17-19
symPower: the dominant eigenvalue and eigenvector of the a real symmetric matrix, 17-20

Geometric

angleLine: angle between two lines given their equations, 12-1
angleSlope: angle between two lines given their slopes, 12-2
area: area of the triangle given its vertices, 12-3
conic: type of conic section given by the general quadratic equation, 12-4
dist: distance between two points, 12-6
distPToLine: distance from a point to a line, 12-6
radius: radius of the circle given it's equation, 12-7
slope: slope of a line passing through two points, 12-8

General and Polynomial Roots

rootAnalytic: finds all roots of a complex analytic function, 26-72
rootBisection: a globally convergent algorithm for the determination of roots of a single dimensional function f; this is a modified version of Dekker's algorithm that uses a user-specified initial bracket of the solution and the secant and inverse interpolation methods, if necessary, to find the bracketed root to the desired error tolerance, 26-74
rootMuller: an algorithm for determining all of the real or complex roots of a complex function; a globally convergent iterative method that finds any number of real or complex zeros using Muller's method, 26-76
rootPolynomial: an algorithm for computing all of the roots of a general real or complex polynomial; a globally convergent modified complex Newton method that combines Newton's method for finding real roots with a method that minimizes the function $g = |f(x+iy)|^2$, 26-78

Import/Export

close: closes a file, 24-1
exportComplexMatrix: export a complex matrix to an ASCII text file, 25-1
exportComplexScalar: export a complex scalar to an ASCII text file, 25-3
exportComplexVector: export a complex vector to an ASCII text file, 25-4
exportIntegerMatrix:export an integer matrix to an ASCII text file, 25-7
exportIntegerScalar: export an integer scalar to an ASCII text file, 25-9
exportIntegerVector: export an integer vector to an ASCII text file, 25-10
exportNumeric: export a numeric symbol to a file, 25-11
exportRealMatrix: export a real matrix to an ASCII text file, 25-12
exportRealScalar: export a real scalar to an ASCII text file, 25-13
exportRealVector: export a real vector to an ASCII text file, 25-15
exportSymbol: export a numeric or text symbol to an ASCII text file, 25-16
flush: flushes file buffers to disk, 24-1
getFileName: return the full path name of a file by using the standard file dialog, 25-17

IEEEInt16ToInteger: converts the input bytes to an 16-bit IEEE integer, 24-2
IEEEInt32ToInteger: converts the input bytes to an 32-bit IEEE integer, 24-3
IEEEInt8ToInteger: converts the input bytes to an 8-bit IEEE integer, 24-2
IEEEReal32ToReal: converts the input bytes to a 32-bit IEEE real scalar, 24-3
IEEEReal64ToReal: converts the input bytes to a 64-bit IEEE real scalar, 24-4
importNumeric: import a numeric symbol from an external file, 25-17
importSymbol: import a numeric or text symbol from an external file, 25-18
integerToIEEEInt16: converts the real scalar into a byte string of length 2, 24-5
integerToIEEEInt32: converts the real scalar into a byte string of length 4, 24-5
integerToIEEEInt8: converts the real scalar into a byte string of length 1, 24-4
isEOF: checks to see if the EOF flag has been set for the specified file, 24-6
macReal80ToReal: converts the input bytes to a scalar 80-bit real scalar, 24-6
macReal96ToReal: converts the input bytes to a scalar 96-bit real scalar, 24-6
open: opens a file, 24-7
putFileName: opens the standard file dialog and returns a string with specified file path, 25-18
read: reads the specified number of bytes from a file, 24-10
readLine: reads one line from a specified file, 24-10
realToIEEEReal32: converts the input bytes to a scalar 32-bit IEEE real scalar, 24-11
realToIEEEReal64: converts the input bytes to a scalar 64-bit IEEE real scalar, 24-11
realToMacReal80: converts the input bytes to a scalar 80-bit real scalar, 24-12
realToMacReal96: converts the input bytes to a scalar 96-bit real scalar, 24-13
seek: positions the file pointer for a file, 24-13
stringLength: returns the number of characters in a string, 24-14
write: writes a string to a file, 24-14
writeLine: writes a string and newline character to a file, 24-15

Integral Equations

intEqnFredholm: algorithm for the solution of Fredholm Integral Equations of the 2nd Kind; automatic method based on the methods of Gaussian quadrature and collocation for either difficult problems or for problems requiring only low accuracy results, 26-19
intEqnVolt1: a quadratically convergent Trapezoidal method solving Volterra Integral Equations of the 1st Kind, 26-23
intEqnVolt2: a quadratically convergent Midpoint method solving Volterra Integral Equations of the 2nd Kind, 26-27

Integral Formulas

adSimp: adaptive Simpson's rule for integrating a function, 7-1
chebSing1: singular integral of a function using the nth degree Chebyshev polynomial of the 1st kind, 7-2
chebSing2: singular integral of a function using the nth degree Chebyshev polynomial of the 2nd kind, 7-4
gauss: nth order Gaussian integration rule for a function, 7-6
herIntegral: infinite integral of a function using a nth order Hermite polynomial, 7-7
integParab: integral of a tabulated function using overlapping parabolas, 7-8
integSpline: integral of a tabulated function using a natural cubic spline, 7-10
lagIntegral: infinite integral of a function using an nth order Laguerre polynomial, 7-12
logSing: nth order logarithmic singular integral of a function, 7-13

moment: n^{th} order rule for the k^{th} moment of a function, 7-14
simp: extended Simpson's rule for integrating a function, 7-15
trap: modified trapezoidal rule for integrating a function, 7-17

Linear Algebra

band: conversion of a banded matrix into compact form, 15-1
bandBkSv: back-substitution solution of a banded system with pivot information, 15-2
bandDet: determinant of a band matrix, 15-4
bandLUD: LU decomposition of a band matrix, 15-5
bandSolve: solution of a banded system, 15-7
bkSv: back-substitution solution of factored Ax = b with pivoting information, 13-1
bordered: a bordered matrix, 14-1
cho: Cholesky factorization of a symmetric positive definite matrix, 15-9
colDim: number of columns in a matrix, 13-2
cond1: L1 condition number of a matrix, 13-3
cond2: L2 condition number of a matrix, 13-4
condF: F (Frobenius) condition number of a matrix, 13-5
condI: infinity condition number of a matrix, 13-6
convLTriag: conversion of a lower triangular matrix into compact form or vice versa, 16-1
convSym: conversion of a symmetric triangular matrix to or from compact form, 16-2
convUTriag: conversion of an upper triangular matrix to or from compact form, 16-3
copy: duplication of a matrix, 16-7
defRankLS: rank deficient least-squares solution of a system, 15-10
det: matrix determinant, 13-8
diag: principal diagonal of a matrix, 13-9
diagonal: a diagonal matrix, 14-2
dingDong: a Ding-Dong matrix, 14-3
elemDivide: element by element divide of a matrix or vector, 13-10
elemMultiply: element by element multiply of a matrix or vector, 13-11
fastGQRD: fast Givens QR decomposition of a matrix, 15-13
frank: a Frank matrix, 14-4
fullRankLS: full rank least-squares solution of a system, 15-15
getColumn: a column of a matrix, 13-13
getRow: a row of a matrix, 13-13
GQRD: Givens QR decomposition of a matrix, 15-19
gram: a Gram matrix, 14-4
hankel: a Hankel matrix, 14-5
hAP: Householder post-multiplication, 15-20
hilbert: a Hilbert matrix, 14-6
hPA: Householder pre-multiplication, 15-22
hPV: product of the Householder matrix formed by two vectors, 15-23
hVector: Householder vector, 15-24
ident: identity matrix, 13-14
imagPart: imaginary part of a complex matrix, 13-15
inv: matrix inverse, 13-16
isDiagDom: test for diagonal dominance of a matrix, 16-5
isNonSingSymIndef: test for symmetry, nonsingularity & indefiniteness of a matrix, 16-6
isOrthogonal: test for orthogonality of a matrix, 16-6
isSymmetric: test for symmetry of a matrix, 16-7
isSymNegDef: test for symmetry and negative definiteness of a matrix, 16-8
isSymPosDef: test for symmetry and positive definiteness of a matrix, 16-9
isSymSemiNegDef: test for symmetry and semi-negative definiteness of a matrix, 16-9
isSymSemiPosDef: test for symmetry and semi-positive definiteness of a matrix, 16-10
isTriangular: test for triangularity of a matrix, 16-11
kahan: a Kahan matrix, 14-7
locateMax: value and the location of the maximum element of the vector or matrix, 13-17
locateMin: value and the location of the minimum element of the vector or matrix, 13-18
lowerBand: lower bandwidth of a matrix, 16-11

lTriag: lower triangular part of a matrix, 13-9
lTriDet: determinant of a lower triangular matrix, 15-26
lTriInv: inverse of a lower triangular matrix, 15-27
lTriSolve: solution of a lower triangular system, 15-28
LUD: LU decomposition, 13-20
mGS: modified Gram-Schmidt factorization of a matrix, 15-30
moler: a Moler matrix, 14-8
norm1: L1 norm of a matrix, 13-22
norm2: L2 norm of a matrix, 13-22
normF: F (Frobenius) norm of a matrix, 13-23
normI: infinity norm of a matrix. 13-24
posBkSv: back-substitution solution of a positive definite system., 15-31
posDet: determinant of a symmetric positive definite matrix, 15-32
posInv: inverse of a symmetric positive definite matrix, 15-33
posSolve: solution of a positive definite system, 15-34
prod: product of the elements of a vector or matrix, 13-25
pseudo: pseudoinverse of a matrix, 15-35
QRD: QR decomposition of a matrix, 15-36
randM: a random matrix with elements between 0 and 1, 13-26
rank: numerical rank of a real matrix, 13-27
realPart: real part of a complex matrix, 13-28
rowDim: number of rows in a matrix, 13-29
scalarAdd: add a scalar to all elements of a matrix or vector, 13-30
schurD: Schur decomposition of a matrix, 15-38
solve: solution of a linear system, 13-31
sparsity: sparseness of a matrix, 16-12
sRandM: a seeded random matrix with elements between 0 and 1, 13-32
subdiag: subdiagonal of a matrix, 13-33
submat:a submatrix of a matrix, 13-34
sum: sum of the elements of a vector or matrix, 13-35
SVD: Singular Value Decomposition of a matrix, 15-40
SVDS: singular values of a matrix, 15-41
symBkSv: back-substitution solution of a symmetric system with pivoting information, 15-42
symDet: determinant of a symmetric matrix, 15-44
symInv: inverse of a symmetric matrix, 15-45
symLDU: LDU decomposition of a symmetric matrix, 15-46
symPermu: complete pivoting on a submatrix a symmetric matrix, 15-49
symSolve: solution of a symmetric system, 15-51
toeplitz: a Toeplitz matrix, 14-9
toepSolve: solution of a Toeplitz system, 15-52
tran: matrix transpose, 13-36
upperBand: upper bandwidth of a matrix, 16-13
uTriag: upper triangular part of a matrix, 13-37
uTriDet: determinant of an upper triangular matrix, 15-54
uTriInv: inverse of an upper triangular matrix, 15-55
uTriSolve: solution of an upper triangular system, 15-57
vandermonde: a Vandermonde matrix, 14-10
vanSolve: solution of a Vandermonde system, 15-58
vNorm1: L1 norm of a vector, 13-38
vNorm2: L2 norm of a vector, 13-39
vNormI: infinity norm of a vector, 13-40
wilkinsonMinus: a Wilkinson W- matrix, 14-11
wilkinsonPlus: a Wilkinson W+ matrix, 14-12

Nonlinear Systems

sysBrent: a modification of Brent's method; this is an efficient locally convergent iterative algorithm for computing solutions of nonlinear systems of equations that avoids the computation of the Jacobian and minimizes the number of function calls necessary, 26-79
sysNewton: standard Newton's algorithm; iterates until a user-specified error tolerance is reached either for a norm dependent on the functional values or dependent on the solution iteratations, 26-84
sysQuasiNewton: Quasi-Newton algorithm; least-change secant update program that combines

Broyden's method with the Sherman-Morrison matrix inversion formula, 26-86

Numerical Functions

abs: absolute value, 10-1
arg: principal value, 10-2
ceil: least integer not smaller than x, 10-2
conj: complex conjugate, 10-3
floor: greatest integer not larger than x, 10-4
fractPart: fractional part, 10-5
gcd: greatest common divisor of an integer vector, 10-6
gcd2: greatest common divisor of two integers, 10-6
intPart: integer part, 10-7
lcm: least common multiple of an integer vector, 10-8
lcm2: least common multiple of two integers, 10-8
max: maximum element of a vector or matrix, 10-9
max2: larger of two values, 10-10
min: minimum element of a vector or matrix, 10-10
min2: smaller of two values, 10-11
mod: remainder, 10-12
pow: value of x to the power y, 10-13
round: round-off value, 10-14
sign: sign of x, 10-14
sqrt: square root, 10-15

Numerical Integration

integAdapt: general purpose adaptive quadrature algorithm; a program that combines globally adaptive interval subdivision and extrapolation that also eliminates several kinds of singularity effects, 26-1
integAlgLogSingular: computes integrals that exhibit various kinds of algebraic or logarithmic singular behavior at the endpoints, 26-3
integBkpts: computes integrals over intervals at break points where local difficulties of the integrand, such as singularities and discontinuities occur, 26-6
integFourier: computes Fourier integrals over the interval (a, ∞) for either a sine or cosine weight function by successively applying a finite interval integrator based on the Clenshaw-Curtis method, 26-8
integGauss: high speed smooth function integration algorithm; non-adaptive automatic integration method consisting of a sequence of quadrature rules with increasingly greater degrees of algebraic precision, 26-10
integInfinite: computes three different types of integrals with infinite limits; either from a finite bound to ∞, -∞ to a finite bound, or from -∞ to ∞, 26-12
integMultiple: multi-dimensional integration algorithm that computes an n-dimensional integral using adaptive m^{th} order multiple point Gaussian integration rules, 26-14
integOscillate: computes integrals with an oscillatory integrand of the type f(x)*sin(wx) or f(x)*cos(wx) over the interval (a, b), 26-17

Optimization

optBFGS: solves an unconstrained nonlinear optimization problem using a quasi-Newton method combined with the Broyden-Fletcher-Goldfarb-Shano rank-two update formula, 26-57
optConGradient: a conjugate gradient algorithm for solving unconstrained nonlinear optimization problems for a general multi-variable objective function, 26-59
optLinProg: an algorithm for the solution of arbitrarily large standard linear programming problems using the simplex method, 26-62
optNelderMead: a Downhill Nelder-Mead simplex search algorithm (not requiring derivative evaluations) for computing the minimum of a function of several parameters that is also useful as a nonlinear equation system solver that does not require derivative information, 26-65
optNonLinCon: a recently developed algorithm for performing constrained nonlinear optimization with nonlinear equality or inequality constraints that uses

the Augmented Lagrangian Method (ALM) to sequentially recompute the Lagrangian multipliers from each successive unconstrained minimization problem for the augmented Lagrangian, 26-68

Ordinary Differential Equations - Initial Value

odeIvpRKF: general purpose non-stiff and moderately stiff ordinary differential equation solution algorithm; implements Fehlberg's popular $4^{th}/5^{th}$ - order method of applying the Runge-Kutta formulas, 26-39

odeIvpSmooth: high speed extrapolation algorithm for smooth systems of ordinary differential equations; an adaptation of the Bulirsch-Stoer-Gragg algorithm, combined with the explicit midpoint rule, that has both stepsize and order control, 26-42

odeIvpSmoothNEq: same as odeIvpSmooth but can return a solution with a varying step size, 26-48

odeIvpStiff: general purpose algorithm for solving stiff systems of ordinary differential equations; a modified version of the cyclic composite method by Tendler, Bickart and Picel that implements stepsize and order control for multi-step schemes up to order seven, 26-51

odeIvpStiffNEq: same as odeIvpStiff but can return a solution with a varying step size, 26-55

Ordinary Differential Equations - Boundary Value

odeBvpGenLinear: algorithm for solving general linear boundary value problems; a standard multiple shooting algorithm utilizing the ODE initial value problem solvers odeIvpRKF or odeIvpSmooth is used to integrate nonstiff problems with general linear boundary conditions, 26-31

odeBvpGenNonlinear: general purpose algorithm for solving general nonlinear boundary value problems; an algorithm that combines Gaussian collocation with quasilinearization to solve mixed-ordered ODE systems that are subject to separated, multipoint boundary conditions, 26-35

Orthogonal Polynomials

aLag: associated Laguerre polynomial $L_{nm}(x)$, 4-1

aLeg: associated Legendre polynomial of the 1^{st} kind $P_{nm}(x)$, 4-2

cheb1: Chebyshev polynomial of the 1^{st} kind $T_n1(x)$, 4-3

cheb2: Chebyshev polynomial of the 2^{nd} kind $T_n2(x)$, 4-4

harm: spherical harmonic function $Y_{lm}(t,p)$, 4-5

her: Hermite polynomial $H_n(x)$, 4-6

jac: Jacobi polynomial $P_{nab}(x)$, 4-7

lag: Laguerre polynomial $L_n(x)$, 4-8

leg: Legendre polynomial of the 1^{st} kind $P_n(x)$, 4-9

qLeg: Legendre polynomial of the 2^{nd} kind $Q_n(x)$, 4-10

Polynomial Functions

degreePoly: degree of the polynomial specified by a vector, 11-1

derivativePoly: k^{th} order derivative at x for the polynomial specified by a vector, 11-2

multPoly: product of two polynomials, specified by vectors, 11-2

poly: polynomial, specified by a vector, evaluated at x, 11-3

ratPoly: ratio of two polynomials, specified by vectors at x, 11-4

Probability and Statistics

avg: average of the values in a vector, 19-1

avgDev: average deviation of the values in a vector, 19-2

betaDF: beta distribution density function, 19-3

betaDist: beta distribution function, 19-4

bin: binomial coefficient, 19-5

Sequence and Series

Signal Processing

response: system response, 18-39
sinFT: sine transform of a real vector, 18-41
twoRealFFT: fast Fourier transform of two real vectors, 18-43
window: product of a real vector with a window type, 18-44
winSum: squares of weights of a window, 18-46
zTrans: z-transform function, 18-47

Special Functions

ai: positive order Airy function, 5-1
bei: n^{th} order imaginary Bessel Kelvin function, 5-2
ber: n^{th} order real Bessel Kelvin function, 5-4
beta: beta function, 5-5
bi: negative order Airy function, 5-6
cn: Jacobi elliptic function, 6-1
comel1: complete elliptic integral of the 1^{st} kind K(n), 6-2
comel2: complete elliptic integral of the 2^{nd} kind E(n), 6-4
comIGamma: complement of the incomplete gamma function, 5-8
cosI: cosine integral, 6-5
daw: Dawson integral, 6-6
dilog: dilogarithm (Spence's integral), 6-7
dn: Jacobi elliptic function, 6-8
el1: elliptic integral of the 1^{st} kind, 6-10
el2: elliptic integral of the 2^{nd} kind, 6-11
erf: error function, 5-9
erfc: complement of the error function, 5-10
expI: exponential integral, 6-13
fCosI: Fresnel cosine integral, 6-14
fHyper: Gauss hypergeometric function, 5-11
fSeries: power series expansion of the Gauss hypergeometric function, 5-12
fSinI: Fresnel sine integral, 6-15
gamma: gamma function, 5-13
hCosI: hyperbolic cosine integral, 6-17
hSinI: hyperbolic sine integral, 6-8
iBeta: (normalized) incomplete beta function, 5-14
iGamma: incomplete gamma function, 5-16
in: Modified Bessel function of the 1^{st} kind, 5-17
jn: Bessel function of the 1^{st} kind, 5-18
js: spherical Bessel function of order n, 5-19
kei: n^{th} order imaginary Bessel Kelvin function, 5-20
ker: n^{th} order real Kelvin function, 5-21
kn: Modified Bessel function of the 2^{nd} kind, 5-22
lnGamma: natural logarithm of the gamma function, 5-23
mHyper: Kummer confluent hypergeometric function, 5-24
mSeries: power series expansion of the Kummer confluent hypergeometric function, 5-26
psi: psi function, 5-27
sinI: sine integral, 6-19
sn: Jacobi elliptic function, 6-20
struve: n^{th} order Struve function, 5-28
uHyper: Tricomi associated confluent hypergeometric function, 5-29
uSeries: power series expansion of the Tricomi associated confluent hypergeometric function, 5-30
weber: n^{th} order parabolic cylinder function, 5-31
yn: Bessel function of the 2^{nd} kind, 5-33
ys: spherical Bessel function of order n, 5-34
zeta: Riemann zeta function, 5-35

Transcendental

arcCosh: inverse hyperbolic cosine, 3-1
arcCoth: inverse hyperbolic cotangent, 3-2
arcCsch: inverse hyperbolic cosecant, 3-3
arcSech: inverse hyperbolic secant, 3-4
arcSinh: inverse hyperbolic sine, 3-5
arcTanh: inverse hyperbolic tangent, 3-6
cosh: hyperbolic cosine, 3-7
coth: hyperbolic cotangent, 3-8
csch: hyperbolic cosecant, 3-9
exp: exponential function, 3-10
gd: gudermannian, 3-12
gdInv: inverse gudermannian, 3-12
ln: natural logarithm, 3-13
log: logarithm base 10, 3-14
logb: logarithm base b, 3-16

Trigonometric

Utility

FUNCTION LIST

D

E

F

G

H

I

J

K

P

Q

R

S

T