GAUSS^{TM} 14

Language Reference

Vol. 1

GAUSS^{TM} 14

Language Reference

Vol. 2

GAUSS™ 14

Language Reference

Vol. 3

Part Number: 008799
Version 14
Document Revision: 1836
12/11/2013

Contents

36 Command Reference Introduction

The GAUSS LANGUAGE REFERENCE describes each of the commands, procedures and functions available in the **GAUSS**™ programming language. These functions can be divided into four categories:

- Mathematical, statistical and scientific functions.
- Data handling routines, including data matrix manipulation and description routines, and file I/O.
- Programming statements, including branching, looping, display features, error checking, and shell commands.
- Graphics functions.

The first category contains those functions to be expected in a high level mathematical language: trigonometric functions and other transcendental functions, distribution functions, random number generators, numerical differentiation and integration routines, Fourier transforms, Bessel functions and polynomial evaluation routines. And, as a matrix programming language, **GAUSS** includes a variety of routines that perform standard matrix operations. Among these are routines to calculate determinants, matrix inverses, decompositions, eigenvalues and eigenvectors, and condition numbers.

Data handling routines include functions which return dimensions of matrices, and information about elements of data matrices, including functions to locate values lying in specific ranges or with certain values. Also under data handling routines fall all those functions that create, save, open and read from and write to **GAUSS** data sets and **GAUSS** Data Archives. A variety of sorting routines which will operate on both numeric and character data are also available.

Programming statements are all of the commands that make it possible to write complex programs in **GAUSS**. These include conditional and unconditional branching, looping, file I/O, error handling, and system-related commands to execute OS shells and access directory and environment information.

The graphics functions of **GAUSS Publication Quality Graphics** (PQG) are a set of routines built on the graphics functions in GraphiC by Scientific Endeavors Corporation. **GAUSS** PQG consists of a set of main graphing procedures and several additional procedures and global variables for customizing the output.

36.1 Documentation Conventions

The following table describes how text formatting is used to identify **GAUSS** programming elements:

Text Style	Use	Example
regular text	narrative	"... text formatting is used ..."
bold text	emphasis	**"...not supported under UNIX."**
italics	variables	"... If *vnames* is a string or has fewer elements than *x* has columns, it will be ..."

`monospace`	code example	`if scalerr(cm);`
		`  cm = inv(x);`
		`endif;`

`monospace`	filename, path, etc.	"...is located in the `examples` subdirectory..."
`monospace`	reference to a **GAUSS**	"...as explained under

`bold`	command or other programming element within a narrative paragraph	`plotScatter`..."
Bold Text	reference to section of the manual	"...see **Operator Precedence**, Section 1.1 ..."

36.2 Command Components

The following list describes each of the components used in the **Command Reference**, Chapter 38 .

Purpose

Describes what the command or function does.

Library

Lists the library that needs to be activated to access the function.

Include

Lists files that need to be included to use the function.

Format

Illustrates the syntax of the command or function.

Input

Describes the input parameters of the function.

Global Input

Describes the global variables that are referenced by the function.

Output

Describes the return values of the function.

Global Output

Describes the global variables that are updated by the function.

Portability

Describes differences under various operating systems.

Remarks

Explanatory material pertinent to the command.

Example

Sample code using the command or function.

Source

The source file in which the function is defined, if applicable.

Globals

Global variables that are accessed by the command.

See Also

Other related commands.

Technical Notes

Technical discussion and reference source citations.

References

Reference material citations.

36.3 Using This Manual

Users who are new to **GAUSS** should make sure they have familiarized themselves with LANGUAGE FUNDAMENTALS, Chapter 9, before proceeding here. That chapter contains the basics of **GAUSS** programming.

In all, there are over 800 routines described in this GAUSS LANGUAGE REFERENCE. We suggest that new **GAUSS** users skim through **Commands by Category**, Chapter 37 , and then browse through **Command Reference**, Chapter 38 , the main part of this manual. Here, users can familiarize themselves with the kinds of tasks that **GAUSS** can handle easily.

Chapter 37 gives a categorical listing of all functions in this GAUSS LANGUAGE REFERENCE and a short discussion of the functions in each category. Complete syntax, description of input and output arguments, and general remarks regarding each function are given in Chapter 38 .

If a function is an "extrinsic" (that is, part of the **Run-Time Library**), its source code can be found on the `src` subdirectory. The name of the file containing the source code is given in Chapter 38 under the discussion of that function.

36.4 Global Control Variables

Several **GAUSS** functions use global variables to control various aspects of their performance. The files `gauss.ext`, `gauss.dec` and `gauss.lcg` contain the `external` statements, `declare` statements, and library references to these globals. All globals used by the **GAUSS Run-Time Library** begin with an underscore '_'.

Default values for these common globals can be found in the file `gauss.dec`, located on the src subdirectory. The default values can be changed by editing this file.

36.4.1 Changing the Default Values

To permanently change the default setting of a common global, two files need to be edited: `gauss.dec` and `gauss.src`.

To change the value of the common global *`__output`* from 1 to 0, for example, edit the file `gauss.dec` and change the statement

```
declare matrix __output = 1;
```

so it reads:

```
declare matrix __output = 0;
```

Also, edit the procedure **`gausset`**, located in the file `gauss.src`, and modify the statement

```
__output = 1;
```

similarly.

36.4.2 The Procedure `gausset`

The global variables affect your program, even if you have not set them directly in a particular command file. If you have changed them in a previous run, they will retain their changed values until you exit **GAUSS** or execute the `new` command.

The procedure **`gausset`** will reset the **Run-Time Library** globals to their default values.

```
gausset;
```

If your program changes the values of these globals, you can use **`gausset`** to reset them whenever necessary. **`gausset`** resets the globals as a whole; you can write your own routine to reset specific ones.

37 Commands by Category

37.1 Mathematical Functions

Scientific Functions

`abs`	Returns absolute value of argument.
`arccos`	Computes inverse cosine.
`arcsin`	Computes inverse sine.
`atan`	Computes inverse tangent.
`atan2`	Computes angle given a point *x,y*.
`besselj`	Computes Bessel function, first kind.
`bessely`	Computes Bessel function, second kind.
`beta`	Computes the complete Beta function, also called the Euler integral.
`boxcox`	Computes the Box-Cox function.
`cos`	Computes cosine.
`cosh`	Computes hyperbolic cosine.
`curve`	Computes a one-dimensional smoothing curve.

`digamma`	Computes the digamma function.
`exp`	Computes the exponential function of *x*.
`fmod`	Computes the floating-point remainder of ***`x/y`***.
`gamma`	Computes gamma function value.
`gammacplx`	Computes gamma function for complex inputs.
`gammaii`	Compute the inverse incomplete gamma function.
`ln`	Computes the natural log of each element.
`lnfact`	Computes natural log of factorial function.
`lngammacplx`	Computes the natural log of the gamma function for complex inputs.
`log`	Computes the log of each element.
`mbesseli`	Computes modified and exponentially scaled modified Bessels of the first kind of the nth order.
`nextn, nextnevn`	Returns allowable matrix dimensions for computing FFT's.
`optn, optnevn`	Returns optimal matrix dimensions for computing FFT's.
`pi`	Returns π.
`polar`	Graphs data using polar coordinates.
`polygamma`	Computes the polygamma function of order `n`.

`psi`	Computes the psi (or digamma) function.
`sin`	Computes sine.
`sinh`	Computes the hyperbolic sine.
`spline`	Computes a two-dimensional interpolatory spline.
`sqrt`	Computes the square root of each element.
`tan`	Computes tangent.
`tanh`	Computes hyperbolic tangent.
`tocart`	Converts from polar to Cartesian coordinates.
`topolar`	Converts from Cartesian to polar coordinates.
`trigamma`	Computes trigamma function.
`zeta`	Computes the Rieman zeta function.

All trigonometric functions take or return values in radian units.

Differentiation and Integration

`gradMT`	Computes numerical gradient.
`gradMTm`	Computes numerical gradient with mask.
`gradMTT`	Computes numerical gradient using available threads.
`gradMTTm`	Computes numerical gradient with mask using available threads.
`gradp, gradcplx`	Computes first derivative of a function;

	gradcplx allows for complex arguments.
hessMT	Computes numerical Hessian.
hessMTg	Computes numerical Hessian using gradient procedure.
hessMTgw	Computes numerical Hessian using gradient procedure with weights.
hessMTm	Computes numerical Hessian with mask.
hessMTmw	Computes numerical Hessian with mask and weights.
hessMTT	Computes numerical Hessian using available threads.
hessMTTg	Computes numerical Hessian using gradient procedure with available threads.
hessMTTgw	Computes numerical Hessian using gradient procedure with weights and using available threads.
hessMTTm	Computes numerical Hessian with mask and available threads.
hessMTw	Computes numerical Hessian with weights.
hessp, hesscplx	Computes second derivative of a function; **hesscplx** allows for complex arguments.
intgrat2	Integrates a 2-dimensional function over an user-defined region.
intgrat3	Integrates a 3-dimensional function over an

	user-defined region.
inthp1	Integrates an user-defined function over an infinite interval.
inthp2	Integrates an user-defined function over the [a, +∞) interval.
inthp3	Integrates an user-defined function over the [a, +∞) interval that is oscillatory.
inthp4	Integrates an user-defined function over the [a, b] interval.
inthpControlCreate	Creates default **inthpControl** structure.
intquad1	Integrates a 1-dimensional function.
intquad2	Integrates a 2-dimensional function over an user-defined rectangular region.
intquad3	Integrates a 3-dimensional function over an user-defined rectangular region.
intsimp	Integrates by Simpson's method.

gradp and **hessp** use a finite difference approximation to compute the first and second derivatives. Use **gradp** to calculate a Jacobian.

intquad1, **intquad2**, and **intquad3** use Gaussian quadrature to calculate the integral of the user-defined function over a rectangular region.

To calculate an integral over a region defined by functions of *x* and *y*, use **intgrat2** and **intgrat3**.

To get a greater degree of accuracy than that provided by **intquad1**, use **intsimp** for 1-dimensional integration.

Linear Algebra

`balance`	Balances a matrix.
`band`	Extracts bands from a symmetric banded matrix.
`bandchol`	Computes the Cholesky decomposition of a positive definite banded matrix.
`bandcholsol`	Solves the system of equations $Ax = b$ for x, given the lower triangle of the Cholesky decomposition of a positive definite banded matrix A.
`bandltsol`	Solves the system of equations $Ax = b$ for x, where A is a lower triangular banded matrix
`bandrv`	Creates a symmetric banded matrix, given its compact form.
`bandsolpd`	Solves the system of equations $Ax = b$ for x, where A is a positive definite banded matrix.
`chol`	Computes Cholesky decomposition, $X=Y`Y$.
`choldn`	Performs Cholesky downdate on an upper triangular matrix.
`cholsol`	Solves a system of equations given the Cholesky factorization of a matrix.
`cholup`	Performs Cholesky update on an upper triangular matrix.
`cond`	Computes condition number of a matrix.
`crout`	Computes Crout decomposition, $X = LU$ (real

	matrices only).
`croutp`	Computes Crout decomposition with row pivoting (real matrices only).
`det`	Computes determinant of square matrix.
`detl`	Computes determinant of decomposed matrix.
`hess`	Computes upper Hessenberg form of a matrix (real matrices only).
`inv`	Inverts a matrix.
`invpd`	Inverts a positive definite matrix.
`invswp`	Computes a generalized sweep inverse.
`lapeighb`	Computes eigenvalues only of a real symmetric or complex Hermitian matrix selected by bounds.
`lapeighi`	Computes eigenvalues only of a real symmetric or complex Hermitian matrix selected by index.
`lapeighvb`	Computes eigenvalues and eigenvectors of a real symmetric or complex Hermitian matrix selected by bounds.
`lapeighvi`	Computes selected eigenvalues and eigenvectors of a real symmetric or complex Hermitian matrix.
`lapgeig`	Computes generalized eigenvalues for a pair of real or complex general matrices.
`lapgeigh`	Computes generalized eigenvalues for a pair of

	Hermitian matrices.
`lapgeighv`	Computes generalized eigenvalues and eigenvectors for a pair of real symmetric or Hermitian matrices.
`lapgeigv`	Computes generalized eigenvalues, left eigenvectors, and right eigenvectors for a pair of real or complex general matrices.
`lapgschur`	Computes the generalized Schur form of a pair of real or complex general matrices.
`lapgsvdcst`	Computes the generalized singular value decomposition of a pair of real or complex general matrices.
`lapgsvds`	Computes the generalized singular value decomposition of a pair of real or complex general matrices.
`lapgsvdst`	Computes the generalized singular value decomposition of a pair of real or complex general matrices.
`lapsvdcusv`	Computes the singular value decomposition a real or complex rectangular matrix, returns compact *u* and *v*.
`lapsvds`	Computes the singular values of a real or complex rectangular matrix.
`lapsvdusv`	Computes the singular value decomposition a real or complex rectangular matrix.
`lu`	Computes LU decomposition with row pivoting

	(real and complex matrices).
`null`	Computes orthonormal basis for right null space.
`null1`	Computes orthonormal basis for right null space.
`orth`	Computes orthonormal basis for column space x.
`pinv`	Generalized pseudo-inverse: Moore-Penrose.
`pinvmt`	Generalized pseudo-inverse: Moore-Penrose.
`qqr`	QR decomposition: returns Q_1 and R.
`qqre`	QR decomposition: returns Q_1, R and a permutation vector, E.
`qqrep`	QR decomposition with pivot control: returns Q_1, R, and E.
`qr`	QR decomposition: returns R.
`qre`	QR decomposition: returns R and E.
`qrep`	QR decomposition with pivot control: returns R and E.
`qrsol`	Solves a system of equations $R'x = b$ given an upper triangular matrix, typically the R matrix from a QR decomposition.
`qrtsol`	Solves a system of equations $R'x = b$ given an upper triangular matrix, typically the R matrix from a QR decomposition.
`qtyr`	QR decomposition: returns $Q'Y$ and R.

`qtyre`	QR decomposition: returns $Q'Y$, R and E.
`qtyrep`	QR decomposition with pivot control: returns $Q'Y$, R and E.
`qyr`	QR decomposition: returns QY and R.
`qyre`	QR decomposition: returns QY, R and E.
`qyrep`	QR decomposition with pivot control: returns QY, R and E.
`rank`	Computes rank of a matrix.
`rref`	Computes reduced row echelon form of a matrix.
`schtoc`	Reduces any 2x2 blocks on the diagonal of the real Schur matrix returned from **`schur`**. The transformation matrix is also updated.
`schur`	Computes Schur decomposition of a matrix (real matrices only).
`solpd`	Solves a system of positive definite linear equations.
`svd`	Computes the singular values of a matrix.
`svd1`	Computes singular value decomposition, $X = USV'$.
`svd2`	Computes **`svd1`** with compact U.
`svdcusv`	Computes the singular value decomposition of a matrix so that: $x = u * s * v'$ (compact u).
`svds`	Computes the singular values of a matrix.

`svdusv`	Computes the singular value decomposition of a matrix so that: $x = u * s * v'$.

The decomposition routines are **`chol`** for Cholesky decomposition, **`crout`** and **`croutp`** for Crout decomposition, **`qqr-qyrep`** for QR decomposition, and **`svd-svdusv`** for singular value decomposition.

`null`, **`null1`**, and **`orth`** calculate orthonormal bases.

`inv`, **`invpd`**, **`solpd`**, **`cholsol`**, **`qrsol`** and the "**`/`**" operator can all be used to solve linear systems of equations.

`rank` and **`rref`** will find the rank and reduced row echelon form of a matrix.

`det`, **`detl`** and **`cond`** will calculate the determinant and condition number of a matrix.

Eigenvalues

`eig`	Computes eigenvalues of general matrix.
`eigh`	Computes eigenvalues of complex Hermitian or real symmetric matrix.
`eighv`	Computes eigenvalues and eigenvectors of complex Hermitian or real symmetric matrix.
`eigv`	Computes eigenvalues and eigenvectors of general matrix.

There are four eigenvalue-eigenvector routines. Two calculate eigenvalues only, and two calculate eigenvalues and eigenvectors. The three types of matrices handled by these routines are:

General:	**`eig`**, **`eigv`**
Symmetric or Hermitian:	**`eigh`**, **`eighv`**

Polynomial Operations

`polychar`	Computes characteristic polynomial of a square matrix.
`polyeval`	Evaluates polynomial with given coefficients.
`polyint`	Calculates Nth order polynomial interpolation given known point pairs.
`polymake`	Computes polynomial coefficients from roots.
`polymat`	Returns sequence powers of a matrix.
`polymult`	Multiplies two polynomials together.
`polyroot`	Computes roots of polynomial from coefficients.

See also **`recserrc`**, **`recsercp`**, and **`conv`**.

Fourier Transforms

`dfft`	Computes discrete 1-D FFT.
`dffti`	Computes inverse discrete 1-D FFT.
`fft`	Computes 1- or 2-D FFT.
`ffti`	Computes inverse 1- or 2-D FFT.
`fftm`	Computes multi-dimensional FFT.

`fftmi`	Computes inverse multi-dimensional FFT.
`fftn`	Computes 1- or 2-D FFT using prime factor algorithm.
`rfft`	Computes real 1- or 2-D FFT.
`rffti`	Computes inverse real 1- or 2-D FFT.
`rfftip`	Computes inverse real 1- or 2-D FFT from packed format FFT.
`rfftn`	Computes real 1- or 2-D FFT using prime factor algorithm.
`rfftnp`	Computes real 1- or 2-D FFT using prime factor algorithm, returns packed format FFT.
`rfftp`	Computes real 1- or 2-D FFT, returns packed format FFT.

Random Numbers

`rndBeta`	Computes random numbers with beta distribution.
`rndCauchy`	Computes Cauchy distributed random numbers with a choice of underlying random number generator.
`rndcon`	Changes constant of the LC random number generator.
`rndCreateState`	Creates a new random number stream for a specified generator type from a seed value.

`rndExp`	Computes exponentially distributed random numbers with a choice of underlying random number generator.
`rndGamma`	Computes gamma pseudo-random numbers with a choice of underlying random number generator.
`rndGeo`	Computes geometric pseudo-random numbers with a choice of underlying random number generator.
`rndGumbel`	Computes Gumbel distributed random numbers with a choice of underlying random number generator.
`rndi`	Returns random integers, $0 <= y < 2^{32}$.
`rndKMbeta`	Computes beta pseudo-random numbers.
`rndKMgam`	Computes gamma pseudo-random numbers.
`rndKMi`	Returns random integers, $0 <= y < 2^{32}$.
`rndKMn`	Computes standard normal pseudo-random numbers.
`rndKMnb`	Computes negative binomial pseudo-random numbers.
`rndKMp`	Computes Poisson pseudo-random numbers.
`rndKMu`	Computes uniform pseudo-random numbers.
`rndKMvm`	Computes von Mises pseudo-random numbers.
`rndLaplace`	Computes Laplacian pseudo-random numbers

	with the choice of underlying random number generator.
rndLogNorm	Computes lognormal pseudo-random numbers with the choice of underlying random number generator.
rndmult	Changes multiplier of the LC random number generator.
rndMVn	Computes multivariate normal random numbers given a covariance matrix.
rndn	Computes normally distributed pseudo-random numbers with a choice of underlying random number generator.
rndnb	Computes random numbers with negative binomial distribution.
rndNegBinomial	Computes negative binomial pseudo-random numbers with a choice of underlying random number generator.
rndp	Computes random numbers with Poisson distribution.
rndPoisson	Computes Poisson pseudo-random numbers with a choice of underlying random number generator.
rndseed	Changes seed of the LC random number generator.
rndStateSkip	To advance a state vector by a specified number of values.

rndu	Computes uniform random numbers with a choice of underlying random number generator.
rndWeibull	Computes Weibull pseudo-random numbers with the choice of underlying random number generator.

The random number generator can be seeded. Set the seed using `rndseed`. For example:

```
    rndseed 44435667;
x = rndu(1,1);
```

Fuzzy Conditional Functions

dotfeq	Fuzzy .==
dotfeqmt	Fuzzy .==
dotfge	Fuzzy .>=
dotfgemt	Fuzzy .>
dotfgt	Fuzzy .>
dotfgtmt	Fuzzy .>
dotfle	Fuzzy .<=
dotflemt	Fuzzy .<=
dotflt	Fuzzy .<
dotfltmt	Fuzzy .<
dotfne	Fuzzy ./=

`dotfnemt`	Fuzzy ./=
`feq`	Fuzzy ==
`feqmt`	Fuzzy ==
`fge`	Fuzzy >=
`fgemt`	Fuzzy >=
`fgt`	Fuzzy >
`fgtmt`	Fuzzy >
`fle`	Fuzzy <=
`flemt`	Fuzzy <=
`flt`	Fuzzy <
`fltmt`	Fuzzy <
`fne`	Fuzzy /=
`fnemt`	Fuzzy /=

The **`mt`** commands use an *`fcmptol`* argument to control the tolerance used for comparison.

The non-**`mt`** commands use the global variable *`_fcmptol`* to control the tolerance used for comparison. By default, this is 1e-15. The default can be changed by editing the file `fcompare.dec`.

Statistical Functions

`acf`	Computes sample autocorrelations.
`astd`	Computes the standard deviation of the

	elements across one dimension of an N-dimensional array.
`astds`	Computes the 'sample' standard deviation of the elements across one dimension of an N-dimensional array.
`ChiBarSquare`	Computes probability of chi-bar-square statistic.
`combinate`	Computes combinations of `n` things taken `k` at a time.
`combinated`	Writes combinations of `n` things taken `k` at a time to a **GAUSS** data set.
`ConScore`	Computes constrained score statistic and its probability.
`conv`	Computes convolution of two vectors.
`corrm`	Computes correlation matrix of a moment matrix.
`corrms`	Computes sample correlation matrix of a moment matrix.
`corrvc`	Computes correlation matrix from a variance- covariance matrix.
`corrx`	Computes correlation matrix.
`corrxs`	Computes sample correlation matrix.
`crossprd`	Computes cross product.
`design`	Creates a design matrix of 0's and 1's.

`dstat`	Computes descriptive statistics of a data set or matrix.
`dstatmt`	Computes descriptive statistics of a data set or matrix.
`dstatmtControlCreate`	Creates default **dstatmtControl** structure.
`gdaDStat`	Computes descriptive statistics on multiple Nx1 variables in a GDA.
`gdaDStatMat`	Computes descriptive statistics on a selection of columns in a variable in a GDA.
`loess`	Computes coefficients of locally weighted regression.
`loessmt`	Computes coefficients of locally weighted regression.
`loessmtControlCreate`	Creates default **loessmtControl** structure.
`meanc`	Computes mean value of each column of a matrix.
`median`	Computes medians of the columns of a matrix.
`moment`	Computes moment matrix ($x'x$) with special handling of missing values.
`momentd`	Computes moment matrix from a data set.
`movingave`	Computes moving average of a series.
`movingaveExpwgt`	Computes exponentially weighted moving average of a series.

`movingaveWgt`	Computes weighted moving average of a series.
`numCombinations`	Computes number of combinations of `n` things taken `k` at a time.
`ols`	Computes least squares regression of data set or matrix.
`olsmt`	Computes least squares regression of data set or matrix.
`olsmtControlCreate`	Creates default **olsmtControl** structure.
`olsqr`	Computes OLS coefficients using QR decomposition.
`olsqr2`	Computes OLS coefficients, residuals, and predicted values using QR decomposition.
`olsqrmt`	Computes OLS coefficients using QR decomposition.
`pacf`	Computes sample partial autocorrelations.
`princomp`	Computes principal components of a data matrix.
`quantile`	Computes quantiles from data in a matrix, given specified probabilities.
`quantiled`	Computes quantiles from data in a data set, given specified probabilities.
`rndvm`	Computes von Mises pseudo-random numbers.

stdc	Computes standard deviation of the columns of a matrix.
stdsc	Computes the 'sample' standard deviation of the elements in each column of a matrix.
toeplitz	Computes Toeplitz matrix from column vector.
varCovM	Computes the population variance-covariance matrix from a moment matrix.
varCovMS	Computes a sample variance-covariance matrix from a moment matrix.
varCovX	Computes the population variance-covariance matrix from a data matrix.
varCovXS	Computes a sample variance-covariance matrix from a data matrix.
varmall	Computes the log-likelihood of a Vector ARMA model.
varmares	Computes the residuals of a Vector ARMA model.
vcm	Computes a variance-covariance matrix from a moment matrix.
vcms	Computes a sample variance-covariance matrix from a moment matrix.
vcx	Computes a variance-covariance matrix from a data matrix.
vcxs	Computes a sample variance-covariance

	matrix from a data matrix.

Advanced statistics and optimization routines are available in the **GAUSS** Applications programs. (Contact Aptech Systems for more information.)

Optimization and Solution

`eqSolve`	Solves a system of nonlinear equations.
`eqSolvemt`	Solves a system of nonlinear equations.
`eqSolvemtControlCreate`	Creates default **eqSolvemtControl** structure.
`eqSolvemtOutCreate`	Creates default **eqSolvemtOut** structure.
`eqSolveSet`	Sets global input used by **`eqSolve`** to default values.
`linsolve`	Solves $Ax = b$ using the inverse function.
`ltrisol`	Computes the solution of $Lx = b$ where L is a lower triangular matrix.
`lusol`	Computes the solution of $LUx = b$ where L is a lower triangular matrix and U is an upper triangular matrix.
`QNewton`	Optimizes a function using the BFGS descent algorithm.

`QNewtonmt`	Minimizes an arbitrary function.
`QNewtonmtControlCreate`	Creates default **QNewtonmtControl** structure.
`QNewtonmtOutCreate`	Creates default **QNewtonmtOut** structure.
`QProg`	Solves the quadratic programming problem.
`QProgmt`	Solves the quadratic programming problem.
`QProgmtInCreate`	Creates an instance of a structure of type `QProgmtInCreate` with the `maxit` member set to a default value.
`sqpSolve`	Solves the nonlinear programming problem using a sequential quadratic programming method.
`sqpSolveMT`	Solves the nonlinear programming problem using a sequential quadratic programming method.
`sqpSolveMTControlCreate`	Creates an instance of a structure of type `sqpSolveMTcontrol` set to default values.
`sqpSolveMTlagrangeCreate`	Creates an instance of a structure of type `sqpSolveMTlagrange` set to default values.
`sqpSolveMToutCreate`	Creates an instance of a structure of type `sqpSolveMTout` set to default

	values.
sqpSolveSet	Resets global variables used by **sqpSolve** to default values.
utrisol	Computes the solution of $Ux = b$ where U is an upper triangular matrix.

Statistical Distributions

cdfBeta	Computes integral of beta function.
cdfBetaInv	Computes the quantile or inverse of the beta cumulative distribution function.
cdfBinomial	Computes the binomial cumulative distribution function.
cdfBinomialInv	Computes the binomial quantile or inverse cumulative distribution function.
cdfBvn	Computes lower tail of bivariate Normal cdf.
cdfBvn2	Returns cdfbvn of a bounded rectangle.
cdfBvn2e	Returns cdfbvn of a bounded rectangle.
cdfCauchy	Computes the cumulative distribution function for the Cauchy distribution.
cdfCauchyinv	Computes the Cauchy inverse cumulative distribution function.
cdfChic	Computes complement of cdf of χ^2.
cdfChii	Computes χ^2 abscissae values given probability and degrees of freedom.

`cdfChinc`	Computes integral of noncentral χ^2.
`cdfExp`	Computes the cumulative distribution function for the exponential distribution.
`cdfExpInv`	Computes the exponential inverse cumulative distribution function.
`cdfFc`	Computes complement of cdf of F.
`cdfFnc`	Computes integral of noncentral F.
`cdfFncInv`	Computes the quantile or inverse of noncentral F cumulative distribution function.
`cdfGam`	Computes integral of incomplete Γ function.
`cdfGenPareto`	Computes the cumulative distribution function for the Generalized Pareto distribution.
`cdfLaplace`	Computes the cumulative distribution function for the Laplace distribution.
`cdfLaplaceInv`	Computes the Laplace inverse cumulative distribution function.
`cdfMvn`	Computes multivariate Normal cdf.
`cdfMvnce`	Computes the complement of the multivariate Normal cumulative distribution function with error management
`cdfMvne`	Computes multivariate Normal cumulative distribution function with error management
`cdfMvn2e`	Computes the multivariate Normal cumulative distribution function with error management over

	the range [a,b]
`cdfMvtce`	Computes complement of multivariate Student's t cumulative distribution function with error management
`cdfMvte`	Computes multivariate Student's t cumulative distribution function with error management
`cdfMvt2e`	Computes multivariate Student's t cumulative distribution function with error management over [a,b]
`cdfN`	Computes integral of Normal distribution: lower tail, or cdf.
`cdfN2`	Computes interval of Normal cdf.
`cdfNc`	Computes complement of cdf of Normal distribution (upper tail).
`cdfNegBinomial`	Computes the cumulative distribution function for the negative binomial distribution.
`cdfNegBinomialInv`	Computes the quantile or inverse negative binomial cumulative distribution function.
`cdfNi`	Computes the inverse of the cdf of the Normal distribution.
`cdfRayleigh`	Computes the Rayleigh cumulative distribution function.
`cdfRayleighInv`	Computes the Rayleigh inverse cumulative distribution function.
`cdfTc`	Computes complement of cdf of `t`-distribution.

`cdfTci`	Computes the inverse of the complement of the Student's t cdf.
`cdfTnc`	Computes integral of noncentral `t`-distribution.
`cdfTvn`	Computes lower tail of trivariate Normal cdf.
`cdfWeibull`	Computes the cumulative distribution function for the Weibull distribution.
`cdfWeibullInv`	Computes the Weibull inverse cumulative distribution function.
`erf`	Computes Gaussian error function.
`erfc`	Computes complement of Gaussian error function.
`erfccplx`	Computes complement of Gaussian error function for complex inputs.
`erfcplx`	Computes Gaussian error function for complex inputs.
`lncdfbvn`	Computes natural log of bivariate Normal cdf.
`lncdfbvn2`	Returns log of cdfbvn of a bounded rectangle.
`lncdfmvn`	Computes natural log of multivariate Normal cdf.
`lncdfn`	Computes natural log of Normal cdf.
`lncdfn2`	Computes natural log of interval of Normal cdf.
`lncdfnc`	Computes natural log of complement of Normal cdf.

`lnpdfmvn`	Computes multivariate Normal log-probabilities.
`lnpdfmvt`	Computes multivariate Student's `t` log-probabilities.
`lnpdfn`	Computes Normal log-probabilities.
`lnpdft`	Computes Student's `t` log-probabilities.
`pdfCauchy`	Computes the probability density function for the Cauchy distribution.
`pdfexp`	Computes the probability density function for the exponential distribution.
`pdfgam`	Computes the probability density function for the Gamma distribution.
`pdfGenPareto`	Computes the probability density function for the Generalized Pareto distribution.
`pdfLaplace`	Computes the probability density function for the Laplace distribution.
`pdflogistic`	Computes the probability density function for the logistic distribution.
`pdfn`	Computes standard Normal probability density function.
`pdfPoisson`	Computes the probability density function for the Poisson distribution.
`pdfPoissonInv`	Computes the quantile or inverse Poisson cumulative distribution function.
`pdfRayleigh`	Computes the probability density function of the

	Rayleigh distribution.
`pdfWeibull`	Computes the probability density function of a Weibull random variable.

Series and Sequence Functions

`recserar`	Computes autoregressive recursive series.
`recsercp`	Computes recursive series involving products.
`recserrc`	Computes recursive series involving division.
`seqa`	Creates an additive sequence.
`seqm`	Creates a multiplicative sequence.

Precision Control

`base10`	Converts number to `x.xxx` and a power of 10.
`ceil`	Rounds up towards +∞.
`floor`	Rounds down towards -∞.
`machEpsilon`	Returns the smallest number such that 1+ `eps`>1.
`round`	Rounds to the nearest integer.
`trunc`	Converts numbers to integers by truncating the fractional portion.

`round`, **`trunc`**, **`ceil`** and **`floor`** convert floating point numbers into integers. The internal representation for the converted integer is double precision (64 bits).

Each matrix element in memory requires 8 bytes of memory.

37.2 Finance Functions

`AmericanBinomCall`	American binomial method Call.
`AmericanBinomCall_Greeks`	American binomial method call Delta, Gamma, Theta, Vega, and Rho.
`AmericanBinomCall_ImpVol`	Implied volatilities for American binomial method calls.
`AmericanBinomPut`	American binomial method Put.
`AmericanBinomPut_Greeks`	American binomial method put Delta, Gamma, Theta, Vega, and Rho.
`AmericanBinomPut_ImpVol`	Implied volatilities for American binomial method puts.
`AmericanBSCall`	American Black and Scholes Call.
`AmericanBSCall_Greeks`	American Black and Scholes call Delta, Gamma, Omega, Theta, and Vega.
`AmericanBSCall_ImpVol`	Implied volatilities for American Black and Scholes calls.
`AmericanBSPut`	American Black and Scholes Put.
`AmericanBSPut_Greeks`	American Black and Scholes put Delta, Gamma, Omega, Theta, and Vega.

`AmericanBSPut_ImpVol`	Implied volatilities for American Black and Scholes puts.
`annualTradingDays`	Computes number of trading days in a given year.
`elapsedTradingDays`	Computes number of trading days between two dates inclusively.
`EuropeanBinomCall`	European binomial method call.
`EuropeanBinomCall_Greeks`	European binomial method call Delta, Gamma, Theta, Vega and Rho.
`EuropeanBinomCall_ImpVol`	Implied volatilities for European binomial method calls.
`EuropeanBinomPut`	European binomial method Put.
`EuropeanBinomPut_Greeks`	European binomial method put Delta, Gamma, Theta, Vega, and Rho.
`EuropeanBinomPut_ImpVol`	Implied volatilities for European binomial method puts.
`EuropeanBSCall`	European Black and Scholes Call.
`EuropeanBSCall_Greeks`	European Black and Scholes call Delta, Gamma, Omega, Theta, and Vega.
`EuropeanBSCall_ImpVol`	Implied volatilities for European Black and Scholes calls.
`EuropeanBSPut`	European Black and Scholes Put.
`EuropeanBSPut_Greeks`	European Black and Scholes put

	Delta, Gamma, Omega, Theta, and Vega.
`EuropeanBSPut_ImpVol`	Implied volatilities for European Black and Scholes puts.
`getNextTradingDay`	Returns the next trading day.
`getNextWeekDay`	Returns the next day that is not on a weekend.
`getPreviousTradingDay`	Returns the previous trading day.
`getPreviousWeekDay`	Returns the previous day that is not on a weekend.

37.3 Matrix Manipulation

Creating Vectors and Matrices

`eye`	Creates identity matrix.
`let`	Creates matrix from list of constants.
`matalloc`	Allocates a matrix with unspecified contents.
`matinit`	Allocates a matrix with specified fill value.
`ones`	Creates a matrix of ones.
`zeros`	Creates a matrix of zeros.

Use **`zeros`**, **`ones`**, or **`matinit`** to create a constant vector or matrix.

Matrices can also be loaded from an ASCII file, from a **GAUSS** matrix file, or from a **GAUSS** data set. (See **File I/O**, Chapter 1, for more information.)

Loading and Storing Matrices

`asciiload`	Loads data from a delimited ASCII text file into an Nx1 vector.
`dataload`	Loads matrices, N-dimensional arrays, strings and string arrays from a disk file.
`datasave`	Saves matrices, N-dimensional arrays, strings and string arrays to a disk file.
`load, loadm`	Loads matrix from ASCII or matrix file.
`loadd`	Loads matrix from data set.
`loadf`	Loads function from disk file.
`loadk`	Loads keyword from disk file.
`save`	Saves symbol to disk file.
`saved`	Saves matrix to data set.

Size, Ranking, and Range

`cols`	Returns number of columns in a matrix.
`colsf`	Returns number of columns in an open data set.

`counts`	Returns number of elements of a vector falling in specified ranges.
`countwts`	Returns weighted count of elements of a vector falling in specified ranges.
`cumprodc`	Computes cumulative products of each column of a matrix.
`cumsumc`	Computes cumulative sums of each column of a matrix.
`indexcat`	Returns indices of elements falling within a specified range.
`maxc`	Returns largest element in each column of a matrix.
`maxindc`	Returns row number of largest element in each column of a matrix.
`minc`	Returns smallest element in each column of a matrix.
`minindc`	Returns row number of smallest element in each column of a matrix.
`prodc`	Computes the product of each column of a matrix.
`rankindx`	Returns rank index of Nx1 vector. (Rank order of elements in vector).
`rows`	Returns number of rows in a matrix.
`rowsf`	Returns number of rows in an open data set.

`sumc`	Computes the sum of each column of a matrix.
`sumr`	Computes the sum of each row of a matrix.

These functions are used to find the minimum, maximum and frequency counts of elements in matrices.

Use **`rows`** and **`cols`** to find the number of rows or columns in a matrix. Use **`rowsf`** and **`colsf`** to find the numbers of rows or columns in an open **GAUSS** data set.

Miscellaneous Matrix Manipulation

`complex`	Creates a complex matrix from two real matrices.
`delif`	Deletes rows from a matrix using a logical expression.
`delrows`	Deletes rows from a matrix; the second argument contains the indices of the rows to be deleted.
`diag`	Extracts the diagonal of a matrix.
`diagrv`	Puts a column vector into the diagonal of a matrix.
`exctsmpl`	Creates a random subsample of a data set, with replacement.
`imag`	Returns the imaginary part of a complex matrix.
`indcv`	Checks one character vector against

	another and returns the indices of the elements of the first vector in the second vector.
`indnv`	Checks one numeric vector against another and returns the indices of the elements of the first vector in the second vector.
`intrsect`	Returns the intersection of two vectors.
`lowmat`	Returns the main diagonal and lower triangle.
`lowmat1`	Returns a main diagonal of 1's and the lower triangle.
`putvals`	Inserts values into a matrix or N-dimensional array.
`real`	Returns the real part of a complex matrix.
`reshape`	Reshapes a matrix to new dimensions.
`rev`	Reverses the order of rows of a matrix.
`rotater`	Rotates the rows of a matrix, wrapping elements as necessary.
`selif`	Selects rows from a matrix using a logical expression.
`setdif`	Returns elements of one vector that are

	not in another.
`shiftr`	Shifts rows of a matrix, filling in holes with a specified value.
`submat`	Extracts a submatrix from a matrix.
`subvec`	Extracts an Nx1 vector of elements from an NxK matrix.
`trimr`	Trims rows from top or bottom of a matrix.
`union`	Returns the union of two vectors.
`upmat`	Returns the main diagonal and upper triangle.
`upmat1`	Returns a main diagonal of 1's and the upper triangle.
`vec`	Stacks columns of a matrix to form a single column.
`vech`	Reshapes the lower triangular portion of a symmetric matrix into a column vector.
`vecr`	Stacks rows of a matrix to form a single column.
`vget`	Extracts a matrix or string from a data buffer constructed with **`vput`**.
`vlist`	Lists the contents of a data buffer constructed with **`vput`**.

vnamecv	Returns the names of the elements of a data buffer constructed with **vput**.
vput	Inserts a matrix or string into a data buffer.
vread	Reads a string or matrix from a data buffer constructed with **vput**.
vtypecv	Returns the types of the elements of a data buffer constructed with **vput**.
xpnd	Expands a column vector into a symmetric matrix.

vech and **xpnd** are complementary functions. **vech** provides an efficient way to store a symmetric matrix; **xpnd** expands the stored vector back to its original symmetric matrix.

delif and **selif** are complementary functions. **delif** deletes rows of a matrix based on a logical comparison; **selif** selects rows based on a logical comparison.

lowmat, **lowmat1**, **upmat**, and **upmat1** extract triangular portions of a matrix.

To delete rows which contain missing values from a matrix in memory, see **packr**.

37.4 Sparse Matrix Handling

denseToSp	Converts a dense matrix to a sparse matrix.
denseToSpRE	Converts a dense matrix to a sparse matrix using a relative epsilon.
packedToSp	Creates a sparse matrix from a packed

	matrix of non-zero values and row and column indices.
`spBiconjGradSol`	Solves the system of linear equations *Ax*=*b* using the biconjugate gradient method.
`spChol`	Computes the LL' decomposition of a sparse matrix.
`spConjGradSol`	Solves the system of linear equations *Ax*=*b* for symmetric matrices using the conjugate gradient method.
`spCreate`	Creates a sparse matrix from vectors of non-zero values, row indices, and column indices.
`spDenseSubmat`	Returns a dense submatrix of a sparse matrix.
`spDiagRvMat`	Inserts submatrices along the diagonal of a sparse matrix.
`spEigv`	Computes a specified number of eigenvalues and eigenvectors of a square, sparse matrix.
`spEye`	Creates a sparse identity matrix.
`spGetNZE`	Returns the non-zero values in a sparse matrix, as well as their corresponding row and column indices.
`spGetNumNZE`	Returns the number of non-zero elements in a sparse matrix.

`spLDL`	Computes the LDL decomposition of a symmetric sparse matrix.
`spLU`	Computes the LU decomposition of a sparse matrix with partial pivoting.
`spOnes`	Generates a sparse matrix containing only ones and zeros
`spSubmat`	Returns a sparse submatrix of sparse matrix.
`spToDense`	Converts a sparse matrix to a dense matrix.
`spTrTDense`	Multiplies a sparse matrix transposed by a dense matrix.
`spTScalar`	Multiplies a sparse matrix by a scalar.
`spZeros`	Creates a sparse matrix containing no non-zero values.

37.5 N-Dimensional Array Handling

Creating Arrays

`aconcat`	Concatenates conformable matrices and arrays in a user-specified dimension.
`aeye`	Creates an N-dimensional array in which the planes described by the two trailing dimensions of the array are

	equal to the identity.
`areshape`	Reshapes a scalar, matrix, or array into an array of user-specified size.
`arrayalloc`	Creates an N-dimensional array with unspecified contents.
`arrayinit`	Creates an N-dimensional array with a specified fill value.
`mattoarray`	Converts a matrix to a type array.

Size, Ranking and Range

`amax`	Moves across one dimension of an N-dimensional array and finds the largest element.
`amin`	Moves across one dimension of an N-dimensional array and finds the smallest element.
`asum`	Computes the sum across one dimension of an N-dimensional array.
`getdims`	Gets the number of dimensions in an array.
`getorders`	Gets the vector of orders corresponding to an array.

Setting and Retrieving Data in an Array

`aconcat`	Concatenates conformable matrices

	and arrays in a user-specified dimension.
areshape	Reshapes a scalar, matrix, or array into an array of user-specified size.
arraytomat	Changes an array to type matrix.
getarray	Gets a contiguous subarray from an N-dimensional array.
getmatrix	Gets a contiguous matrix from an N-dimensional array.
getmatrix4D	Gets a contiguous matrix from a 4-dimensional array.
getscalar3D	Gets a scalar from a 3-dimensional array.
getscalar4D	Gets a scalar from a 4-dimensional array.
putarray	Puts a contiguous subarray into an N-dimensional array and returns the resulting array.
setarray	Sets a contiguous subarray of an N-dimensional array.

Miscellaneous Array Functions

amean	Computes the mean across one dimension of an N-dimensional array.
amult	Performs matrix multiplication on the

	planes described by the two trailing dimensions of N-dimensional arrays.
`arrayindex`	Saves a matrix of structures to a file on the disk.
`atranspose`	Transposes an N-dimensional array.
`loopnextindex`	Increments an index vector to the next logical index and jumps to the specified label if the index did not wrap to the beginning.
`nextindex`	Returns the index of the next element or subarray in an array.
`previousindex`	Returns the index of the previous element or subarray in an array.
`singleindex`	Converts a vector of indices for an N-dimensional array to a scalar vector index.
`walkindex`	Walks the index of an array forward or backward through a specified dimension.

37.6 Structures

`dsCreate`	Creates an instance of a structure of type **DS** set to default values.
`loadstruct`	Loads a structure into memory from a file on the disk.

`pvCreate`	Returns an initialized an instance of structure of type **PV**.
`pvGetIndex`	Gets row indices of a matrix in a parameter vector.
`pvGetParNames`	Generates names for parameter vector stored in structure of type **PV**.
`pvGetParVector`	Retrieves parameter vector from structure of type **PV**.
`pvLength`	Returns the length of a parameter vector.
`pvList`	Retrieves names of packed matrices in structure of type **PV**.
`pvPack`	Packs general matrix into a structure of type **PV** with matrix name.
`pvPacki`	Packs general matrix or array into a **PV** instance with name and index.
`pvPackm`	Packs general matrix into a structure of type **PV** with a mask and matrix name.
`pvPackmi`	Packs general matrix or array into a **PV** instance with a mask, name, and index.
`pvPacks`	Packs symmetric matrix into a structure of type **PV**.
`pvPacksi`	Packs symmetric matrix into a **PV**

	instance with matrix name and index.
`pvPacksm`	Packs symmetric matrix into a structure of type **PV** with a mask.
`pvPacksmi`	Packs symmetric matrix into a **PV** instance with a mask, matrix name, and index.
`pvPutParVector`	Inserts parameter vector into structure of type **PV**.
`pvTest`	Tests an instance of structure of type **PV** to determine if it is a proper structure of type **PV**.
`pvUnpack`	Unpacks matrices stored in a structure of type **PV**.
`savestruct`	Saves a matrix of structures to a file on the disk.

37.7 Data File Reading/Writing

Spreadsheets

`SpreadsheetReadM`	Reads and writes Excel files.
`SpreadsheetReadSA`	Reads and writes Excel files.
`SpreadsheetWrite`	Reads and writes Excel files.
`xlsGetSheetCount`	Gets the number of sheets in an Excel spreadsheet.

`xlsGetSheetSize`	Gets the size (rows and columns) of a specified sheet in an Excel spreadsheet.
`xlsGetSheetTypes`	Gets the cell format types of a row in an Excel spreadsheet.
`xlsMakeRange`	Builds an Excel range string from a row/column pair.
`xlsReadM`	Reads from an Excel spreadsheet, into a **GAUSS** matrix.
`xlsReadSA`	Reads from an Excel spreadsheet, into a **GAUSS** string array or string.
`xlsWrite`	Writes a **GAUSS** matrix, string, or string array to an Excel spreadsheet.
`xlsWriteM`	Writes a **GAUSS** matrix to an Excel spreadsheet.
`xlsWriteSA`	Writes a **GAUSS** string or string array to an Excel spreadsheet.

Text Files

`fcheckerr`	Gets the error status of a file.
`fclearerr`	Gets the error status of a file, then clears it.
`fflush`	Flushes a file's output buffer.
`fgets`	Reads a line of text from a file.

`fgetsa`	Reads lines of text from a file into a string array.
`fgetsat`	Reads lines of text from a file into a string array.
`fgetst`	Reads a line of text from a file.
`fopen`	Opens a file.
`fputs`	Writes strings to a file.
`fputst`	Writes strings to a file.
`fseek`	Positions the file pointer in a file.
`fstrerror`	Returns an error message explaining the cause of the most recent file I/O error.
`ftell`	Gets the position of the file pointer in a file.

Database

This section summarizes all procedures within the GAUSS database module. A general usage description will be found in **Databases with GAUSS**, Section 1 .

Database Setup

`dbAddDatabase`	Adds a database to the list of database connections using the driver type or a connection URL.
`dbGetDrivers`	Returns a list of available database drivers.

`dbIsDriverAvailable`	Returns 1 if a specified database driver is available.
`dbRemoveDatabase`	Removes a database connection from the list of open database connections. Frees all related resources.

Database Properties

`dbGetConnectOptions`	Returns the connection options string used for a database connection.
`dbGetDatabaseName`	Returns the name of the database.
`dbGetDriverName`	Returns the name of the connection's database driver.
`dbGetHostName`	Returns the database connection's host name.
`dbGetPassword`	Returns a connection's password.
`dbGetNumericalPrecPolicy`	Returns the default numerical precision policy for a specified database connection.
`dbGetPort`	Returns the database connection's port number if it has been set.
`dbIsOpen`	Reports whether a specified database connection is open.
`dbIsValid`	Reports whether a specified database connection has a valid driver.

`dbSetConnectOptions`	Sets database-specific options.
`dbSetDatabaseName`	Sets the connection's database name to name.
`dbSetHostName`	Sets the specified database connection's host name.
`dbSetNumericalPrecPolicy`	Sets the default numerical precision policy used by queries created on this database connection.
`dbSetPassword`	Sets the database connection's password.
`dbSetPort`	Sets the specified database connection's port number.

Database Information

`dbGetPrimaryIndex`	Returns the primary index for the specified table.
`dbGetTableHeaders`	Returns a string array populated with the names of all the fields in a specified table (or view).
`dbGetTables`	Returns the database's tables, system tables and views.
`dbHasFeature`	Returns a 1 if the database supports the specified feature.

Database Errors

`dbGetLastErrorNum`	Returns numerical information about the last error that occurred on the database.
`dbGetLastErrorText`	Returns text information about the last error that occurred on the database.
`dbIsOpenError`	Reports whether an error occurred while attempting to open the database connection.
`dbQueryGetLastErrorNum`	Returns numerical error information about the last error that occurred (if any) with the last executed query.
`dbQueryGetLastErrorText`	Returns text error information about the last error that occurred (if any) with the last executed query.

Database Connect

`dbClose`	Closes a database connection and destroys any remaining queries.
`dbOpen`	Opens a specified database connection using the current connection values.

Database Transaction

`dbCommit`	Commits a transaction to the database if the driver supports transactions and a **`dbTransaction()`** has been started.

`dbCreateQuery`	Process an SQL statement and prepare a query.
`dbExecQuery`	Executes an SQL statement and creates a query.
`dbRollback`	Rolls back a transaction on the database.
`dbTransaction`	Begins a transaction on the database.

Query Building

`dbQueryBindValue`	Set the placeholder placeholder to be bound to value val in the prepared statement.
`dbQueryGetBoundValue`	Returns the value for a placeholder in a query.
`dbQueryGetBoundValues`	Returns an Nx2 string array containing the placeholders and their corresponding values in a query.
`dbQueryExecPrepared`	Executes a previously created and prepared query.
`dbQueryPrepare`	Prepares a SQL query for execution.

Query Manipulation

`dbQueryClear`	Clears the result set and releases any resources held by the query. Sets the

	query state to inactive.
`dbQueryFinish`	Instructs the database driver that no more data will be fetched from this query until it is re-executed.

Query Information

`dbQueryCols`	Returns the number of fields in the record.
`dbQueryGetLastInsertID`	Returns the object ID of the most recent inserted row if supported by the database.
`dbQueryGetLastQuery`	Returns the text of the current query being used.
`dbQueryGetNumRowsAffected`	Reports the number of rows affected by the result's SQL statement.
`dbQueryIsActive`	Returns 1 if the query is active.
`dbQueryIsForwardOnly`	Reports whether you can only scroll forward through a result set.
`dbQueryIsNull`	Reports whether the current field pointed at by an active query positioned on a valid record is NULL.
`dbQueryIsSelect`	Reports whether the specified query is a SELECT statement.
`dbQueryIsValid`	Reports whether the specified query is positioned on a valid record.

`dbQueryRows`	Returns the size of the result (number of rows returned), or -1 if the size cannot be determined or if the database does not support reporting information about query sizes.
`dbQuerySetForwardOnly`	Sets forward only mode to forward. If forward is true, only **`dbQuerySeekNext`**() and **`dbQuerySeek`**() with positive values, are allowed for navigating the results.

Query Iteration

`dbQueryGetPosition`	Returns the current internal position of the query.
`dbQuerySeek`	Retrieves the record at a specified position, if available, and positions the query on the retrieved record.
`dbQuerySeekFirst`	Retrieves the first record in the result, if available, and positions the query on the retrieved record.
`dbQuerySeekLast`	Retrieves the last record in the result, if available, and positions the query on the retrieved record.
`dbQuerySeekNext`	Retrieves the next record in the result, if available, and positions the query on the retrieved record.
`dbQuerySeekPrevious`	Retrieves the previous record in the

	result, if available, and positions the query on the retrieved record.

Query Data Retrieval

`dbQueryFetchAllM`	Returns the result set for the current query as a matrix.
`dbQueryFetchAllSA`	Returns the result set for the current query as a string array.
`dbQueryFetchOneM`	Returns a single row as an Nx1 matrix where N is the column count of the SELECT statement.
`dbQueryFetchOneSA`	Returns a single row as a string vector containing the field information for the current query.
`dbQueryGetField`	Returns the value of a specified field in the current record.

GAUSS Data Archives

`gdaAppend`	Appends data to a variable in a GDA.
`gdaCreate`	Creates a GDA.
`gdaDStat`	Computes descriptive statistics on multiple Nx1 variables in a GDA.
`gdaDStatMat`	Computes descriptive statistics on a selection of columns in a variable in a GDA.

`gdaGetIndex`	Gets the index of a variable in a GDA.
`gdaGetName`	Gets the name of a variable in a GDA.
`gdaGetNames`	Gets the names of all the variables in a GDA.
`gdaGetOrders`	Gets the orders of a variable in a GDA.
`gdaGetType`	Gets the type of a variable in a GDA.
`gdaGetTypes`	Gets the types of all the variables in a GDA.
`gdaGetVarInfo`	Gets information about all of the variables in a GDA.
`gdaIsCplx`	Checks to see if a variable in a GDA is complex.
`gdaLoad`	Loads variables in a GDA into the workspace.
`gdaPack`	Packs the data in a GDA, removing all empty bytes.
`gdaRead`	Gets a variable from a GDA.
`gdaReadByIndex`	Gets a variable from a GDA, given a variable index.
`gdaReadSome`	Reads part of a variable from a GDA.
`gdaReadSparse`	Gets a sparse matrix from a **GAUSS**

	Data Archive.
`gdaReadStruct`	Gets a structure from a **GAUSS** Data Archive.
`gdaReportVarInfo`	Gets information about all of the variables in a **GAUSS** Data Archive and returns it in a string array formatted for printing.
`gdaSave`	Writes variables in a workspace to a GDA.
`gdaUpdate`	Updates a variable in a GDA.
`gdaUpdateAndPack`	Updates a variable in a GDA, leaving no empty bytes if the updated variable is smaller or larger than the variable it is replacing.
`gdaWrite`	Writes a variable to a GDA.
`gdaWrite32`	Writes a variable to a GDA using 32-bit system file write commands.
`gdaWriteSome`	Overwrites part of a variable in a GDA.

These functions all operate on **GAUSS** Data Archives (GDA's). For more information, see **GAUSS Data Archives**, Section 1.0.1.

Data Sets

`close`	Closes an open data set (`.dat` file).

`closeall`	Closes all open data sets.
`create`	Creates and opens a data set.
`datacreate`	Creates a **`v96`** real data set.
`datacreatecomplex`	Creates a **`v96`** complex data set.
`datalist`	Lists selected variables from a data set.
`dataopen`	Opens a data set.
`eof`	Tests for end of file.
`getnr`	Computes number of rows to read per iteration for a program that reads data from a disk file in a loop.
`getnrmt`	Computes number of rows to read per iteration for a program that reads data from a disk file in a loop.
`iscplxf`	Returns whether a data set is real or complex.
`loadd`	Loads a small data set.
`open`	Opens an existing data set.
`readr`	Reads rows from open data set.
`saved`	Creates small data sets.
`seekr`	Moves pointer to specified location in open data set.
`tempname`	Creates a temporary file with a unique

	name.
`typef`	Returns the element size (2, 4 or 8 bytes) of data in open data set.
`writer`	Writes matrix to an open data set.

These functions all operate on **GAUSS**data sets (`.dat` files). For more information, see **File I/O**, Chapter 1.

To create a **GAUSS** data set from a matrix in memory, use **`saved`**. To create a data set from an existing one, use `create`. To create a data set from a large ASCII file, use the ATOG utility (see **ATOG**, Chapter 1.)

Data sets can be opened, read from, and written to using `open`, **`readr`**, **`seekr`** and **`writer`**. Test for the end of a file using **`eof`**, and close the data set using `close` or `closeall`.

The data in data sets may be specified as character or numeric. (See **File I/O**, Chapter 1.) See also `create` and **`vartypef`**.

`typef` returns the element size of the data in an open data set.

Data Set Variable Names

`getname`	Returns column vector of variable names in a data set.
`getnamef`	Returns string array of variable names in a data set.
`indices`	Retrieves column numbers and names from a data set.
`indices2`	Similar to **`indices`**, but matches columns with names for dependent and independent variables.

`indicesf`	Retrieves column numbers and names from a data set.
`indicesfn`	Retrieves column numbers and names from a data set.
`makevars`	Decomposes matrix to create column vectors.
`setvars`	Creates globals using the names in a data set.
`vartypef`	Returns column vector of variable types (numeric/character) in a data set.

Use **`getnamef`** to retrieve the variable names associated with the columns of a **GAUSS** data set and **`vartypef`** to retrieve the variable types. Use **`makevars`** and **`setvars`** to create global vectors from those names. Use **`indices`** and **`indices2`** to match names with column numbers in a data set.

Data Coding

`code`	Codes the data in a vector by applying a logical set of rules to assign each data value to a category.
`code(dataloop)`	Creates new variables with different values based on a set of logical expressions.
`dataloop (dataloop)`	Specifies the beginning of a data loop.
`delete (dataloop)`	Removes specific rows in a data loop

	based on a logical expression.
`drop (dataloop)`	Specifies columns to be dropped from the ouput data set in a data loop.
`dummy`	Creates a dummy matrix, expanding values in vector to rows with ones in columns corresponding to true categories and zeros elsewhere.
`dummybr`	Similar to **`dummy`**.
`dummydn`	Similar to **`dummy`**.
`extern (dataloop)`	Allows access to matrices or strings in memory from inside a data loop.
`isinfnanmiss`	Returns true if the argument contains an infinity, NaN, or missing value.
`scalmiss`	Returns 1 if matrix has any missing values, 0 otherwise.
`keep (dataloop)`	Specifies columns (variables) to be saved to the output data set in a data loop.
`lag (dataloop)`	Lags variables a specified number of periods.
`lag1`	Lags a matrix by one time period for time series analysis.
`lagn`	Lags a matrix a specified number of time periods for time series analysis.
`listwise (dataloop)`	Controls listwise deletion of missing

	values.
`make (dataloop)`	Specifies the creation of a new variable within a data loop.
`miss`	Changes specified values to missing value code.
`missex`	Changes elements to missing value using logical expression.
`missrv`	Changes missing value codes to specified values.
`msym`	Sets symbol to be interpreted as missing value.
`outtyp (dataloop)`	Specifies the precision of the output data set.
`packr`	Delete rows with missing values.
`recode`	Similar to **`code`**, but leaves the original data in place if no condition is met.
`recode (dataloop)`	Changes the value of a variable with different values based on a set of logical expressions.
`scalinfnanmiss`	Returns true if the argument is a scalar infinity, NaN, or missing value.
`scalmiss`	Tests whether a scalar is the missing value code.
`select (dataloop)`	Selects specific rows (observations) in

	a data loop based on a logical expression.
`subscat`	Simpler version of **`recode`**, but uses ascending bins instead of logical conditions.
`substute`	Similar to **`recode`**, but operates on matrices.
`vector (dataloop)`	Specifies the creation of a new variable within a data loop.

`code`, **`recode`**, and **`subscat`** allow the user to code data variables and operate on vectors in memory. **`substute`** operates on matrices, and **`dummy`**, **`dummybr`** and **`dummydn`** create matrices.

`missex`, **`missrv`** and **`miss`** should be used to recode missing values.

Sorting and Merging

`intrleav`	Produces one large sorted data file from two smaller sorted files having the same keys.
`intrleavsa`	Interleaves the rows of two string arrays that have been sorted on a common column.
`mergeby`	Produces one large sorted data file from two smaller sorted files having a single key column in common.
`mergevar`	Accepts a list of names of global matrices, and concatenates the

	corresponding matrices horizontally to form a single matrix.
`sortc`	Quick-sorts rows of matrix based on numeric key.
`sortcc`	Quick-sorts rows of matrix based on character key.
`sortd`	Sorts data set on a key column.
`sorthc`	Heap-sorts rows of matrix based on numeric key.
`sorthcc`	Heap-sorts rows of matrix based on character key.
`sortind`	Returns a sorted index of a numeric vector.
`sortindc`	Returns a sorted index of a character vector.
`sortmc`	Sorts rows of matrix on the basis of multiple columns.
`sortr`	Sorts rows of a matrix of numeric data.
`sortrc`	Sorts rows of a matrix of character data.
`uniqindx`	Returns a sorted unique index of a vector.
`uniqindxsa`	Computes the sorted index of a string vector, omitting duplicate elements.

`unique`	Removes duplicate elements of a vector.
`uniquesa`	Removes duplicate elements from a string vector.

`sortc`, **`sorthc`**, and **`sortind`** operate on numeric data only. **`sortcc`**, **`sorthcc`**, and **`sortindc`** operate on character data only.

`sortd`, **`sortmc`**, **`unique`**, and **`uniqindx`** operate on both numeric and character data.

Use **`sortd`** to sort the rows of a data set on the basis of a key column.

Both **`intrleav`** and **`mergeby`** operate on data sets.

37.8 Compiler Control

`#define`	Defines a case-insensitive text-replacement or flag variable.
`#definecs`	Defines a case-sensitive text-replacement or flag variable.
`#else`	Alternates clause for `#if-#else-#endif` code block.
`#endif`	End of `#if-#else-#endif` code block.
`#ifdef`	Compiles code block if a variable has been `#define`'d.
`#iflight`	Compiles code block if running **GAUSS** Light.

`#ifndef`	Compiles code block if a variable has not been `#define`'d.
`#ifos2win`	Compiles code block if running Windows.
`#ifunix`	Compiles code block if running UNIX.
`#include`	Includes code from another file in program.
`#linesoff`	Compiles program without line number and file name records.
`#lineson`	Compiles program with line number and file name records.
`#srcfile`	Inserts source file name record at this point (currently used when doing data loop translation).
`#srcline`	Inserts source file line number record at this point (currently used when doing data loop translation).
`#undef`	Undefines a text-replacement or flag variable.

These commands are compiler directives. That is, they do not generate **GAUSS** program instructions; rather, they are instructions that tell **GAUSS** how to process a program during compilation. They determine what the final compiled form of a program will be. They are not executable statements and have no effect at run-time. (See **Compiler Directives**, Chapter 1.1, for more information.)

37.9 Multi-Threading

ThreadBegin	Marks beginning of a block of code to be executed as a thread.
ThreadEnd	Marks end of a block of code to be executed as a thread.
ThreadJoin	Completes definition of a set of threads, waits for their work.
ThreadStat	Marks a single statement to be executed as a thread.

Together, ThreadBegin/ThreadEnd and ThreadStat define a set of threads that will execute simultaneously. ThreadJoin completes the definition of that set. ThreadJoin waits for the threads in the set to finish their calculations, the results of which are then available for further use.

```
ThreadBegin; // Thread 1
   y = x'x;
   z = y'y;
ThreadEnd;
ThreadBegin; // Thread 2
   q = r'r;
   r = q'q;
ThreadEnd;
ThreadStat n = m'm; // Thread 3
ThreadStat p = o'o; // Thread 4
ThreadJoin; // waits for Threads 1-4 to finish

b = z + r + n'p; // Using the results
```

37.10 Program Control

Execution Control

`call`	Calls function and discards return values.
`end`	Terminates a program and closes all files.
`pause`	Pauses for the specified time.
`run`	Runs a program in a text file.
`sleep`	Sleeps for the specified time.
`stop`	Stops a program and leaves files open.
`system`	Quits and returns to the OS.

Both `stop` and `end` will terminate the execution of a program; `end` will close all open files, and `stop` will leave those files open. Neither `stop` nor `end` is required in a **GAUSS** program.

Branching

`goto`	Unconditional branching.
`if...endif`	Conditional branching.
`pop`	Retrieves `goto` arguments.

```
if iter > itlim;
   goto errout("Iteration limit exceeded");
elseif iter =\,= 1;
   j = setup(x,y);
```

```
    else;
       j = iterate(x,y);
    endif;
     .
     .
     .
    errout:

    pop errmsg;
    print errmsg;
    end;
```

Looping

`break`	Jumps out the bottom of a `do` or `for` loop.
`continue`	Jumps to the top of a `do` or `for` loop.
`do while...endo`	Executes a series of statements in a loop as long as a given expression is TRUE (or FALSE).
`do until...endo`	Loops if FALSE.
`for...endfor`	Loops with integer counter.

```
    iter = 0;
    do while dif > tol;
       { x,x0 } = eval(x,x0);
       dif = abs(x-x0);
       iter = iter + 1;
       if iter > maxits;
          break;
       endif;
       if not prtiter;
```

```
            continue;
        endif;
        format /rdn 1,0;
        print "Iteration: " iter;;
        format /re 16,8;
        print ", Error: " maxc(dif);
    endo;

    for i (1, cols(x), 1);
        for j (1, rows(x), 1);
            x[i,j] = x[i,j] + 1;
        endfor;
    endfor;
```

Subroutines

`gosub`	Branches to subroutine.
`pop`	Retrieves `gosub` arguments.
`return`	Returns from subroutine.

Arguments can be passed to subroutines in the branch to the subroutine label and then popped, in first-in-last-out order, immediately following the subroutine label definition. See `gosub`.

Arguments can then be returned in an analogous fashion through the `return` statement.

Procedures, Keywords, and Functions

`endp`	Terminates a procedure definition.
`fn`	Allows user to create one-line functions.
`keyword`	Begins the definition of a keyword

	procedure. Keywords are user-defined functions with local or global variables.
`local`	Declares variables local to a procedure.
`proc`	Begins definition of multi-line procedure.
`retp`	Returns from a procedure.

Here is an example of a **GAUSS**procedure:

```
proc (3) = crosprod(x,y);
   local r1, r2, r3;
   r1 = x[2,.].*y[3,.]-x[3,.].*y[2,.];
   r2 = x[3,.].*y[1,.]-x[1,.].*y[3,.];
   r3 = x[1,.].*y[2,.]-x[2,.].*y[1,.];
  retp( r1,r2,r3);
endp;
```

The "(3) = " indicates that the procedure returns three arguments. All local variables, except those listed in the argument list, must appear in the `local` statement. Procedures may reference global variables. There may be more than one **`retp`** per procedure definition; none is required if the procedure is defined to return 0 arguments. The `endp` is always necessary and must appear at the end of the procedure definition. Procedure definitions cannot be nested. The syntax for using this example function is

```
{ a1,a2,a3 } = crosprod(u,v);
```

See **Procedures and Keywords**, Chapter 1, and **Libraries**, Chapter 1, for details.

Libraries

`declare`	Initializes variables at compile time.
`external`	External symbol definitions.
`lib`	Builds or updates a **GAUSS** library.
`library`	Sets up list of active libraries.

`call` allows functions to be called when return values are not needed. This is especially useful if a function produces printed output (**`dstat`**, **`ols`** for example) as well as return values.

Compiling

`compile`	Compiles and saves a program to a `.gcg` file.
`#include`	Inserts code from another file into a **GAUSS** program.
`loadp`	Loads compiled procedure.
`save`	Saves the compiled image of a procedure to disk.
`saveall`	Saves the contents of the current workspace to a file.
`use`	Loads previously compiled code.

GAUSS procedures and programs may be compiled to disk files. By then using this compiled code, the time necessary to compile programs from scratch is eliminated. Use `compile` to compile a command file. All procedures, matrices and strings referenced by that program will be compiled as well.

Stand-alone applications may be created by running compiled code under the **GAUSS** Run-Time Module. Contact Aptech Systems for more information on this product.

To save the compiled images of procedures that do not make any global references, use `save`. This will create an `.fcg` file. To load the compiled procedure into memory, use `loadp`. (This is not recommended because of the restriction on global references and the need to explicitly load the procedure in each program that references it. It is included here to maintain backward compatibility with previous versions.)

Miscellaneous Program Control

`gausset`	Resets the global control variables declared in `gauss.dec`.
`sysstate`	Gets or sets general system parameters.

37.11 OS Functions and File Management

`cdir`	Returns current directory.
`ChangeDir`	Changes directory in program.
`chdir`	Changes directory interactively.
`DeleteFile`	Deletes files.
`dlibrary`	Dynamically links and unlinks shared libraries.

`dllcall`	Calls functions located in dynamic libraries.
`dos`	Provides access to the operating system from within **GAUSS**.
`envget`	Gets an environment string.
`exec`	Executes an executable program file.
`execbg`	Provides access to the operating system from within **GAUSS**.
`fileinfo`	Takes a file specification, returns names and information of files that match.
`filesa`	Takes a file specification, returns names of files that match.
`getpath`	Returns an expanded filename including the drive and path.
`searchsourcepath`	Searches the source path and (if specified) the src subdirectory of the **GAUSS** installation directory for a specified file.
`shell`	Shells to OS.

37.12 Workspace Management

`clear`	Sets matrices equal to 0.

clearg	Sets global symbols to 0.
delete	Deletes specified global symbols.
hasimag	Examines matrix for nonzero imaginary part.
iscplx	Returns whether a matrix is real or complex.
maxbytes	Returns maximum memory to be read from a dataset at a time inside some **GAUSS** functions.
maxvec	Returns maximum allowed vector size.
new	Clears current workspace.
show	Displays global symbol table.
type	Returns type of argument (matrix or string).
typecv	Returns types of symbols (argument contains the names of the symbols to be checked).

When working with limited workspace, it is a good idea to `clear` large matrices that are no longer needed by your program.

37.13 Error Handling and Debugging

debug	Executes a program under the source level debugger.

`error`	Creates user-defined error code.
`errorlog`	Sends error message to screen and log file.
`#linesoff`	Omits line number and file name records from program.
`#lineson`	Includes line number and file name records in program.
`scalerr`	Tests for a scalar error code.
`trace`	Traces program execution for debugging.
`trap`	Controls trapping of program errors.
`trapchk`	Examines the trap flag.

To trace the execution of a program, use `trace`.

User-defined error codes may be generated using **`error`**.

37.14 String Handling

`chrs`	Converts ASCII values to a string.
`convertsatostr`	Converts a 1x1 string array to a string.
`convertstrtosa`	Converts a string to a 1x1 string array.
`cvtos`	Converts a character vector to a string.
`ftocv`	Converts an NxK matrix to a character matrix.

`ftos`	Converts a floating point scalar to a string.
`ftostrC`	Converts a matrix to a string array using a C language format specification.
`getf`	Loads ASCII or binary file into string.
`indsav`	Checks one string array against another and returns
`intrsectsa`	Returns the intersection of two string vectors, with duplicates removed.
`loads`	Loads a string file (`.fst` file).
`lower`	Converts a string to lowercase.
`ntos`	Converts a scalar or matrix to a string or string array. Simpler to use than **`ftos`**
`parse`	Parses a string, returning a character vector of tokens.
`putf`	Writes a string to disk file.
`stocv`	Converts a string to a character vector.
`stof`	Converts a string to floating point numbers.
`strcombine`	Converts an NxM string array to an Nx1 string vector by combining each element in a column separated by a user-defined delimiter string.

`strindx`	Finds starting location of one string in another string.
`strlen`	Returns length of a string.
`strput`	Lays a substring over a string.
`strrindx`	Finds starting location of one string in another string, searching from the end to the start of the string.
`strsect`	Extracts a substring of a string.
`strsplit`	Splits an Nx1 string vector into an NxK string array of the individual tokens.
`strsplitPad`	Splits an Nx1 string vector into an NxK string array of the individual tokens. Pads on the right with null strings.
`strtof`	Converts a string array to a numeric matrix.
`strtofcplx`	Converts a string array to a complex numeric matrix.
`strtriml`	Strips all whitespace characters from the left side of each element in a string array.
`strtrimr`	Strips all whitespace characters from the right side of each element in a string array.

`strtrunc`	Truncates all elements of a string array to not longer than the specified number of characters.
`strtruncl`	Truncates the left side of all elements of a string array by a user-specified number of characters.
`strtruncpad`	Truncates all elements of a string array to the specified number of characters, adding spaces on the end as needed to achieve the exact length.
`strtruncr`	Truncates the right side of all elements of a string array by a user-specified number of characters.
`token`	Extracts the leading token from a string.
`upper`	Changes a string to uppercase.
`vals`	Converts a string to ASCII values.
`varget`	Accesses the global variable named by a string.
`vargetl`	Accesses the local variable named by a string.
`varput`	Assigns a global variable named by a string.
`varputl`	Assigns a local variable named by a string.

strlen, **strindx**, **strrindx**, and **strsect** can be used together to parse strings.

Use **ftos** to print to a string.

To create a list of generic variable names (X1,X2,X3,X4,... for example), use **ftocv**.

37.15 Time and Date Functions

date	Returns current system date.
datestr	Formats date as "**mm/dd/yy**".
datestring	Formats date as "**mm/dd/yyyy**".
datestrymd	Formats date as "**yyyymmdd**".
dayinyr	Returns day number of a date.
dayofweek	Returns day of week.
dtdate	Creates a matrix in DT scalar format.
dtday	Creates a matrix in DT scalar format containing only the year, month, and day. Time of day information is zeroed out.
dttime	Creates a matrix in DT scalar format containing only the hour, minute, and second. The date information is zeroed out.
dttodtv	Converts DT scalar format to DTV vector format.

`dttostr`	Converts a matrix containing dates in DT scalar format to a string array.
`dttoutc`	Converts DT scalar format to UTC scalar format.
`dtvnormal`	Normalizes a date and time (DTV) vector.
`dtvtodt`	Converts DTV vector format to DT scalar format.
`dtvtoutc`	Converts DTV vector format to UTC scalar format.
`etdays`	Difference between two times in days.
`ethsec`	Difference between two times in hundredths of a second.
`etstr`	Converts elapsed time to string.
`hsec`	Returns elapsed time since midnight in hundredths of a second.
`strtodt`	Converts a string array of dates to a matrix in DT scalar format.
`time`	Returns current system time.
`timedt`	Returns system date and time in DT scalar format.
`timestr`	Formats time as "**`hh:mm:ss`**".
`timeutc`	Returns the number of seconds since January 1, 1970 Greenwich Mean

	Time.
todaydt	Returns system date in DT scalar format. The time returned is always midnight (00:00:00), the beginning of the returned day.
utctodt	Converts UTC scalar format to DT scalar format.
utctodtv	Converts UTC scalar format to DTV vector format.

Use **hsec** to time segments of code. For example,

```
    et = hsec;
    x = y*y;
    et = hsec - et;
```

will time the **GAUSS** multiplication operator.

37.16 Console I/O

con	Requests console input, creates matrix.
cons	Requests console input, creates string.
key	Gets the next key from the keyboard buffer. If buffer is empty, returns a 0.
keyav	Checks if keystroke is available.
keyw	Gets the next key from the keyboard buffer. If buffer is empty, waits for a

	key.
wait	Waits for a keystroke.
waitc	Flushes buffer, then waits for a keystroke.

key can be used to trap most keystrokes. For example, the following loop will trap the ALT-H key combination:

```
kk = 0;
do until kk == 1035;
   kk = key;
endo;
```

Other key combinations, function keys and cursor key movement can also be trapped. See **key**.

cons and **con** can be used to request information from the console. **keyw**, **wait**, and **waitc** will wait for a keystroke.

37.17 Output Functions

Text Output

cls	Clears the window.
comlog	Controls interactive command logging.
csrcol	Gets column position of cursor on window.
csrlin	Gets row position of cursor on window.

`ed`	Accesses an alternate editor.
`edit`	Edits a file with the **GAUSS** editor.
`format`	Defines format of matrix printing.
`formatcv`	Sets the character data format used by **`printfmt`**.
`formatnv`	Sets the numeric data format used by **`printfmt`**.
`header`	Prints a header for a report.
`headermt`	Prints a header for a report.
`locate`	Positions the cursor on the window.
`output`	Redirects `print` statements to auxiliary output.
`outwidth`	Sets line width of auxiliary output.
`print`	Prints to window.
`printdos`	Prints a string for special handling by the OS.
`printfm`	Prints matrices using a different format for each column.
`printfmt`	Prints character, numeric, or mixed matrix using a default format controlled by the functions **`formatcv`** and **`formatnv`**.
`satostrC`	Copies from one string array to

	another using a C language format specifier string for each element.
`screen [on	off]`	Directs/suppresses `print` statements to window.
`tab`	Positions the cursor on the current line.

The results of all printing can be sent to an output file using `output`. This file can then be printed or ported as an ASCII file to other software.

To produces boxes, etc. using characters from the extended ASCII set, use **`chrs`**.

DOS Compatibility Windows

`doswin`	Opens the DOS compatibility window with default settings.
`DOSWinCloseall`	Closes the DOS compatibility window.
`DOSWinOpen`	Opens the DOS compatibility window and gives it the specified title and attributes.

37.18 GAUSS Graphics

This section summarizes all procedures available within the **GAUSS** graphics system. A general usage description will be found in **GAUSS Graphics**, Chapter 1.

Graph Types

`plotBar`	Creates a bar plot.
`plotBox`	Creates a box plot.
`plotContour`	Creates a contour plot.
`plotHist`	Calculates and creates a frequency histogram plot.
`plotHistF`	Creates a histogram plot from a vector of frequencies.
`plotHistP`	Calculates and creates a percentage frequency histogram plot.
`plotLogLog`	Creates a 2-dimensional line plot with logarithmic scaling of the both the X and Y axes.
`plotLogX`	Creates a 2-dimensional line plot with logarithmic scaling of the X axis.
`plotLogY`	Creates a 2-dimensional line plot with logarithmic scaling of the Y axis.
`plotPolar`	Creates a polar plot.
`plotScatter`	Creates a 2-dimensional scatter plot.
`plotSurface`	Creates a 3-dimensional surface plot.
`plotTS`	Creates a graph of time series data.
`plotXY`	Creates a 2-dimensional line plot.

Adding Data to Existing Graphs

`plotAddBar`	Adds a bar or a set of bars to an existing 2-D graph.
`plotAddBox`	Adds a box plot to an existing 2-D graph.
`plotAddHist`	Adds a histogram to an existing 2-D graph.
`plotAddHistF`	Adds a frequency histogram to an existing 2-D graph.
`plotAddHistP`	Adds a percent frequency histogram to an existing 2-D graph.
`plotAddPolar`	Adds a graph using polar coordinates to an existing polar graph.
`plotAddScatter`	Adds a set of points to an existing 2-D graph.
`plotAddTS`	Adds a curve of time series data to an existing time series plot.
`plotAddXY`	Adds an XY plot to an existing 2-D graph.

Plot Control

`plotClearLayout`	Clears any previously set plot layouts.
`plotCustomLayout`	Plots a graph of user-specified size at a user-specified location.

`plotGetDefaults`	Gets default settings for graph types.
`plotLayout`	Divides a plot into a grid of subplots and assigns the cell location in which to draw the next created graph.
`plotOpenWindow`	Opens a new, empty graph whicow to be used by the next drawn graph.
`plotSave`	Saves the last created graph to a user specified file type.
`plotSetBar`	Sets the fill style and format of bars in a histogram or bar graph.
`plotSetBkdColor`	Sets background color of a graph.
`plotSetGrid`	Controls the settings for the background grid of a plot.
`plotSetLegend`	Adds a legend to a graph.
`plotSetLineColor`	Sets line colors for a graph.
`plotSetLineStyle`	Sets line styles for a graph.
`plotSetLineSymbol`	Sets line symbols displayed on the plotted points of a graph.
`plotSetLineThickness`	Sets line thickness for a graph.
`plotSetNewWindow`	Sets whether or not graph should be drawn in the same window or a new window.
`plotSetTitle`	Controls the settings for the title for a graph.

`plotSetXLabel`	Controls the settings for the X-axis label on a graph.
`plotSetXTicInterval`	Controls the interval between X-axis tic labels and also allows the user to specify the first tic to be labeled for 2-D time series graphs.
`plotSetXTicLabel`	Controls the formatting and angle of X-axis tic labels for 2-D time series graphs.
`plotSetYLabel`	Controls the settings for the Y-axis label on a graph.
`plotSetZLabel`	Controls the settings for the Z-axis label on a graph.

37.19 PQG Graphics

This section summarizes all procedures and global variables available within the PUBLICATION QUALITY GRAPHICS (PQG) System. A general usage description will be found in **PQG Graphics Colors**, Chapter 1. Note that PUBLICATION QUALITY GRAPHICS (PQG) graphic functions are included as legacy code and have been replaced with new plot functions.

Graph Types

`bar`	Generates bar graph.
`box`	Graphs data using the box graph percentile method.
`contour`	Graphs contour data.

`draw`	Supplies additional graphic elements to graphs.
`hist`	Computes and graphs frequency histogram.
`histf`	Graphs a histogram given a vector of frequency counts.
`histp`	Graphs a percent frequency histogram of a vector.
`loglog`	Graphs X,Y using logarithmic X and Y axes.
`logx`	Graphs X,Y using logarithmic X axis.
`logy`	Graphs X,Y using logarithmic Y axis.
`surface`	Graphs a 3-D surface.
`xy`	Graphs X,Y using Cartesian coordinate system.
`xyz`	Graphs X,Y,Z using 3-D Cartesian coordinate system.

Axes Control and Scaling

`_paxes`	Turns axes on or off.
`_pcross`	Controls where axes intersect.
`_pgrid`	Controls major and minor grid lines.
`_pticout`	Controls direction of tick marks on axes.

_pxpmax	Controls precision of numbers on X axis.
_pxsci	Controls use of scientific notation on X axis.
_pypmax	Controls precision of numbers on Y axis.
_pysci	Controls use of scientific notation on Y axis.
_pzpmax	Controls precision of numbers on Z axis.
_pzsci	Controls use of scientific notation on Z axis.
scale	Scales X,Y axes for 2-D plots.
scale3d	Scales X,Y, and Z axes for 3-D plots.
xtics	Scales X axis and controls tick marks.
ytics	Scales Y axis and controls tick marks.
ztics	Scales Z axis and controls tick marks.

Text, Labels, Titles, and Fonts

_paxht	Controls size of axes labels.
_pdate	Controls date string contents.
_plegctl	Sets location and size of plot legend.
_plegstr	Specifies legend text entries.

_pmsgctl	Controls message position.
_pmsgstr	Specifies message text.
_pnum	Controls axes numeric labels and orientation.
_pnumht	Controls size of axes numeric labels.
_ptitlht	Controls main title size.
asclabel	Defines character labels for tick marks.
fonts	Loads fonts for labels, titles, messages, and legend.
title	Specifies main title for graph.
xlabel	Specifies X axis label.
ylabel	Specifies Y axis label.
zlabel	Specifies Z axis label.

Main Curve Lines and Symbols

_pboxctl	Controls box plotter.
_pboxlim	Outputs percentile matrix from box plotter.
_pcolor	Controls line color for main curves.
_plctrl	Controls main curve and frequency of data symbols.
_pltype	Controls line style for main curves.

_plwidth	Controls line thickness for main curves.
_pstype	Controls symbol type for main curves.
_psymsiz	Controls symbol size for main curves.
_pzclr	Z level color control for **contour** and **surface**.

Extra Lines and Symbols

_parrow	Creates arrows.
_parrow3	Creates arrows for 3-D graphs.
_perrbar	Plots error bars.
_pline	Plots extra lines and circles.
_pline3d	Plots extra lines for 3-D graphs.
_psym	Plots extra symbols.
_psym3d	Plots extra symbols for 3-D graphs.

Graphic Panel, Page, and Plot Control

_pageshf	Shifts the graph for printer output.
_pagesiz	Controls size of graph for printer output.
_plotshf	Controls plot area position.
_plotsiz	Controls plot area size.

_protate	Rotates the graph 90 degrees.
axmargin	Controls axes margins and plot size.
begwind	Graphic panel initialization procedure.
endwind	Ends graphic panel manipulation; displays graphs.
getwind	Gets current graphic panel number.
loadwind	Loads a graphic panel configuration from a file.
makewind	Creates graphic panel with specified size and position.
margin	Controls graph margins.
nextwind	Sets to next available graphic panel number.
savewind	Saves graphic panel configuration to a file.
setwind	Sets to specified graphic panel number.
window	Creates tiled graphic panels of equal size.

axmargin is preferred to the older *_plotsiz* and *_plotshf* globals for establishing an absolute plot size and position.

Output Options

_pscreen	Controls graphics output to window.
_psilent	Controls final beep.
_ptek	Controls creation and name of `graphics.tkf` file.
_pzoom	Specifies zoom parameters.
graphprt	Generates print, conversion file.
pqgwin	Sets the graphics viewer mode.
setvwrmode	Sets the graphics viewer mode.
tkf2eps	Converts `.tkf` file to Encapsulated PostScript file.
tkf2ps	Converts `.tkf` file to PostScript file.

Miscellaneous

_pbox	Draws a border around graphic panel/window.
_pcrop	Controls cropping of graphics data outside axes area.
_pframe	Draws a frame around 2-D, 3-D plots.
_pmcolor	Controls colors to be used for axes, title, *x* and *y* labels, date, box, and background.
graphset	Resets all PQG globals to default

	values.
`rerun`	Displays most recently created graph.
`view`	Sets 3-D observer position in workbox units.
`viewxyz`	Sets 3-D observer position in plot coordinates.
`volume`	Sets length, width, and height ratios of 3-D workbox.

38 Command Reference

a

abs

Purpose

Returns the absolute value or complex modulus of *x*.

Format

```
y = abs(x);
```

Input

x	NxK matrix or sparse matrix or N-dimensional array.

Output

y	NxK matrix or sparse matrix or N-dimensional array containing absolute values of *x*.

Example

```
//Set rng seed for repeatable
//random numbers
rndseed 929212;

x = rndn(2,2);
y = abs(x);
```

The code above assigns the variables as follows:

```
x =  -0.23061709      0.054931120
      0.88863202     -0.82246522

y =   0.23061709      0.054931120
      0.88863202      0.82246522
```

In this example, a 2x2 matrix of Normal random numbers is generated and the absolute value of the matrix is computed.

acf

Purpose

Computes sample autocorrelations.

Format

rk = **acf**(*y*, *k*, *d*);

Input

y	Nx1 vector, data.
k	scalar, maximum number of autocorrelations to compute.
d	scalar, order of differencing.

Output

rk	Kx1 vector, sample autocorrelations.

Example

```
    x = { 20.80,
          18.58,
          23.39,
          20.47,
          21.78,
          19.56,
          19.58,
          18.91,
          20.08,
          21.88 };

    rk = acf(x,4,2);
    print rk;
```

The code above produces the following output:

```
     -0.74911771
      0.48360914
     -0.34229330
      0.17461180
```

Source

`tsutil.src`

aconcat

Purpose

Concatenates conformable matrices and arrays in a user-specified dimension.

Format

y = **aconcat**(*a*, *b*, *dim*);

Input

a	matrix or N-dimensional array.
b	matrix or K-dimensional array, conformable with *a*.
dim	scalar, dimension in which to concatenate.

Output

y	M-dimensional array, the result of the concatenation.

Remarks

`a` and `b` are conformable only if all of their dimensions except `dim` have the same sizes. If `a` or `b` is a matrix, then the size of dimension 1 is the number of columns in the matrix, and the size of dimension 2 is the number of rows in the matrix.

Example

```
//Create a 2x3x4 array with each element set to 0
a = arrayinit(2|3|4,0);

//Create a 3x4 matrix with each element set to 3
b = 3*ones(3,4);
y = aconcat(a,b,3);
```

y will be a 3x3x4 array, where [1,1,1] through [2,3,4] are zeros and [3,1,1] through [3,2,4] are threes.

```
//Create an additive sequence from 1-20 and 'reshape' it
//into a 4x5 matrix
a = reshape(seqa(1,1,20),4,5);

b = zeros(4,5);
y = aconcat(a,b,3);
```

y will be a 2x4x5 array, where [1,1,1] through [1,4,5] are sequential integers beginning with 1, and [2,1,1] through [2,4,5] are zeros.

```
//The pipe operator '|' causes vertical concatenation so
//that the statement 2|3|4 creates a 3x1 column vector
//equal to { 2, 3, 4 }
a = arrayinit(2|3|4,0);
b = seqa(1,1,24);
```

```
//'Reshape' the vector 'b' into a 2x3x4 dimensional array
b = areshape(b,2|3|4);
y = aconcat(a,b,5);
```

y will be a 2x1x2x3x4 array, where [1,1,1,1,1] through [1,1,2,3,4] are zeros, and [2,1,1,1,1] through [2,1,2,3,4] are sequential integers beginning with 1.

```
a = arrayinit(2|3|4,0);
b = seqa(1,1,6);
b = areshape(b,2|3|1);
y = aconcat(a,b,1);
```

y will be a 2x3x5 array, such that:

[1,1,1] through [1,3,5] =

```
0  0  0  0  1
0  0  0  0  2
0  0  0  0  3
```

[2,1,1] through [2,3,5] =

```
0  0  0  0  4
0  0  0  0  5
0  0  0  0  6
```

See Also

areshape

aeye

Purpose

Creates an N-dimensional array in which the planes described by the two trailing dimensions of the array are equal to the identity.

Format

a = **aeye**(*ord*);

Input

ord	Nx1 vector of orders, the sizes of the dimensions of *a*.

Output

a	N-dimensional array, containing 2-dimensional identity arrays.

Remarks

If *ord* contains numbers that are not integers, they will be truncated to integers.

The planes described by the two trailing dimensions of *a* will contain 1's down the diagonal and 0's everywhere else.

Example

```
v = { 2, 3, 3 };
```

```
    a = aeye(v);
```

a will be a 2x3x3 array, such that:

[1,1,1] through [1,3,3] =

```
    1 0 0
    0 1 0
    0 0 1
```

[2,1,1] through [2,3,3] =

```
    1 0 0
    0 1 0
    0 0 1
```

See Also

eye

amax

Purpose

Moves across one dimension of an N-dimensional array and finds the largest element.

Format

y = **amax**(*x*, *dim*);

Input

x	N-dimensional array.
dim	scalar, number of dimension across which to find the maximum value.

Output

y	N-dimensional array.

Remarks

The output *y*, will have the same sizes of dimensions as *x*, except that the dimension indicated by *dim* will be collapsed to 1.

Example

```
rndseed 9823432;

//Create random normal numbers with a standard deviation
//of 10 and round them to the nearest integer
x = round(10*rndn(24,1));

//Reshape them from a 24x1 vector into 2x3x4 array
x = areshape(x,2|3|4);

// Calculate the max across the second dimension
dim = 2;
y = amax(x,dim);
```

After this calculation:

x[1,1,1] through *x*[1,3,4] =

```
    -14.000000      4.0000000       6.0000000      -4.0000000
     1.0000000      8.0000000       10.000000       9.0000000
    -3.0000000      12.000000       5.0000000      -26.000000
```

x[2,1,1] through *x*[2,3,4] =

```
     4.0000000      6.0000000       4.0000000       2.0000000
     1.0000000      16.000000       9.0000000      -4.0000000
    -4.0000000     -8.0000000      -10.000000       8.0000000
```

y[1,1,1] through *y*[1,1,4] =

```
    1.0000000       12.000000       10.000000       9.0000000
```

y[2,1,1] through *y*[2,1,4] =

```
    4.0000000       16.000000       9.0000000       8.0000000
```

Use the same *x* array and calculate the max across dimension 1:

```
    y2 = amax(x,1);
```

After this calculation, *x* remains the same, but *y2* is:

y2[1,1,1] through *y2*[1,3,1] =

```
    6.0000000
    10.000000
    12.000000
```

y2[2,1,1] through *y2*[2,3,1] =

```
    6.0000000
    16.000000
    8.0000000
```

See Also

amin, maxc

amean

Purpose

Computes the mean across one dimension of an N-dimensional array.

Format

y = **amean**(*x*, *dim*);

Input

x	N-dimensional array.
dim	scalar, number of dimension to compute the mean across.

Output

y	[N-1]-dimensional array.

Remarks

The output *y*, will have the same sizes of dimensions as *x*, except that the dimension indicated by `dim` will be collapsed to 1.

Example

```
//Create an additive sequence from 1-24
x = seqa(1,1,24);

//'Reshape' this 24x1 vector into a 2x3x4 dimensional array
x = areshape(x,2|3|4);

y = amean(x,3);
```

x is a 2x3x4 array, such that:

[1,1,1] through [1,3,4] =

```
1.0000000      2.0000000      3.0000000      4.0000000
5.0000000      6.0000000      7.0000000      8.0000000
9.0000000      10.000000      11.000000      12.000000
```

[2,1,1] through [2,3,4] =

```
13.000000      14.000000      15.000000      16.000000
17.000000      18.000000      19.000000      20.000000
21.000000      22.000000      23.000000      24.000000
```

y will be a 1x3x4 array, such that:

[1,1,1] through [1,3,4] =

```
    7.0000000       8.0000000       9.0000000       10.000000
    11.000000       12.000000       13.000000       14.000000
    15.000000       16.000000       17.000000       18.000000
```

```
    y = amean(x,1);
```

Using the same array *x* as the above example, this example computes the mean across the first dimension. *y* will be a 2x3x1 array, such that:

[1,1,1] through [1,3,1] =

```
    2.5000000
    6.5000000
    10.500000
```

[2,1,1] through [2,3,1] =

```
    14.500000
    18.500000
    22.500000
```

See Also

asum

AmericanBinomCall

Purpose

Prices American call options using binomial method.

Format

```
c = AmericanBinomCall(S0, K, r, div, tau, sigma, N);
```

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

Output

c	Mx1 vector, call premiums.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach," *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
    S0 = 718.46;
    K = { 720, 725, 730 };
    r = .0498;
    sigma = .2493;

    t0 = dtday(2001, 1, 30);
    t1 = dtday(2001, 2, 16);
    tau = elapsedTradingDays(t0,t1) /
        annualTradingDays(2001);

    c = AmericanBinomCall(S0,K,r,0,tau,sigma,60);
    print c;
```

produces the output:

```
     17.344044
     15.058486
     12.817427
```

Source

finprocs.src

AmericanBinomCall_Greeks

Purpose

Computes Delta, Gamma, Theta, Vega, and Rho for American call options using binomial method.

Format

```
{ d, g, t, v, rh } = AmericanBinomCall_Greeks(S0, K, r,
div, tau, sigma, N);
```

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

Global Input

_fin_thetaType	scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.
_fin_epsilon	scalar, finite difference stepsize. Default = 1e-8.

Output

d	Mx1 vector, delta.
g	Mx1 vector, gamma.

t	Mx1 vector, theta.
v	Mx1 vector, vega.
rh	Mx1 vector, rho.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach," *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
S0 = 305;
K = 300;
r = .08;
sigma = .25;
tau = .33;
div = 0;

print AmericanBinomCall_Greeks(S0,K,r,0,tau,sigma,30);
```

produces:

```
  0.70631204
  0.00076381912
-17.400851
 68.703851
 76.691829
```

Source

`finprocs.src`

See Also

AmericanBinomCall_ImpVol, AmericanBinomCall, AmericanBinomPut_Greeks, AmericanBSCall_Greeks

AmericanBinomCall_ImpVol

Purpose

Computes implied volatilities for American call options using binomial method.

Format

sigma = **AmericanBinomCall_ImpVol**(*c*, *S0*, *K*, *r*, *div*, *tau*, *N*);

Input

c	Mx1 vector, call premiums
S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.

tau	scalar, elapsed time to exercise in annualized days of trading.
N	number of time segments.

Output

sigma	Mx1 vector, volatility.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach," *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
c = { 13.70, 11.90, 9.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
div = 0;

t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2001);

sigma = AmericanBinomCall_ImpVol(c,S0,K,r,0,tau,30);
print sigma;
```

produces:

```
    0.19629517
    0.16991943
    0.12874756
```

Source

`finprocs.src`

AmericanBinomPut

Purpose

Prices American put options using binomial method.

Format

c = **AmericanBinomPut**(*S0*, *K*, *r*, *div*, *tau*, *sigma*, *N*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

Output

c	Mx1 vector, put premiums.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach," *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;

t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2001);

c = AmericanBinomPut(S0,K,r,0,tau,sigma,60);
print c;
```

produces:

```
16.986117
19.729923
22.548538
```

Source

`finprocs.src`

AmericanBinomPut_Greeks

Purpose

Computes Delta, Gamma, Theta, Vega, and Rho for American put options using binomial method.

Format

{ *d*, *g*, *t*, *v*, *rh* } = **AmericanBinomPut_Greeks**(*S0*, *K*, *r*, *div*, *tau*, *sigma*, *N*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

Global Input

_fin_thetaType	scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.
_fin_epsilon	scalar, finite difference stepsize. Default = 1e-8.

Output

d	Mx1 vector, delta.
g	Mx1 vector, gamma.
t	Mx1 vector, theta.
v	Mx1 vector, vega.
rh	Mx1 vector, rho.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach," *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
     S0 = 305;
     K = 300;
     r = .08;
     div = 0;
```

```
    sigma = .25;
    tau = .33;

    print AmericanBinomPut_Greeks(S0,K,r,0,tau,sigma,60);
```

produces

```
     -0.38324908
      0.00076381912
      8.1336630
     68.337294
    -27.585043
```

Source

`finprocs.src`

See Also

AmericanBinomPut_ImpVol, AmericanBinomPut, AmericanBinomCall_Greeks, AmericanBSPut_Greeks

AmericanBinomPut_ImpVol

Purpose

Computes implied volatilities for American put options using binomial method.

Format

sigma = **AmericanBinomPut_ImpVol**(*c*, *S0*, *K*, *r*, *div*, *tau*, *N*);

Input

c	Mx1 vector, put premiums
S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
N	number of time segments.

Output

sigma	Mx1 vector, volatility.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach," *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
    p = { 14.60, 17.10, 20.10 };
    S0 = 718.46;
    K = { 720, 725, 730 };
    r = .0498;
    div = 0;

    t0 = dtday(2001, 1, 30);
    t1 = dtday(2001, 2, 16);
    tau = elapsedTradingDays(t0,t1) /
        annualTradingDays(2001);

    sigma = AmericanBinomPut_ImpVol(p,S0,K,r,0,tau,30);
    print sigma;
```

produces:

```
    0.12466064
    0.16583252
    0.21203735
```

Source

`finprocs.src`

AmericanBSCall

Purpose

Prices American call options using Black, Scholes, and Merton method.

Format

c = **AmericanBSCall**(*S0*, *K*, *r*, *div*, *tau*, *sigma*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

Output

c	Mx1 vector, call premiums.

Example

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;

t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2001);

c = AmericanBSCall(S0,K,r,0,tau,sigma);
print c;
```

produces:

```
    32.005720
    31.083232
    30.367548
```

Source

`finprocs.src`

AmericanBSCall_Greeks

Purpose

Computes Delta, Gamma, Theta, Vega, and Rho for American call options using Black, Scholes, and Merton method.

Format

{ *d*, *g*, *t*, *v*, *rh* } = **AmericanBSCall_Greeks**(*S0*, *K*, *r*, *div*, *tau*, *sigma*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.

tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

Global Input

_fin_thetaType	scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.
_fin_epsilon	scalar, finite difference stepsize. Default = 1e-8.

Output

d	Mx1 vector, delta.
g	Mx1 vector, gamma.
t	Mx1 vector, theta.
v	Mx1 vector, vega.
rh	Mx1 vector, rho.

Example

```
S0 = 305;
K = 300;
r = .08;
sigma = .25;
tau = .33;
print AmericanBSCall_Greeks(S0,K,r,0,tau,sigma);
```

produces:

```
      0.40034039
      0.016804021
     -55.731079
      115.36906
      46.374528
```

Source

finprocs.src

See Also

AmericanBSCall_ImpVol, AmericanBSCall, AmericanBSPut_Greeks, AmericanBinomCall_Greeks

AmericanBSCall_ImpVol

Purpose

Computes implied volatilities for American call options using Black, Scholes, and Merton method.

Format

sigma = **AmericanBSCall_ImpVol**(*c*, *S0*, *K*, *r*, *div*, *tau*);

Input

c	Mx1 vector, call premiums.
S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.

Output

sigma	Mx1 vector, volatility.

Example

```
c = { 13.70, 11.90, 9.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;

t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2001);

sigma = AmericanBSCall_ImpVol(c,S0,K,r,0,tau);
print sigma;
```

produces:

```
    0.10259888
    0.088370361
    0.066270752
```

Source

`finprocs.src`

AmericanBSPut

Purpose

Prices American put options using Black, Scholes, and Merton method.

Format

c = **AmericanBSPut**(*S0*, *K*, *r*, *div*, *tau*, *sigma*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

Output

c	Mx1 vector, put premiums.

Example

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;

t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2001);

c = AmericanBSPut(S0,K,r,0,tau,sigma);
print c;
```

produces:

```
16.870783
19.536842
22.435487
```

Source

finprocs.src

AmericanBSPut_Greeks

Purpose

Computes Delta, Gamma, Theta, Vega, and Rho for American put options using Black, Scholes, and Merton method.

Format

```
{ d, g, t, v, rh } = AmericanBSPut_Greeks(S0, K, r, div,
tau, sigma);
```

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

Global Input

_fin_thetaType	scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.
_fin_epsilon	scalar, finite difference stepsize. Default = 1e-8.

Output

d	Mx1 vector, delta.
g	Mx1 vector, gamma.
t	Mx1 vector, theta.
v	Mx1 vector, vega.
rh	Mx1 vector, rho.

Example

```
S0 = 305;
K = 300;
r = .08;
sigma = .25;
tau = .33;

print AmericanBSPut_Greeks(S0,K,r,0,tau,sigma);
```

produces:

```
-0.33296721
 0.0091658294
-17.556118
 77.614237
-40.575963
```

Source

finprocs.src

See Also

AmericanBSCall_ImpVol, AmericanBSCall_Greeks, AmericanBSPut_ImpVol

AmericanBSPut_ImpVol

Purpose

Computes implied volatilities for American put options using Black, Scholes, and Merton method.

Format

sigma = **AmericanBSPut_ImpVol**(*c*, *S0*, *K*, *r*, *div*, *tau*);

Input

c	Mx1 vector, put premiums.
S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.

Output

sigma	Mx1 vector, volatility.

Example

```
p = { 14.60, 17.10, 20.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;

t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2001);

sigma = AmericanBSPut_ImpVol(p,S0,K,r,0,tau);
print sigma;
```

produces:

```
0.12753662
0.16780029
0.21396729
```

Source

`finprocs.src`

amin

Purpose

Moves across one dimension of an N-dimensional array and finds the smallest element.

Format

y = **amin**(*x*, *dim*);

Input

x	N-dimensional array.
dim	scalar, number of dimension across which to find the minimum value.

Output

y	N-dimensional array.

Remarks

The output *y*, will have the same sizes of dimensions as *x*, except that the dimension indicated by *dim* will be collapsed to 1.

Example

```
//Setting the rng seed allows for repeatable
//random numbers
rndseed 8237348;

//Create a 24x1 vector of random normal numbers
//with a standard deviation of 10 and then round
//to the nearest integer value
x = round(10*rndn(24,1));

//Reshape the 24x1 vector into a 2x3x4 dimensional array
```

```
//NOTE: The pipe operator '|' is for vertical concatenation
x = areshape(x,2|3|4);

dim = 2;
y = amin(x,dim);
```

x is a 2x3x4 array, such that:

[1,1,1] through [1,3,4] =

```
 1.0000000      -11.000000       9.0000000      -8.0000000
-2.0000000      -10.000000      -6.0000000      -5.0000000
-5.0000000       17.000000       9.0000000      -2.0000000
```

[2,1,1] through [2,3,4] =

```
-4.0000000      -2.0000000       7.0000000      -2.0000000
 4.0000000       13.000000      -16.000000       11.000000
 2.0000000      -1.0000000       12.000000      -16.000000
```

y will be a 2x1x4 array, such that:

[1,1,1] through [1,1,4] =

```
-5.0000000      -11.000000      -6.0000000      -8.0000000
```

[2,1,1] through [2,1,4] =

```
-4.0000000      -2.0000000      -16.000000      -16.000000
```

```
y = amin(x,1);
```

Using the same array *x* as the above example, this example finds the minimum value across the first dimension.

y will be a 2x3x1 array, such that:

[1,1,1] through [1,3,1] =

```
    -11.000000
    -10.000000
    -5.0000000
```

[2,1,1] through [2,3,1] =

```
    -4.0000000
    -16.000000
    -16.000000
```

See Also

amax, minc

amult

Purpose

Performs matrix multiplication on the planes described by the two trailing dimensions of N-dimensional arrays.

Format

y = **amult**(*a*, *b*);

Input

a	N-dimensional array.
b	N-dimensional array.

Output

y	N-dimensional array, containing the product of the matrix multiplication of the planes described by the two trailing dimensions of *a* and *b*.

Remarks

All leading dimensions must be strictly conformable, and the two trailing dimensions of each array must be matrix-product conformable.

Example

```
//Create an additive sequence from 1-12 and reshape it into
//a 2x3x2 dimensional array
a = areshape(seqa(1,1,12),2|3|2);

b = areshape(seqa(1,1,16),2|2|4);

//Multiply the two 3x2 matrices in 'a' by the corresponding
//2x4 matrices in 'b'
y = amult(a,b);
```

a is a 2x3x2 array, such that:

[1,1,1] through [1,3,2] =

```
    1.0000000        2.0000000
    3.0000000        4.0000000
    5.0000000        6.0000000
```

[2,1,1] through [2,3,2] =

```
    7.0000000        8.0000000
    9.0000000        10.000000
    11.000000        12.000000
```

b is a 2x2x4 array, such that:

[1,1,1] through [1,2,4] =

```
    1.0000000       2.0000000       3.0000000       4.0000000
    5.0000000       6.0000000       7.0000000       8.0000000
```

[2,1,1] through [2,2,4] =

```
    9.0000000       10.000000       11.000000       12.000000
    13.000000       14.000000       15.000000       16.000000
```

y will be a 2x3x4 array, such that:

[1,1,1] through [1,3,4] =

```
    11.000000       14.000000       17.000000       20.000000
    23.000000       30.000000       37.000000       44.000000
    35.000000       46.000000       57.000000       68.000000
```

[2,1,1] through [2,3,4] =

```
    167.00000       182.00000       197.00000       212.00000
    211.00000       230.00000       249.00000       268.00000
    255.00000       278.00000       301.00000       324.00000
```

annualTradingDays

Purpose

Computes number of trading days in a given year.

Format

n = **annualTradingDays**(*a*);

Input

a	scalar, year.

Output

n	number of trading days in year.

Remarks

A trading day is a weekday that is not a holiday as defined by the New York Stock Exchange from 1888 through 2012. Holidays are defined in `holidays.asc`. You may edit that file to modify or add holidays.

Source

`finutils.src`

Globals

_fin_annualTradingDays, *_fin_holidays*

See Also

eTD, gNTD, gPTD, gNWD, gPWD

arccos

Purpose

Computes the inverse cosine.

Format

y = **arccos**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing the angle in

radians whose cosine is x.

Remarks

If x is complex or has any elements whose absolute value is greater than 1, complex results are returned.

Example

```
//Format print statements to show 3 digits
//after the decimal point
format /rd 6,3;

x = { -1, -0.5, 0, 0.5, 1 };
y = arccos(x);

print "x = " x;
print "y = " y;
```

The code above, produces the following output:

```
x =
    -1.000
    -0.500
     0.000
     0.500
     1.000
y =
     3.142
     2.094
     1.571
    1.047
    0.000
```

Source

`trig.src`

arcsin

Purpose

Computes the inverse sine.

Format

y = **`arcsin`**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array, the angle in radians whose sine is *x*.

Remarks

If *x* is complex or has any elements whose absolute value is greater than 1, complex results are returned.

Example

```
    //Set 'x' to be the sequence -1, -0.5, 0, 0.5, 1
    x = seqa(-1, 0.5, 5);
    y = arcsin(x);
```

Assigns *y* to be equal to:

```
    -1.5707963
    -0.52359878
     0.00000000
     0.52359878
     1.5707963
```

Source

`trig.src`

areshape

Purpose

Reshapes a scalar, matrix, or array into an array of user-specified size.

Format

y = **areshape**(*x*, *o*);

Input

x	scalar, matrix, or N-dimensional array.
o	Mx1 vector of orders, the sizes of the dimensions of the new array.

Output

y	M-dimensional array, created from data in *x*.

Remarks

If there are more elements in *x* than in *y*, the remaining elements are discarded. If there are not enough elements in *x* to fill *y*, then when **areshape** runs out of elements, it goes back to the first element of *x* and starts getting additional elements from there.

Example

```
    x = 3;
    orders = { 2,3,4 };
    y = areshape(x,orders);
```

y will be a 2x3x4 array of threes.

```
    x = reshape(seqa(1,1,90),30,3);
    orders = { 2,3,4,5 };
    y = areshape(x,orders);
```

y will be a 2x3x4x5 array. Since *y* contains 120 elements and *x* contains only 90, the first 90 elements of *y* will be set to the sequence of integers from 1 to 90 that are contained in *x*, and the last 30 elements of *y* will be set to the sequence of integers from 1 to 30 contained in the first 30 elements of *x*.

```
x = reshape(seqa(1,1,60),20,3);
orders = { 3,2,4 };
y = areshape(x,orders);
```

y will be a 3x2x4 array. Since *y* contains 24 elements, and *x* contains 60, the elements of *y* will be set to the sequence of integers from 1 to 24 contained in the first 24 elements of *x*.

See Also

aconcat

arrayalloc

Purpose

Creates an N-dimensional array with unspecified contents.

Format

y = **arrayalloc**(*o*, *cf*);

Input

o	Nx1 vector of orders, the sizes of the dimensions of the array.
cf	scalar, 0 to allocate real array, or 1 to allocate complex array.

Output

y	N-dimensional array.

Remarks

The contents are unspecified. This function is used to allocate an array that will be written to in sections using `setarray`.

Example

```
orders = { 2,3,4 };
y = arrayalloc(orders, 1);
```

y will be a complex 2x3x4 array with unspecified contents.

```
//Tell GAUSS to replace all instances of 'REAL' with a 0
#define REAL 0
orders = { 7, 5, 3 };

//Create a real 7x5x3 dimensional array; before GAUSS
//interprets this statement it will replace 'REAL' with
//a scalar 0
y = arrayalloc(orders, REAL);
```

See Also

arrayinit, setarray

arrayindex

Purpose

Converts a scalar vector index to a vector of indices for an N-dimensional array.

Format

i = **arrayindex**(*si*, *o*);

Input

si	scalar, index into vector or 1-dimensional array.
o	Nx1 vector of orders of an N-dimensional array.

Output

i	Nx1 vector of indices, index of corresponding element in N-dimensional array.

Remarks

This function and its opposite, **singleindex**, allow you to easily convert between an N-dimensional index and its corresponding location in a 1-dimensional object of the same size.

Example

```
//Set the rng seed for repeatable random numbers
rndseed 982348;

orders = { 2,3,4,5 };

//Create 120x1 vector of uniform random numbers
//(2*3*4*5 = 120)
v = rndu(prodc(orders),1);

//Reshape the 120x1 random vector into a 2x3x4x5
//dimensional array
a = areshape(v,orders);

vi = 50;
ai = arrayindex(vi,orders);

print "vi = " vi;
print "ai = " ai;
print "v[vi] = " v[vi];
//The double semi-colon below suppresses the
//new-line allowing the string and the data to be
//printed on the same line
print "getarray(a, ai) = ";; getarray(a,ai);
```

The code above, produces the following output:

```
vi =   50.000
ai =
   1.000
   3.000
   2.000
   5.000
```

```
v[vi] =    0.047
getarray(a, ai) =    0.047
```

This example allocates a vector of random numbers and creates a 4-dimensional array using the same data. The 50th element of the vector `v` corresponds to the element of array `a` that is indexed with `ai`.

See Also

singleindex

arrayinit

Purpose

Creates an N-dimensional array with a specified fiSll value.

Format

y = **`arrayinit`**(*o*, *v*);

Input

o	Nx1 vector of orders, the sizes of the dimensions of the array.
v	scalar, value to initialize. If *v* is complex the result will be complex.

Output

y	N-dimensional array with each element equal to the value of *v*.

Example

```
    val = 3.14;
    orders = { 2, 100, 9 };
    y = arrayinit(orders, val);
```

y will be a 2x100x9 array with each element equal to 3.14.

See Also

arrayalloc

arraytomat

Purpose

Converts an array to type matrix.

Format

y = **arraytomat**(*a*);

Input

a	N-dimensional array.

Output

y	KxL or 1xL matrix or scalar, where L is the size of the fastest moving dimension of the array and K is the size of the second fastest moving dimension.

Remarks

arraytomat will take an array of 1 or 2 dimensions or an N-dimensional array, in which the N-2 slowest moving dimensions each have a size of 1.

Example

```
//Create 25x1 vector containing the sequence 0.5, 1,
//1.5...12.5
x = seqa(0.5, 0.5, 25);

//Reshape into a 1x6x4 array, discarding the 25th element
//of 'x'
a = areshape(x, 1|6|4);

//Set 'y' to be a 6x4 variable of type matrix, with the
//same contents as 'a'
y = arraytomat(a);
```

The code above sets *y* equal to:

```
0.5    1.0    1.5    2.0
2.5    3.0    3.5    4.0
4.5    5.0    5.5    6.0
6.5    7.0    7.5    8.0
```

```
     8.5     9.0     9.5    10.0
    10.5    11.0    11.5    12.0
```

See Also

mattoarray

asciiload

Purpose

Loads data from a delimited ASCII text file into an Nx1 vector.

Format

y = **asciiload**(*filename*);

Input

filename	string, name of data file.

Output

y	Nx1 vector.

Remarks

The file extension must be included in the file name.

Numbers in ASCII files must be delimited with spaces, commas, tabs, or newlines.

This command loads as many elements as possible from the file into an Nx1 vector. This allows you to verify if the load was successful by calling **rows**(*y*) after **asciiload** to see how many elements were actually loaded. You may then **reshape** the Nx1 vector to the desired form. You could, for instance, put the number of rows and columns of the matrix right in the file as the first and second elements and **reshape** the remainder of the vector to the desired form using those values.

Example

To load the file `myfile.asc`, containing the following data:

```
  2.805  16.568
 -4.871   3.399
 17.361 -12.725
```

you may use any of the following commands:

```
//This statement assumes 'myfile.asc' is in the current
//working directory
y = asciiload("myfile.asc");
```

```
//This code assumes that 'myfile.asc' is
//located in the C:\gauss13 directory
//Note the double backslashes for path separators
fpath = "C:\\gauss13\\myfile.asc";
y = asciiload(fpath);
```

```
path = "C:\\gauss13\\";
fname = "myfile.asc";
//The '$+' operator adds two strings together into one
//string
y = asciiload(path$+fname);
```

All of the above commands will set *y* to be equal to:

```
     2.805
    16.568
    -4.871
     3.399
    17.361
   -12.725
```

See Also

load, dataload

asclabel

Purpose

To set up character labels for the X and Y axes. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

asclabel(*x1*, *y1*);

Input

x1	string or Nx1 character vector, labels for the tick marks on the X axis. Set to 0 if no character labels for this axis are desired.

yl	string or Mx1 character vector, labels for the tick marks on the Y axis. Set to 0 if no character labels for this axis are desired.

Example

This illustrates how to label the X axis with the months of the year:

```
    let lab = JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC;
    asclabel(lab,0);
```

This will also work:

```
    lab = "JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DE
    C";
    asclabel(lab,0);
```

If the string format is used, then escape characters may be embedded in the labels. For example, the following produces character labels that are multiples of λ. The font Simgrma must be previously loaded in a **fonts** command.

```
    fonts("simplex simgrma");
    lab = "\2010.25\202l \2010.5\202l \2010.75\202l l";
    asclabel(lab,0);
```

Here, the "**\202l**" produces the "λ" symbol from Simgrma.

Source

pgraph.src

See Also

xtics, ytics, scale, scale3d, fonts

astd

Purpose

Computes the standard deviation of the elements across one dimension of an N-dimensional array.

Format

```
y = astd(x, dim);
```

Input

x	N-dimensional array.
dim	scalar, number of dimension to sum across.

Output

y	N-dimensional array, standard deviation across specified dimension of *x*.

Remarks

The output *y*, will have the same sizes of dimensions as *x*, except that the dimension indicated by *dim* will be collapsed to 1.

This function essentially computes:

```
sqrt(1/(N-1)*sumc((x-meanc(x)')^2))
```

Thus, the divisor is N-1 rather than N, where N is the number of elements being summed. See **astds** for the alternate definition.

Example

```
    //Create a 1e6x1 vector of random normal numbers with a
    //standard deviation of 25 and reshape it into a
    //2e5x3x2 array
    a = areshape(25*rndn(2e6,1),2e5|3|2);
    y = astd(a,3);
```

The code above should produce a 3x2 matrix with all elements close to 25 similar to what we see below. Since the example uses random numbers, your answer may vary slightly.

```
    24.997   25.030
    25.012   24.986
    24.978   25.000
```

See Also

astds, stdc

astds

Purpose

Computes the 'sample' standard deviation of the elements across one dimension of an N-dimensional array.

Format

```
y = astds(x, dim);
```

Input

x	N-dimensional array.
dim	scalar, number of dimension to sum across.

Output

y	N-dimensional array, standard deviation across specified dimension of *x*.

Remarks

The output *y*, will have the same sizes of dimensions as *x*, except that the dimension indicated by *dim* will be collapsed to 1.

This function essentially computes:

```
sqrt(1/(N)*sumc((x-meanc(x)')^2))
```

Thus, the divisor is N rather than N-1, where N is the number of elements being summed. See **astd** for the alternate definition.

Example

```
a = areshape(25*rndn(16,1),4|2|2);
y = astds(a,3);
```

```
print "a = " a;
print "y = " y;
```

The code above produces the following output (due to the use of random data in this example your answers will be different):

```
a =

Plane [1,.,.]

  12.538  -56.786
 -40.283  -58.287

Plane [2,.,.]

   4.047   -0.325
  17.617   -9.248

Plane [3,.,.]

  17.908   40.048
   8.916  -37.247

Plane [4,.,.]

  -0.977   16.058
 -38.189    0.984

 y =

 Plane [1,.,.]
```

```
   7.321    35.659
  26.441    23.333
```

In this example, 16 standard Normal random variables are generated. They are multiplied by 25 and **areshape**'d into a 4x2x2 array, and the standard deviation is computed across the third dimension of the array.

See Also

astd, stdsc

asum

Purpose

Computes the sum across one dimension of an N-dimensional array.

Format

y = **asum**(*x*, *dim*);

Input

x	N-dimensional array.
dim	scalar, number of dimension to sum across.

Output

y	N-dimensional array.

Remarks

The output *y*, will have the same sizes of dimensions as *x*, except that the dimension indicated by *dim* will be collapsed to 1.

Example

```
x = seqa(1,1,24);

dims = { 2, 3, 4 };
x = areshape(x,dims);

y = asum(x,3);
```

x is a 2x3x4 array, such that:

```
Plane [1,.,.]

   1.000    2.000    3.000    4.000
   5.000    6.000    7.000    8.000
   9.000   10.000   11.000   12.000

Plane [2,.,.]

  13.000   14.000   15.000   16.000
  17.000   18.000   19.000   20.000
  21.000   22.000   23.000   24.000
```

and *y* is equal to:

```
Plane [1,.,.]

  14.000   16.000   18.000   20.000
```

```
       22.000    24.000    26.000    28.000
       30.000    32.000    34.000    36.000
```

```
     y = asum(x,1);
```

Using the same array *x* as the above example, this example computes the sum across the first dimension. *y* will be a 2x3x1 array, such that:

```
     Plane [1,.,.]

       10.000
       26.000
       42.000

     Plane [2,.,.]

       58.000
       74.000
       90.000
```

See Also

amean

atan

Purpose

Returns the arctangent of its argument.

Format

y = **atan**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing the arctangents of *x* in radians.

Remarks

y will be the same size as *x*, containing the arctangents of the corresponding elements of *x*.

For real *x*, the arctangent of *x* is the angle whose tangent is *x*. The result is a value in radians in the range $-\pi/2$ to $+\pi/2$. To convert radians to degrees, multiply by $180/\pi$.

For complex *x*, the arctangent is defined everywhere except *i* and −*i*. If *x* is complex, *y* will be complex.

Example

```
    //Create a sequence with 5 elements starting at -pi and
    //increasing by pi/2
    x = seqa(-pi, pi/2, 5)
    y = atan(x);
```

After the code above:

```
       -3.142       -1.263
       -1.571       -1.004
```

```
    x = 0.000   y =  0.000
        1.571        1.004
        3.142        1.263
```

See Also

atan2, sin, cos, pi, tan

atan2

Purpose

Computes an angle from an *x*, *y* coordinate.

Format

z = **atan2**(*y*, *x*);

Input

y	NxK matrix or P-dimensional array where the last two dimensions are NxK, the *y* coordinate.
x	LxM matrix or P-dimensional array where the last two dimensions are LxM, ExE conformable with *y*, the *x* coordinate.

Output

z	max(N,L) by max(K,M) matrix or P-dimensional array

where the last two dimensions are max(N,L) by max(K,M).

Remarks

Given a point x, y in a Cartesian coordinate system, **atan2** will give the correct angle with respect to the positive X axis. The answer will be in radians from $-\pi$ to $+\pi$.

To convert radians to degrees, multiply by $180/\pi$.

atan2 operates only on the real component of x, even if x is complex.

Example

```
//Create the sequence -π, -π/2, 0, π/2, π
x = seqa(-pi, pi/2, 5);
y = 1;

zpol = atan2(y,x);
zdeg = zpol*(180/pi);
```

After the code above:

```
   -3.142         2.833         162.343
   -1.571         2.575         147.518
x = 0.000  zpol = 1.571  zdeg =  90.000
    1.571         0.567          32.482
    3.142         0.308          17.657
```

See Also

atan, sin, cos, pi, tan, arcsin, arccos

atranspose

Purpose

Transposes an N-dimensional array.

Format

y = **atranspose**(*x*, *nd*);

Input

x	N-dimensional array.
nd	Nx1 vector of dimension indices, the new order of dimensions.

Output

y	N-dimensional array, transposed according to *nd*.

Remarks

The vector of dimension indices must be a unique vector of integers, 1-N, where 1 corresponds to the first element of the vector of orders.

Example

```
x = seqa(1,1,24);
x = areshape(x,2|3|4);
```

```
    nd = { 2,1,3 };
    y = atranspose(x,nd);
```

This example transposes the dimensions of *x* that correspond to the first and second elements of the vector of orders. *x* is a 2x3x4 array, such that:

```
    Plane [1,.,.]

       1.000    2.000    3.000    4.000
       5.000    6.000    7.000    8.000
       9.000   10.000   11.000   12.000

    Plane [2,.,.]

      13.000   14.000   15.000   16.000
      17.000   18.000   19.000   20.000
      21.000   22.000   23.000   24.000
```

y is a 3x2x4 array, such that:

```
    Plane [1,.,.]

       1.000    2.000    3.000    4.000
      13.000   14.000   15.000   16.000

    Plane [2,.,.]

       5.000    6.000    7.000    8.000
      17.000   18.000   19.000   20.000

    Plane [3,.,.]

       9.000   10.000   11.000   12.000
      21.000   22.000   23.000   24.000
```

```
nd = { 2,3,1 };
y = atranspose(x,nd);
```

Using the same array *x* as the example above, this example transposes all three dimensions of *x*, returning a 3x4x2 array *y*, such that:

```
Plane [1,.,.]

   1.000   13.000
   2.000   14.000
   3.000   15.000
   4.000   16.000

Plane [2,.,.]

   5.000   17.000
   6.000   18.000
   7.000   19.000
   8.000   20.000

Plane [3,.,.]

   9.000   21.000
  10.000   22.000
  11.000   23.000
  12.000   24.000
```

See Also

areshape

axmargin

Purpose

Sets absolute margins for the plot axes which control placement and size of plot. NOTE: This function is for the deprecated PQG graphics.

Library

`pgraph`

Format

`axmargin`(*`l`*, *`r`*, *`t`*, *`b`*);

Input

`l`	scalar, the left margin in inches.
`r`	scalar, the right margin in inches.
`t`	scalar, the top margin in inches.
`b`	scalar, the bottom margin in inches.

Remarks

`axmargin` sets an absolute distance from the axes to the edge of the graphic panel. Note that the user is responsible for allowing enough space in the margin if axes labels, numbers and title are used on the graph, since **`axmargin`** does not size the plot automatically as in the case of **`margin`**.

All input inch values for this procedure are based on a full size window of 9x6.855 inches. If this procedure is used within a graphic panel, the values will be scaled to window inches automatically.

If both **margin** and **axmargin** are used for a graph, **axmargin** will override any sizes specified by **margin**.

Example

The statement:

```
    axmargin(1,1,.5,.855);
```

will create a plot area of 7 inches horizontally by 5.5 inches vertically, and positioned 1 inch right and .855 up from the lower left corner of the graphic panel/page.

Source

pgraph.src

b

balance

Purpose

Balances a square matrix.

Format

{ *b*, *z* } = **`balance`**(*x*);

Input

x	KxK matrix or N-dimensional array where the last two dimensions are KxK.

Output

b	KxK matrix or N-dimensional array where the last two dimensions are KxK, balanced matrix.
z	KxK matrix or N-dimensional array where the last two dimensions are KxK, diagonal scale matrix.

Remarks

`balance` returns a balanced matrix *b* and another matrix *z* with scale factors in powers of two on its diagonal. *b* is balanced in the sense that the absolute sums of the magnitudes of elements in corresponding rows and columns are nearly equal.

balance is most often used to scale matrices to improve the numerical stability of the calculation of their eigenvalues. It is also useful in the solution of matrix equations.

In particular,

$$b = z^{-1}xz$$

balance uses the BALANC function from EISPACK..

Example

```
    let x[3,3] = 100 200 300
                  40  50  60
                   7   8   9;
    { b,z } = balance(x);

        b = 100.0  100.0  37.5
             80.0   50.0  15.0
             56.0   32.0   9.0

        z = 4.0  0.0  0.0
            0.0  2.0  0.0
            0.0  0.0  0.5
```

band

Purpose

Extracts bands from a symmetric banded matrix.

Format

a = **band**(*y*, *n*);

Input

y	KxK symmetric banded matrix.
n	scalar, number of subdiagonals.

Output

a	Kx(N+1) matrix, 1 subdiagonal per column.

Remarks

y can actually be a rectangular PxQ matrix. K is then defined as min(P,Q). It will be assumed that *a* is symmetric about the principal diagonal for *y*[1:K,1:K].

The subdiagonals of *y* are stored right to left in *a*, with the principal diagonal in the rightmost or (N+1)th column of *a*. The upper left corner of *a* is unused; it is set to 0.

This compact form of a banded matrix is what **bandchol** expects.

Example

```
x = { 1 2 0 0,
      2 8 1 0,
      0 1 5 2,
      0 0 2 3 };

//Extract only the principal diagonal
```

```
b0 = band(x,0);

//Extract the principal diagonal and the first subdiagonal
b1 = band(x,1);

//Extract the principal diagonal and the first two subdiago
nals
b2 = band(x,2);
```

After the code above:

```
        1         0   1          0   0   1
b0 = 8   b1 = 2   8   b2 = 0   2   8
        5         1   5          0   1   5
        3         2   3          0   2   3
```

See Also

bandchol, bandcholsol, bandltsol, bandrv, bandsolpd

bandchol

Purpose

Computes the Cholesky decomposition of a positive definite banded matrix.

Format

l = **bandchol**(*a*);

Input

a	KxN compact form matrix.

Output

l	KxN compact form matrix, lower triangle of the Cholesky decomposition of *a*.

Remarks

Given a positive definite banded matrix *A*, there exists a matrix *L*, the lower triangle of the Cholesky decomposition of *A*, such that $A = LL'$. *a* is the compact form of *A*; see **band** for a description of the format of *a*.

l is the compact form of *L*. This is the form of matrix that **bandcholsol** expects.

Example

```
x = { 1 2 0 0,
      2 8 1 0,
      0 1 5 2,
      0 0 2 3 };

bx = band(x, 1);
bl = bandchol(bx);

l = chol(x);
```

After the code above:

```
        0   1          0   1          1   2   0   0
   bx = 2   8     bl = 2   2      l = 0   2   1   0
        1   5          1   2          0   0   2   1
        2   3          1   1          0   0   0   1
```

See Also

band, bandcholsol, bandltsol, bandrv, bandsolpd

bandcholsol

Purpose

Solves the system of equations $Ax = b$ for x, given the lower triangle of the Cholesky decomposition of a positive definite banded matrix A.

Format

x = **bandcholsol**(*b*, *l*);

Input

b	KxM matrix.
l	KxN compact form matrix.

Output

x	KxM matrix.

Remarks

Given a positive definite banded matrix *A*, there exists a matrix *L*, the lower triangle of the Cholesky decomposition of *A*, such that $A = LL'$. *l* is the compact form of *L*; see **band** for a description of the format of *l*.

b can have more than one column. If so, $Ax = b$ is solved for each column. That is,

```
    A*x[.,i] = b[.,i]
```

Example

```
    //Create matrix 'A' and right-hand side 'b'
    A = { 1 2 0 0,
          2 8 1 0,
          0 1 5 2,
          0 0 2 3 };
    b = { 1.3, 2.1, 0.7, 1.8 };

    //Create banded matrix form of 'A'
    Aband = band(A,1);

    //Cholesky factorization of the banded 'A'
    Lband = bandchol(Aband);

    //Solve the system of equations
    x = bandcholsol(b, Lband);
```

After the code above is run:

```
            0.000  1.000       1.495      1.300         1.300
    Lband = 2.000  2.000  x = -0.098  b = 2.100  A*x = 2.100
            0.500  2.179      -0.110      0.700         0.700
            0.918  1.469       0.673      1.800         1.800
```

See Also

band, bandchol, bandltsol, bandrv, bandsolpd

bandltsol

Purpose

Solves the system of equations $Ax = b$ for x, where A is a lower triangular banded matrix.

Format

```
x = bandltsol(b, A);
```

Input

b	KxM matrix.
A	KxN compact form matrix.

Output

x	KxM matrix.

Remarks

A is a lower triangular banded matrix in compact form. See **band** for a description of the format of *A*.

b can have more than one column. If so, $Ax = b$ is solved for each column. That is,

```
    A*x[.,i] = b[.,i];
```

Example

```
    //Create matrix 'A' and right-hand side 'b'
    A = { 1 2 0 0,
          2 8 1 0,
          0 1 5 2,
          0 0 2 3 };
    b = { 1.3, 2.1, 0.7, 1.8 };

    //Create a matrix containing the lower triangular part
    /of 'A'
    Alower = lowmat(A);

    //Create banded matrix from of 'Alower'
    Abandlow = band(Alower, 1);

    //Solve the system of equations
    x = bandltsol(b, Abandlow);
```

After the code above:

```
             1  0  0  0          0  1      1.300     1.3
       1.3
    Alower = 2  8  0  0 Aband = 2  8 x = -0.063 b = 2.1 Alower*
    x = 2.1
             0  1  5  0          1  5      0.153     0.7
       0.7
             0  0  2  3          2  3      0.498     1.8
       1.8
```

See Also

band, bandchol, bandcholsol, bandrv, bandsolpd

bandrv

Purpose

Creates a symmetric banded matrix, given its compact form.

Format

y = **bandrv**(*a*);

Input

a	KxN compact form matrix.

Output

y	KxK symmetrix banded matrix.

Remarks

a is the compact form of a symmetric banded matrix, as generated by **band**. *a* stores subdiagonals right to left, with the principal diagonal in the rightmost (Nth) column. The upper left corner of *a* is unused. **bandchol** expects a matrix of this form.

y is the fully expanded form of *a*, a KxK matrix with N-1 subdiagonals.

Example

```
    x = { 1 2 0 0,
          2 8 1 0,
          0 1 5 2,
          0 0 2 3 };

    //Create a version of 'x' in band format
    xBand = band(x,1);

    //Expand the banded version of 'x' back to a full matrix
    xNew = bandrv(xBand);
```

After the code above:

```
             0   1       1   2   0   0          1   2   0   0
    xBand =  2   8   x = 2   8   1   0   xNew = 2   8   1   0
             1   5       0   1   5   2          0   1   5   2
             2   3       0   0   2   3          0   0   2   3
```

See Also

band, bandchol, bandcholsol, bandltsol, bandsolpd

bandsolpd

Purpose

Solves the system of equations $Ax = b$ for x, where A is a positive definite banded matrix.

Format

x = **bandsolpd**(b, A);

Input

b	KxM matrix.
A	KxN compact form matrix.

Output

x	KxM matrix.

Remarks

A is a positive definite banded matrix in compact form. See **band** for a description of the format of *A*.

b can have more than one column. If so, $Ax = b$ is solved for each column. That is,

```
    A*x[.,i] = b[.,i]
```

See Also

band, bandchol, bandcholsol, bandltsol, bandrv

b

bar

Purpose

Generates a bar graph. NOTE: This function is for the deprecated PQG graphics, use **plotBar** instead.

Library

pgraph

Format

```
    bar(val, ht);
```

Input

val	Nx1 numeric vector, bar labels. If scalar 0, a sequence from 1 to **rows**(*ht*) will be created.
ht	NxK numeric vector, bar heights.

Global Input

_pbarwid	scalar, width and type of bars in bar graphs and histograms. The valid range is 0-1. If this is 0, the bars will be a single pixel wide. If this is 1, the bars will touch each other. If this value is positive, the bars will overlap. If negative, the bars will be plotted side-by-side. The default is 0.5.
_pbartyp	Kx2 matrix. The first column controls the bar shading:

0	no shading.
1	dots.
2	vertical cross-hatch.
3	diagonal lines with positive slope.

4	diagonal lines with negative slope.
5	diagonal cross-hatch.
6	solid.

The second column controls the bar color.

Remarks

Use **scale** or **ytics** to fix the scaling for the bar heights.

Example

In this example, three overlapping sets of bars will be created. The three heights for the *i*th bar are stored in $x[i,.]$.

```
library pgraph;
graphset;

t = seqa(0,1,10);
x =(t^2/2).*(1~0.7~0.3);

_plegctl = { 1 4 };
_plegstr = "Accnt #1\000Accnt #2\000Accnt #3";
title("Theoretical Savings Balance");
xlabel("Years");
ylabel("Dollars x 1000");
_pbartyp = { 1 10 }; /* Set color of the bars */
_pnum = 2;

bar(t,x); /* Use t vector to label X axis. */
```

Source

```
pbar.src
```

See Also

asclabel, xy, logx, logy, loglog, scale, hist

base10

Purpose

Breaks number into a number of the form #.####... and a power of 10.

Format

{ *M*, *P* } = **base10**(*x*);

Input

x	scalar, number to break down.

Output

M	scalar, in the range $-10 < M < 10$.
P	scalar, integer power such that: $M*10^P = x$

Example

```
    { b, e } = base10(4500);
```

After the code above:

```
    b = 4.5   e = 3
```

and

```
    b*10^e = 4.5*10^3 = 4500
```

Source

base10.src

begwind

Purpose

Initializes global graphic panel variables. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

```
    begwind;
```

Remarks

This procedure must be called before any other graphic panel functions are called.

Source

`pwindow.src`

See Also

endwind, window, makewind, nextwind, getwind

besselj

Purpose

Computes a Bessel function of the first kind, $J_n(x)$.

Format

y = **`besselj`**(n, x);

Input

n	NxK matrix or P-dimensional array where the last two dimensions are NxK, the order of the Bessel function. Nonintegers will be truncated to an integer.
x	LxM matrix or P-dimensional array where the last two dimensions are LxM, ExE conformable with *n*.

Output

y	max(N,L) by max(K,M) matrix or P-dimensional array where the last two dimensions are max(N,L) by max(K,M).

Example

```
    //Create the sequence 0.1, 0.2, 0.3,...,19.9
    x = seqa(0, 0.1, 200);

    //Calculate a first order Bessel function
    ord = 1;
    y0 = besselj(ord, x);

    //Calculate the first and second order Bessel function
    ord = { 1 2 };
    y = besselj(ord, x);

    //Plot the output of the first and third order Bessel
    //functions
    plotXY(x, y);
```

In the code above, the calculation of both the first and second order Bessel functions assigns the return from the first order calculation to be the first column of *y* and the return from the calculation of the second order function to be the second column of *y*.

The **plotXY** function treats each incoming column as a separate line.

See Also

bessely, mbesseli

bessely

Purpose

Computes a Bessel function of the second kind (Weber's function), $Y_n(x)$.

Format

`y = bessely(n, x);`

Input

n	NxK matrix or P-dimensional array where the last two dimensions are NxK, the order of the Bessel function. Nonintegers will be truncated to an integer.
x	LxM matrix or P-dimensional array where the last two dimensions are LxM, ExE conformable with *n*.

Output

y	max(N,L) by max(K,M) matrix or P-dimensional array where the last two dimensions are max(N,L) by max(K,M).

Example

```
//Create the sequence 0.1, 0.2, 0.3, 0.4, 0.5
x = seqa(0.1, 0.1, 5);

//Create the sequence 1, 1.1, 1.2, 1.3, 1.4
```

```
    x2 = seqa(1, 0.1, 5);

    //Calculate a first order bessel function against 'x' and
    //calculate a third order bessel function agains 'x2'
    //NOTE: The '~' provides horizontal concatenation
    ord = { 1 3 };
    y = bessely(ord, x~x2);
```

After the code above:

```
        -6.459 -5.822          0.100  1.000
        -3.324 -4.507          0.200  1.100
    y = -2.293 -3.590  x~x2 =  0.300  1.200
        -1.781 -2.930          0.400  1.300
        -1.471 -2.442          0.500  1.400
```

See Also

besselj, mbesseli

beta

Purpose

Computes the standard Beta function, also called the Euler integral. The beta function is defined as:

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

Format

$f = \textbf{beta}(x, y);$

Input

x	scalar or NxK matrix; *x* may be real or complex.
y	LxM matrix, ExE conformable with *x*.

Output

f	NxK matrix.

Technical Notes

The Beta function's relationship with the Gamma function is:

$$\frac{\mathrm{gamma}(x) \times \mathrm{gamma}(y)}{\mathrm{gamma}(x+y)}$$

See Also

cdfBeta, gamma, gammacplx, zeta

box

Purpose

Graphs data using the box graph percentile method. NOTE: This function uses the deprecated PQG graphics. Use **`plotBox`** instead.

Library

`pgraph`

Format

```
box(grp, y);
```

Input

grp	1xM vector. This contains the group numbers corresponding to each column of *y* data. If scalar 0, a sequence from 1 to **cols**(*y*) will be generated automatically for the X axis.
y	NxM matrix. Each column represents the set of *y* values for an individual percentiles box symbol.

Global Input

_pboxctl 5x1 vector, controls box style, width, and color.

[1] box width between 0 and 1. If zero, the box plot is drawn as two vertical lines representing the quartile ranges with a filled circle representing the 50th percentile.

[2] box color. If this is set to 0, the colors may be individually controlled using the global variable *_pcolor*.

[3] Min/max style for the box symbol. One of the following:

1 Minimum and maximum taken from the actual limits of the data. Elements 4 and 5 are ignored.

		2	Statistical standard with the minimum and maximum calculated according to interquartile range as follows:
			intqrange = 75th - 25th
			min = 25th - 1.5 *intqrange*
			max = 75th + 1.5 *intqrange*
			Elements 4 and 5 are ignored.
		3	Minimum and maximum percentiles taken from elements 4 and 5.
	[4]	Minimum percentile value (0-100) if `_pboxctl`[3] = 3.	
	[5]	Maximum percentile value (0-100) if `_pboxctl`[3] = 3.	
`_plctrl`	1xM vector or scalar as follows:		
	0	Plot boxes only, no symbols.	
	1	Plot boxes and plot symbols which lie outside the `min` and `max` box values.	
	2	Plot boxes and all symbols.	
	-1	Plot symbols only, no boxes.	
	These capabilities are in addition to the usual line control capabilities of `_plctrl`.		
`_pcolor`	1xM vector or scalar for symbol colors. If scalar, all symbols will be one color.		

Remarks

If missing values are encountered in the *y* data, they will be ignored during calculations and will not be plotted.

Source

`pbox.src`

boxcox

Purpose

Computes the Box-Cox function.

Format

y = **boxcox** (*x*, *lambda*);

Input

x	MxN matrix or P-dimensional array where the last two dimensions are MxN.
lambda	KxL matrix or P-dimensional array where the last two dimensions are KxL, ExE conformable to *x*.

Output

y	max(M,L)xmax(N,K) or P-dimensional array where the last two dimensions are max(M,L)xmax(N,K).

Remarks

Allowable range for *x* is: x > 0

The **boxcox** function computes:

```
boxcox(x) = (x^λ - 1)/λ
```

Example

```
x = { .2, .4, .8, 1, 1.2, 1.4 };
lambda = .4;
y = boxcox(x,lambda);
```

After the code above:

```
        -1.187
        -0.767
y = -0.213
         0.000
         0.189
         0.360
```

break

Purpose

Breaks out of a do or for loop.

Format

```
break;
```

Example

```
x = rndn(4,4);

//Loop through each row of 'x' using 'r' as the loop
//counter
for r(1, rows(x), 1);
   //For each row, loop through its elements
   for c(1, cols(x), 1);
      if c == r;      /* Set the diagonal to 1 */
         x[r,c] = 1;
      elseif c > r;   /* leave upper triangle as it is */
         break;       /* terminate inner loop */
      else;
         x[r,c] = 0; /* set lower triangle elements to 0 */
      endif;
   endfor;            /* break jumps to the statement after
                         this endfor */
endfor;
```

After running the code above, *x* should be a lower triangular matrix similar to below. Due to the use of random data, your matrix will have different non-zero elements above the diagonal.

```
1.000  1.288 -0.060  1.801
0.000  1.000  1.609  1.474
0.000  0.000  1.000 -0.768
0.000  0.000  0.000  1.000
```

Remarks

This command works just like in C.

See Also

continue, do, for

call

Purpose

Calls a function or procedure when the returned value is not needed and can be ignored, or when the procedure is defined to return nothing.

Format

```
call function_name(argument_list);
call function_name;
```

Remarks

This is useful when you need to execute a function or procedure and do not need the value that it returns. It can also be used for calling procedures that have been defined to return nothing.

function_name can be any intrinsic **GAUSS** function, a procedure (proc), or any valid expression.

Example

```
call chol(x);
y = detl;
```

The above example is the fastest way to compute the determinant of a positive definite matrix. The result of **`chol`** is discarded and **`detl`** is used to retrieve the determinant that was computed during the call to **`chol`**.

See Also

proc

cdfBeta

Purpose

Computes the incomplete Beta function (i.e., the cumulative distribution function of the Beta distribution).

Format

y = **`cdfBeta`**(*x*, *a*, *b*);

Input

x	NxK matrix.
a	LxM matrix, ExE conformable with *x*.
b	PxQ matrix, ExE conformable with *x* and *a*.

Output

y	max(N,L,P) by max(K,M,Q) matrix.

Remarks

y is the integral from 0 to *x* of the beta distribution with parameters *a* and *b*. Allowable ranges for the arguments are:

```
    0 ≤ x ≤ 1
    a > 0
    b > 0
```

A -1 is returned for those elements with invalid inputs.

Example

```
    x = { .1, .2, .3, .4 };
    a = 0.5;
    b = 0.3;
    y = cdfBeta(x,a,b);
        0.1423
    y = 0.2066
        0.2606
        0.3109
```

See Also

cdfChic, cdfFc, cdfN, cdfNc, cdfTc, gamma

Technical Notes

cdfBeta has the following approximate accuracy:

		max(*a*, *b*)	<=	500	absolute error is approx. ±5e-13
500	<	max(*a*, *b*)	<=	10,000	absolute error is approx. ±5e-11
10,000	<	max(*a*, *b*)	<=	200,000	absolute error is approx. ±1e-9

References

1. Bol'shev, L.N."Asymptotically Perason's Transformations." Teor. Veroyat. Primen. *Theory of Probability and its Applications*. Vol. 8, No. 2, 1963, 129-55.
2. Boston N.E. and E.L. Battiste. "Remark on Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 17, No. 3, March 1974, 156-57.
3. Ludwig, O.G. "Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 6, No. 6, June 1963, 314.
4. Mardia, K.V. and P.J. Zemroch. *Tables of the F- and related distributions with algorithms*. Academic Press, New York, 1978. ISBN 0-12-471140-5.
5. Peizer, D.B. and J.W. Pratt. "A Normal Approximation for Binomial, F, Beta, and Other Common, Related Tail Probabilities, I." *Journal of the American Statistical Association*. Vol. 63, Dec. 1968, 1416-56.
6. Pike, M.C. and J.W. Pratt. "Remark on Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 10, No. 6, June 1967, 375-76.

cdfBetaInv

Purpose

Computes the quantile or inverse of the Beta cumulative distribution function.

Format

x = **cdfBetaInv**(*p*,*a*,*b*);

Input

p	NxK matrix, Nx1 vector or scalar. $0 < p < 1$.
a	ExE conformable with *p*. $0 < a$.
b	ExE conformable with *p*. $0 < b$.

Output

x	NxK matrix, Nx1 vector or scalar.

Remarks

For invalid inputs, **`cdfBetaInv`** will return a scalar error code which, when its value is assessed by function **`scalerr`**, corresponds to the invalid input. If the first input is out of range, **`scalerr`** will return a 1; if the second is out of range, **`scalerr`** will return a 2; etc.

See Also

cdfBeta, cdfBinomial, cdfNegBinomial

cdfBinomial

Purpose

Computes the binomial cumulative distribution function.

Format

p = **`cdfBinomial`**(successes,trials,prob);

Input

successes	NxK matrix, Nx1 vector or scalar. *successes* must be a positive number and < trials
trials	ExE conformable with successes. *trials* must be >

	successes.
prob	The probability of success on any given trial. ExE conformable with *successes*. 0 < prob < 1.

Output

p	NxK matrix, Nx1 vector or scalar.

Example

What are the chances that a baseball player with a long-term batting average of .317 could break Ichiro Suzuki's record of 270 hits in a season if he had as many at bats as Ichiro had that year, 704?

```
p = cdfBinomial(270,704,.317); /* The cumulative
                                  probability of our player
                                  getting 270 or fewer hits
                                  in the season */
p = 0.9999199430052614
```

Therefore the odds of this player breaking Ichiro's record:

```
= 1-p
= 0.0000000000037863 or 0.0000000003786305%
```

Remarks

For invalid inputs, **cdfBinomial** will return a scalar error code which, when its value is assessed by function **scalerr**, corresponds to the invalid input. If the first input is out of range, **scalerr** will return a 1; if the second is out of range, **scalerr** will return a 2; etc.

See Also

cdfBinomialInv, cdfNegBinomial

cdfBinomialInv

Purpose

Computes the binomial quantile or inverse cumulative distribution function.

Format

s = **`cdfBinomialInv`**(p,trials,prob);

Input

p	NxK matrix, Nx1 vector or scalar. $0 < p < 1$.
trials	ExE conformable with *p*. trials > 0.
prob	The probability of success on any given trial. ExE conformable with *p*. 0 < prob < 1.

Output

s	The number of successes. NxK matrix, Nx1 vector or scalar.

Example

What is a reasonable range of wins for a basketball team playing 82 games in a season, with a 60% chance of winning any game? For our example we will define a reasonable range as falling between the top and bottom deciles.

```
    range = { .10, .9 };
    s = cdfBinomialInv(range,82,.6);
```

```
    s = 43 55
```

This means that a team with a 60% chance of winning any one game would win between 43 and 55 games in 80% of seasons.

Remarks

For invalid inputs, **cdfBinomialInv** will return a scalar error code which, when its value is assessed by function **scalerr**, corresponds to the invalid input. If the first input is out of range, **scalerr** will return a 1; if the second is out of range, **scalerr** will return a 2; etc.

See Also

cdfBinomial, cdfNegBinomial, cdfNegBinomialInv

cdfBvn

Purpose

Computes the cumulative distribution function of the standardized bivariate Normal density (lower tail).

Format

c = **cdfBvn**(*h*, *k*, *r*);

Input

`h`	NxK matrix, the upper limits of integration for variable 1.
`k`	LxM matrix, ExE conformable with `h`, the upper limits of integration for variable 2.
`r`	PxQ matrix, ExE conformable with `h` and `k`, the correlation coefficients between the two variables.

Output

`c`	max(N,L,P) by max(K,M,Q) matrix, the result of the double integral from -∞ to `h` and -∞ to `k` of the standardized bivariate Normal density $f(x, y, r)$.

Remarks

The function integrated is:

$$f(x, y, r) = \frac{e^{-0.5w}}{2\pi\sqrt{-r^2}}$$

with

$$w = \frac{x^2 - 2rxy + y^2}{1 - r^2}$$

Thus, `x` and `y` have 0 means, unit variances, and correlation = `r`.

Allowable ranges for the arguments are:

```
    -∞ ≤ h ≤ +∞
    -∞ ≤ k ≤ +∞
    -1 < r < 1
```

A -1 is returned for those elements with invalid inputs.

To find the integral under a general bivariate density, with *x* and *y* having nonzero means and any positive standard deviations, use the transformation equations:

```
    h = (ht - ux) ./ sx;
    k = (kt - uy)
```

where `ux` and `uy` are the (vectors of) means of *x* and *y*, `sx` and `sy` are the (vectors of) standard deviations of *x* and *y*, and `ht` and `kt` are the (vectors of) upper integration limits for the untransformed variables, respectively.

See Also

cdfN, cdfTvn

Technical Notes

The absolute error for **`cdfBvn`** is approximately ±5.0e-9 for the entire range of arguments.

References

1. Daley, D.J. "Computation of Bi- and Tri-variate Normal Integral." *Appl. Statist.* Vol. 23, No. 3, 1974, 435-38.
2. Owen, D.B. "A Table of Normal Integrals." *Commun. Statist.-Simula. Computa.*, B9(4). 1980, 389-419.

cdfBvn2

Purpose

Returns the bivariate Normal cumulative distribution function of a bounded rectangle.

Format

y = **cdfBvn2**(*h*, *dh*, *k*, *dk*, *r*);

Input

h	Nx1 vector, starting points of integration for variable 1.
dh	Nx1 vector, increments for variable 1.
k	Nx1 vector, starting points of integration for variable 2.
dk	Nx1 vector, increments for variable 2.
r	Nx1 vector, correlation coefficients between the two variables.

Output

y	Nx1 vector, the integral over the rectangle bounded by *h*, *h* + *dh*, *k*, and *k* + *dk* of the standardized bivariate Normal distribution.

Remarks

Scalar input arguments are okay; they will be expanded to Nx1 vectors.

cdfBvn2 computes:

```
    cdfBvn(h + dh, k+ dk, r) + cdfBvn(h, k, r) - cdfBvn(h,
    k + dk, r) - cdfBvn(h + dh, k, r)
```

cdfBvn2 computes an error estimate for each set of inputs. The size of the error depends on the input arguments. If **trap 2** is set, a warning message is displayed when the error reaches 0.01***abs**(*y*). For an estimate of the actual error, see **cdfBvn2e**.

Example

Example 1

```
    print cdfBvn2(1,-1,1,-1,0.5);
       1.4105101488974692e-001
```

Example 2

```
    print cdfBvn2(1,-1e-15,1,-1e-15,0.5);
       4.9303806576313238e-32
```

Example 3

```
    print cdfBvn2(1,-1e-45,1,-1e-45,0.5);
       0.0000000000000000e+000
```

Example 4

```
    trap 2,2;
    print cdfBvn2(1,-1e-45,1,1e-45,0.5);
    WARNING: Dubious accuracy from cdfBvn2:
    0.000e+000 +/- 2.8e-060
    0.0000000000000000e+000
```

Source

`lncdfn.src`

See Also

cdfBvn2e, lncdfbvn2

cdfBvn2e

Purpose

Returns the bivariate Normal cumulative distribution function of a bounded rectangle.

Format

{ *y*, *e* } = **cdfBvn2e**(*h*, *dh*, *k*, *dk*, *r*);

Input

h	Nx1 vector, starting points of integration for variable 1.
dh	Nx1 vector, increments for variable 1.
k	Nx1 vector, starting points of integration for variable 2.
dk	Nx1 vector, increments for variable 2.

r	Nx1 vector, correlation coefficients between the two variables.

Output

y	Nx1 vector, the integral over the rectangle bounded by *h*, *h* + *dh*, *k*, and *k* + *dk* of the standardized bivariate Normal distribution.
e	Nx1 vector, an error estimate.

Remarks

Scalar input arguments are okay; they will be expanded to Nx1 vectors. **cdfBvn2e** computes:

```
cdfBvn(h + dh, k + dk, r) + cdfBvn(h, k, r) - cdfBvn(h,
k + dk, r) - cdfBvn(h + dh, k, r)
```

The real answer is *y* ± *e*. The size of the error depends on the input arguments.

Example

Example 1

```
print cdfBvn2e(1,-1,1,-1,0.5);

   1.4105101488974692e-001
   1.9927918166193113e-014
```

Example 2

C

```
    print cdfBvn2e(1,-1e-15,1,-1e-15,0.5);

       7.3955709864469857e-032
       2.8306169312687801e-030
```

Example 3

```
    print cdfBvn2e(1,-1e-45,1,-1e-45,0.5);

       0.0000000000000000e+000
       2.8306169312687770e-060
```

See Also

cdfBvn2, lncdfbvn2

cdfCauchy

Purpose

Computes the cumulative distribution function for the Cauchy distribution.

Format

y = **cdfCauchy**(*x*,*a*,*b*);

Input

x	NxK matrix, an Nx1 vector or scalar.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.

b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The cumulative distribution function for the Cauchy distribution is defined as:

$$\frac{1}{2}+\frac{1}{\pi}arctan\left(\frac{x-a}{b}\right)$$

See Also

pdfCauchy

cdfCauchyInv

Purpose

Computes the Cauchy inverse cumulative distribution function.

Format

```
y = cdfCauchyInv(p,a,b);
```

C

Input

p	NxK matrix, Nx1 vector or scalar. *p* must be greater than zero and less than 1.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *p*.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *p*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

See Also

pdfCauchy, cdfCauchy

cdfChic

Purpose

Computes the complement of the cdf of the chi-square distribution.

Format

y = **cdfChic**(*x*, *n*);

Input

x	NxK matrix.
n	LxM matrix, ExE conformable with *x*.

Output

y	max(N,L) by max(K,M) matrix.

Remarks

y is the integral from *x* to ∞ of the chi-square distribution with *n* degrees of freedom.

The elements of *n* must all be positive integers. The allowable ranges for the arguments are:

```
    x > 0
    n > 0
```

A -1 is returned for those elements with invalid inputs.

This equals $1 - X_n^2(x)$, Thus, to get the chi-squared cdf, subtract **cdfChic**(*x*, *n*) from 1. The complement of the cdf is computed because this is what is most commonly needed in statistical applications, and because it can be computed with fewer problems of roundoff error.

Example

```
    x = { .1, .2, .3, .4 };
    n = 3;
    y = cdfChic(x,n);

        0.991837
    y = 0.977589
        0.960028
        0.940242
```

See Also

cdfBeta, cdfFc, cdfNc, cdfTc, gamma

Technical Notes

For n <= 1000, the incomplete gamma function is used and the absolute error is approx. ±6e-13.

For n > 1000, a Normal approximation is used and the absolute error is ±2e-8.

For higher accuracy when n > 1000, use:

```
    1 - cdfGam(0.5*x, 0.5*n);
```

References

1. Bhattacharjee, G.P. "Algorithm AS 32, the Incomplete Gamma Integral." *Applied Statistics*. Vol. 19, 1970, 285-87.

2. Mardia K.V. and P.J. Zemroch. *Tables of the F- and related distributions with algorithms*. Academic Press, New York, 1978. ISBN 0-12-471140-5.

3. Peizer, D.B. and J.W. Pratt. "A Normal Approximation for Binomial, F, Beta, and other Common, Related Tail Probabilities, I." *Journal of the American Statistical Association*. Vol. 63, Dec. 1968, 1416-56.

cdfChii

Purpose

Compute chi-square abscissae values given probability and degrees of freedom.

Format

c = **`cdfChii`**(*p*, *n*);

Input

p	MxN matrix, probabilities.
n	LxK matrix, ExE conformable with *p*, degrees of freedom.

Output

c	max(M,L) by max(N,K) matrix, abscissae values for chi-squared distribution.

Example

The following generates a 3x3 matrix of pseudo-random numbers with a chi-squared distribution with expected value of 4:

```
//Set the rng seed for repeatable random numbers
rndseed 464578;

//Set the 'probabilities' input equal to a 3x3 matrix of
```

```
//uniform random numbers and the degrees of freedom' input
//to be a 3x3 matrix with each element equal to '4'
x = cdfChii(rndu(3,3),4+zeros(3,3));
```

After the code above:

```
     0.934227 6.231914 4.227479
x =  2.647158 1.203957 10.559593
     5.868060 1.368600 1.963283
```

Source

`cdfchii.src`

See Also

gammaii

cdfChinc

Purpose

Computes the cumulative distribution function for the noncentral chi-square distribution.

Format

y = **cdfChinc**(*x*, *v*, *d*);

Input

x	Nx1 vector, values of upper limits of integrals, must be

	greater than 0.
v	scalar, degrees of freedom, *v*> 0.
d	scalar, noncentrality parameter, *d*> 0. This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. (See Scheffe, *The Analysis of Variance*, App. IV, 1959.)

Output

y	Nx1 vector.

Remarks

y is the integral from 0 to *x* of the noncentral chi-square distribution with *v* degrees of freedom and noncentrality *d*.

cdfChinc can return a vector of values, but the degrees of freedom and noncentrality parameter must be the same for all values of *x*.

For invalid inputs, **cdfChinc** will return a scalar error code which, when its value is assessed by function **scalerr**, corresponds to the invalid input. If the first input is out of range, **scalerr** will return a 1; if the second is out of range, **scalerr** will return a 2; etc.

Relation to **cdfChic**:

```
    cdfChic(x, v) = 1 - cdfChinc(x, v, 0);
```

Example

```
x = { .5, 1, 5, 25 };
print cdfChinc(x,4,2);
```

The code above returns:

```
 0.0042086234
 0.016608592
 0.30954232
 0.99441140
```

See Also

cdfFnc, cdfTnc

cdfChincInv

Purpose

Computes the quantile or inverse of noncentral chi-square cumulative distribution function.

Format

x = **cdfChincInv**(y, df, nonc);

Input

y	NxK matrix, Nx1 vector or scalar. The integral from 0 to *x*.

df	ExE conformable with *y*. The degrees of freedom. `df > 0`.
nonc	ExE conformable with *y*. The noncentrality parameter. Note: This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. `nonc > 0`.

Output

x	NxK matrix, Nx1 vector or scalar. The upper limit of the integrals of the noncentral chi-square distribution with *df* degrees of freedom and noncentrality *nonc*.

Remarks

Note: Input *nonc* is the square root of the noncentrality parameter that sometimes goes under the symbol lambda.

For invalid inputs, **cdfChincinv** will return a scalar error code which, when its value is assessed by function **scalerr**, corresponds to the invalid input. If the first input is out of range, **scalerr** will return a 1; if the second is out of range, **scalerr** will return a 2; etc.

See Also

cdfChinc, cdfChic, cdfFnc, cdfTnc

cdfExp

Purpose

Computes the cumulative distribution function for the exponential distribution.

Format

```
y = cdfExp(x,a,m);
```

Input

x	NxK matrix, an Nx1 vector or scalar.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *a* must be less than *x*.
m	Mean parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *m* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The cumulative distribution function for the exponential distribution is defined as

$$1 - exp\left(-\frac{x-a}{b}\right)$$

See Also

pdfExp

cdfExpInv

Purpose

Computes the exponential inverse cumulative distribution function.

Format

```
y = cdfExpInv(p,a,b);
```

Input

p	NxK matrix, Nx1 vector or scalar. *p* must be greater than zero and less than 1.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *p*.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *p*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

See Also

pdfExp, cdfExp

cdfFc

Purpose

Computes the complement of the cumulative distribution function of the *F* distribution.

Format

y = **`cdfFc`**(*x*, *n1*, *n2*);

Input

x	NxK matrix.
n1	LxM matrix, ExE conformable with *x*.
n2	PxQ matrix, ExE conformable with *x* and *n1*.

Output

y	max(N,L,P) by max(K,M,Q) matrix

Remarks

y is the integral from *x* to ∞ of the *F* distribution with *n1* and *n2* degrees of freedom.

This equals

```
    1 - G(x, n1, n2)
```

where *G* is the *F* cdf with *n1* and *n2* degrees of freedom. Thus, to get the *F* cdf, use:

```
    1 - cdfFc(x, n1, n2);
```

The complement of the cdf is computed because this is what is most commonly needed in statistical applications, and because it can be computed with fewer problems of roundoff error.

Allowable ranges for the arguments are:

```
     x > 0
    n1 > 0
    n2 > 0
```

A -1 is returned for those elements with invalid inputs.

For max(n1,n2) <= 1000, the absolute error is approx. ±5e-13. For max(n1,n2) > 1000, Normal approximations are used and the absolute error is approx. ±2e-6.

For higher accuracy when max(n1,n2) > 1000, use

```
    cdfBeta(n2/(n2 + n1*x), n2/2, n1/2);
```

Example

```
    x = { .1, .2, .3, .4 };
    n1 = 0.5;
    n2 = 0.3;
    print cdfFc(x,n1,n2);
```

The code above, produces:

```
     0.751772
     0.708152
     0.680365
     0.659816
```

See Also

cdfBeta, cdfChic, cdfN, cdfNc, cdfTc, gamma

References

1. Bol'shev, L.N. "Asymptotically Perason's Transformations." Teor. Veroyat. Primen. *Theory of Probability and its Applications*. Vol. 8, No. 2, 1963, 129-55.
2. Bosten, N.E. and E.L. Battiste. "Remark on Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 17, No. 3, March 1974, 156-57.
3. Kennedy, W.J., Jr. and J.E. Gentle. *Statistical Computing*. Marcel Dekker, Inc., New York, 1980.
4. Ludwig, O.G. "Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 6, No. 6, June 1963, 314.
5. Mardia, K.V. and P.J. Zemroch. *Tables of the F- and related distributions with algorithms*. Academic Press, New York, 1978. ISBN 0-12-471140-5.
6. Peizer, D.B. and J.W. Pratt. "A Normal Approximation for Binomial, F, Beta, and other Common, Related Tail Probabilities, I." *Journal of the American Statistical Association*. Vol. 63, Dec. 1968, 1416-56.
7. Pike, M.C. and I.D. Hill, "Remark on Algorithm 179 Incomplete Beta Ratio."

Comm. ACM. Vol. 10, No. 6, June 1967, 375-76.

cdfFnc

Purpose

Computes the cumulative distribution function of the noncentral *F* distribution.

Format

```
y = cdfFnc(x, n1, n2, d);
```

Input

x	Nx1 vector, values of upper limits of integrals, $x > 0$.
v1	scalar, degrees of freedom of numerator, $n1 > 0$.
v2	scalar, degrees of freedom of denominator, $n2 > 0$.
d	scalar, noncentrality parameter, $d > 0$.
	This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. (See Scheffe, *The Analysis of Variance*, App. IV, 1959.)

Output

y	Nx1 vector.

Remarks

For invalid inputs, **cdfFnc** will return a scalar error code which, when its value is assessed by function **scalerr**, corresponds to the invalid input. If the first input is out of range, **scalerr** will return a 1; if the second is out of range, **scalerr** will return a 2; etc.

Technical Notes

Relation to cdfFc:

```
    cdfFc(x, n1, n2) = 1 - cdfFnc(x, n1, n2,0);
```

See Also

cdfTnc, cdfChinc

cdfFncInv

Purpose

Computes the quantile or inverse of noncentral *F* cumulative distribution function.

Format

x = **cdfFncInv**(y, dfn, dfd, nonc);

Input

y	NxK matrix, Nx1 vector or scalar.

dfn	ExE conformable with *y*. The degrees of freedom numerator. *dfn > 0*.
dfd	ExE conformable with *y*. The degrees of freedom denominator. *dfd > 0*.
nonc	ExE conformable with *y*. The noncentrality parameter. Note: This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. *nonc > 0*.

Output

x	NxK matrix, Nx1 vector or scalar. The upper limit of the integrals of the noncentral *F* distribution.

Remarks

Note: Input *nonc* is the square root of the noncentrality parameter that sometimes goes under the symbol lambda.

For invalid inputs, **cdfFncInv** will return a scalar error code which, when its value is assessed by function **scalerr**, corresponds to the invalid input. If the first input is out of range, **scalerr** will return a 1; if the second is out of range, **scalerr** will return a 2; etc.

See Also

cdfFnc, cdfChinc, cdfChic, cdfTnc

cdfGam

Purpose

Computes the incomplete gamma function.

Format

```
g = cdfGam(x, intlim);
```

Input

x	NxK matrix of data.
intlim	LxM matrix, ExE compatible with *x*, containing the integration limit.

Output

g	max(N,L) by max(K,M) matrix.

Remarks

The incomplete gamma function returns the integral

$$\int_0^{int\ lim} \frac{e^{-t} t^{(x-1)}}{gamma(x)} dt$$

The allowable ranges for the arguments are:

```
          x > 0
     intlim > 0
```

A -1 is returned for those elements with invalid inputs.

Example

```
     x = { 0.5 1 3 10 };
     intlim = seqa(0,.2,6);
     g = cdfGam(x,intlim);
```

After the code above:

```
              0.000000      0.000000 0.000000 0.000000 0.000000
              0.200000      0.472911 0.181269 0.001148 0.000000
     intlim = 0.400000 g =  0.628907 0.329680 0.007926 0.000000
              0.600000      0.726678 0.451188 0.023115 0.000000
              0.800000      0.794097 0.550671 0.047423 0.000000
              1.000000      0.842701 0.632121 0.080301 0.000000
```

This computes the integrals over the range from 0 to 1, in increments of 0.2, at the parameter values 0.5, 1, 3, 10.

Technical Notes

cdfGam has the following approximate accuracy:

```
                 x < 500      : the absolute error is approx. ±6e-13
          500 <= x <= 10,000  : the absolute error is approx. ±3e-11
       10,000 <  x            : a Normal approximation is used and
                                the absolute error is approx. ±3e-10
```

References

1. Bhattacharjee, G.P. "Algorithm AS 32, the Incomplete Gamma Integral." *Applied Statistics*. Vol. 19, 1970, 285-87.
2. Mardia, K.V. and P.J. Zemroch. *Tables of the F- and Related Distributions with Algorithms*. Academic Press, New York, 1978. ISBN 0-12-471140-5.
3. Peizer, D.B. and J.W. Pratt. "A Normal Approximation for Binomial, F, Beta, and other Common, Related Tail Probabilities, I." *Journal of the American Statistical Association*. Vol. 63, Dec. 1968, 1416-56.

cdfGenPareto

Purpose

Computes the cumulative distribution function for the Generalized Pareto distribution.

Format

```
y = cdfGenPareto(x,a,o,k);
```

Input

x	NxK matrix, an Nx1 vector or scalar.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.
o	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *o* must be greater than 0.

k	Shape parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The cumulative distribution function for the Generalized Pareto distribution is defined as:

$$f(x) = \begin{cases} 1 - \left(1 + k\frac{(x-\mu}{\sigma}\right)^{-1/k} & k \neq 0 \\ 1 - exp\left(-\frac{(x-\mu)}{\sigma}\right) & k = 0 \end{cases}$$

See Also

pdfGenPareto

cdfLaplace

Purpose

Computes the cumulative distribution function for the Laplace distribution.

Format

```
y = cdfLaplace(x,a,b);
```

Input

x	NxK matrix, an Nx1 vector or scalar.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The cumulative distribution function for the Laplace distribution is defined as

$$F(x) = \begin{cases} \frac{1}{2}exp(-\lambda(\mu - x)) & X \le \mu \\ 1 - \frac{1}{2}exp(-\lambda(\mu - x)) & X > \mu \end{cases}$$

See Also

cdfLaplaceInv

cdfLaplaceInv

Purpose

Computes the Laplace inverse cumulative distribution function.

Format

```
y = cdfLaplaceInv(p,a,b);
```

Input

p	NxK matrix, Nx1 vector or scalar. *p* must be greater than 0 and less than 1.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *p*.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *p*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

See Also

cdfLaplace

cdfLogistic

Purpose

Computes the cumulative distribution function for the logistic distribution.

Format

```
y = cdfLogistic(x,a,b);
```

Input

x	NxK matrix, an Nx1 vector or scalar.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The cumulative distribution function for the logistic distribution is defined as:

$$F(x) = \frac{1}{1 + exp(-z)}$$

where

$$z \equiv \frac{x - \mu}{\sigma}$$

See Also

pdfLogistic

cdfLogisticInv

Purpose

Computes the logistic inverse cumulative distribution function.

Format

y = **`cdfLogisticInv`**(*p*, *a*, *b*);

Input

p	NxK matrix, Nx1 vector or scalar. *p* must be greater than 0 and less than 1.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *p*.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *p*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

See Also

pdfLogistic, cdfLogistic

cdfMvn

Purpose

Computes multivariate Normal cumulative distribution function.

Format

y = **cdfMvn**(*x*, *r*);

Input

x	KxL matrix, abscissae.
r	KxK matrix, correlation matrix.

Output

y	Lx1 vector, $Pr(X < x \mid r)$.

Source

`lncdfn.src`

See Also

cdfBvn, cdfN, lncdfmvn

cdfMvnce

Purpose

Computes the complement of the multivariate Normal cumulative distribution function with error management.

Format

{*y*, *err*, *retcode*} = **cdfMvnce**(*ctl*, *x*, *r*, *m*);

Input

ctl	instance of a **cdfmControl** structure with members.	
	ctl.maxEvaluations	scalar, maximum number of evaluations.
	ctl.absErrorTolerance	scalar absolute error tolerance.
	ctl.relative	error tolerance.
x	NxK matrix, abscissae.	
r	KxK matrix, correlation matrix.	
m	Kx1 vector, means.	

Output

y	Lx1 vector, $Pr(X > x \mid r,m)$.	
err	Lx1 vector, estimates of absolute error.	
retcode	Lx1 vector, return codes,	
	0	normal completion with *err < ctl.absErrorTolerance*.
	1	*err > ctl.absErrorTolerance* and *ctl.maxEvaluations* exceeded; increase *ctl.maxEvaluations* to decrease error.
	2	K > 100 or K < 1.
	3	R not positive semi-definite.
	missing	R not properly defined.

Remarks

cdfMvne evaluates the following integral

$$\phi(x_i, R, m) = \frac{1}{\sqrt{|R|\ (2\pi)^m}} \int_{x_{i1}}^{\infty} \int_{x_{i2}}^{\infty} \cdots \int_{x_{iK}}^{\infty} e^{-\frac{1}{2}\left(x-m'\right)'R^{-1}\left(z-m'\right)} dz$$

Source

cdfm.src

See Also

cdfMvn2e, cdfMvnce, cdfMvte

References

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts", *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.
2. Genz, A., "Numerical computation of multivariate normal probabilities", *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfMvne

Purpose

Computes multivariate Normal cumulative distribution function with error management.

Format

```
{y, err, retcode} = cdfMvne(ctl, x, r, m);
```

Input

ctl	instance of a **cdfmControl** structure with members.	
	ctl.maxEvaluations	scalar, maximum number of evaluations.
	ctl.absErrorTolerance	scalar absolute error tolerance.
	ctl.relative	error tolerance.
x	NxK matrix, abscissae.	

r	KxK matrix, correlation matrix.
m	Kx1 vector, means.

Output

y	Lx1 vector, $Pr(X < x \mid r,m)$.	
err	Lx1 vector, estimates of absolute error.	
retcode	Lx1 vector, return codes.	
	0	normal completion with `err < ctl.absErrorTolerance`.
	1	`err > ctl.absErrorTolerance` and `ctl.maxEvaluations` exceeded; increase `ctl.maxEvaluations` to decrease error
	2	K > 100 or K < 1
	3	`R` not positive semi-definite
	missing	`R` not properly defined

Example

```
    x = { 0.5 -0.5,
          0.1  0.7,
          1.3 -0.9 };

    cor = { 1  0,
            0  1 };
```

```
m = { 0, 0, 0 };

//Declare cdfmControl structure
struct cdfmControl ctl;

//Fill in control structure
//with default values
ctl = cdfmControlCreate();

{ y, err, ret } = cdfmvne(ctl, x, cor, m);
```

After the code above:

```
y =
        0.21334213
        0.40920912
        0.16624302
```

Remarks

cdfMvne evaluates the following integral

$$\Phi(x_i, R, m) = \frac{1}{\sqrt{|R|\,(2\pi)^m}} \int_{x_{i1}}^{\infty} \int_{x_{i2}}^{\infty} \dots \int_{x_{iK}}^{\infty} e^{-\frac{1}{2}(z-m')'R^{-1}(z-m')'}\, dz$$

Source

`cdfm.src`

See Also

cdfmControlCreate, cdfMvn2e, cdfMvte

References

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts," *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.
2. Genz, A., "Numerical computation of multivariate normal probabilities," *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfMvn2e

Purpose

Computes the multivariate Normal cumulative distribution function with error management over the range [a,b].

Format

```
{y, err, retcode} = cdfMvn2e(ctl, a, b, r, m);
```

Input

ctl	instance of a **cdfmControl** structure with members.	
	ctl.maxEvaluations	scalar, maximum number of evaluations.
	ctl.absErrorTolerance	scalar absolute error tolerance.
	ctl.relative	error tolerance.
a	NxK matrix, lower limits.	

b	NxK matrix, upper limits.
r	KxK matrix, correlation matrix.
m	Kx1 vector, means.

Output

y	Lx1 vector, $Pr(X > a$ and $X < b \mid r,m)$.	
err	Lx1 vector, estimates of absolute error.	
retcode	Lx1 vector, return codes.	
	0	normal completion with `err < ctl.absErrorTolerance`.
	1	`err > ctl.absErrorTolerance` and `ctl.maxEvaluations` exceeded; increase `ctl.maxEvaluations` to decrease error.
	2	K > 100 or K < 1.
	3	`R` not positive semi-definite.
	missing	`R` not properly defined.

Remarks

`cdfMvne` evaluates the following integral

$$\Phi(a_i, b_i, R, m) = \frac{1}{\sqrt{|R|\,(2\pi)^m}} \int_{a_{i1}}^{b_{i1}} \int_{a_{i2}}^{b_{i2}} \cdots \int_{a_{iK}}^{b_{iK}} e^{-\frac{1}{2}(z-m')'R^{-1}(z-m')'}\, dz$$

Source

`cdfm.src`

See Also

cdfmControlCreate, cdfMvne, cdfMvnce, cdfMvt2e

References

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts," *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.
2. Genz, A., "Numerical computation of multivariate normal probabilities," *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfMvtce

Purpose

Computes complement of multivariate Student's t cumulative distribution function with error management.

Format

{*y*, *err*, *retcode*} = **cdfMvtce**(*ctl*, *x*, *R*, *m*, *n*);

Input

ctl	instance of a **cdfmControl** structure with members.	
	ctl.maxEvaluations	scalar, maximum number of

		evaluations.
	`ctl.absErrorTolerance`	scalar absolute error tolerance.
	`ctl.relErrorTolerance`	tolerance.
`x`	NxK matrix, abscissae.	
`R`	KxK matrix, correlation matrix.	
`m`	Kx1 vector, noncentralities.	
`n`	scalar, degrees of freedom.	

Output

`y`	Lx1 vector, $Pr(X > x \vert r,m)$.	
`err`	Lx1 vector, estimates of absolute error.	
`retcode`	Lx1 vector, return codes.	
	`0`	normal completion with `err < ctl.absErrorTolerance`.
	`1`	`err > ctl.absErrorTolerance` and `ctl.maxEvaluations` exceeded; increase `ctl.maxEvaluations` to decrease error.
	`2`	K > 100 or K < 1.
	`3`	`R` not positive semi-definite.
	`missing`	`R` not properly defined.

Remarks

The central multivariate Student's t cdf for the i-th row of x is defined by

$$T(x_i, R, n) = \frac{\Gamma\left(\frac{n+K}{2}\right)}{\Gamma\left(\frac{n}{2}\right)\sqrt{|R|\,(n\pi)^K}} \int_{x_{i1}}^{\infty} \int_{x_{i2}}^{\infty} \dots \int_{x_{iK}}^{\infty} \left(1 + \frac{z'\Sigma^{-1}z}{n}\right)^{-\frac{n+K}{2}} dz$$

$$\equiv \frac{2^{1-\frac{n}{2}}}{\Gamma\left(\frac{n}{2}\right)} \int_{x_{i1}}^{\infty} s^{n-1} e^{-\frac{x^2}{2}} \Phi\left(-\infty, \frac{sx_i}{\sqrt{n}}, R\right) ds$$

where

$$\Phi(x_i, R, m) = \frac{1}{\sqrt{|R|\,(2\pi)^m}} \int_{x_{i1}}^{\infty} \int_{x_{i2}}^{\infty} \dots \int_{x_{iK}}^{\infty} e^{-\frac{1}{2}(z-m')'R^{-1}(z-m')'} dz$$

For the noncentral cdf we have

$$T(x_i, R, n, m) = \frac{2^{1-\frac{n}{2}}}{\Gamma\left(\frac{n}{2}\right)} \int_{0}^{\infty} s^{n-1} e^{-\frac{x^2}{2}} \Phi\left(\frac{sx_i}{\sqrt{n}} - m', \infty, R\right) ds$$

Source

cdfm.src

See Also

cdfMvt2e, cdfMvtce, cdfMvne

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts," *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.

2. Genz, A., "Numerical computation of multivariate normal probabilities," *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfMvte

Purpose

Computes multivariate Student's t cumulative distribution function with error management.

Format

$\{y,\ err,\ retcode\}$ = **cdfMvte**(*ctl*, *x*, *R*, *m*, *n*);

Input

ctl	instance of a **cdfmControl** structure with members.	
	ctl.maxEvaluations	scalar, maximum number of evaluations.
	ctl.absErrorTolerance	scalar absolute error tolerance.
	ctl.relErrorTolerance	tolerance.
x	NxK matrix, abscissae.	
R	KxK matrix, correlation matrix.	
m	Kx1 vector, noncentralities.	
n	scalar, degrees of freedom.	

Output

y	Lx1 vector, $Pr(X < x \vert r,m)$.	
err	Lx1 vector, estimates of absolute error.	
retcode	Lx1 vector, return codes.	
	0	normal completion with *err < ctl.absErrorTolerance*.
	1	*err > ctl.absErrorTolerance* and *ctl.maxEvaluations* exceeded; increase *ctl.maxEvaluations* to decrease error.
	2	K > 100 or K < 1.
	3	*R* not positive semi-definite.
	missing	*R* not properly defined.

Remarks

The central multivariate Student's t cdf for the i-th row of *x* is defined by

$$T(x_i, R, n) = \frac{\Gamma\left(\frac{n+K}{2}\right)}{\Gamma\left(\frac{n}{2}\right)\sqrt{|R|\,(n\pi)^K}} \int_{x_{i1}}^{\infty}\int_{x_{i2}}^{\infty}\ldots\int_{x_{iK}}^{\infty}\left(1 + \frac{z'\Sigma^{-1}z}{n}\right)^{-\frac{n+K}{2}} dz$$

$$\equiv \frac{2^{1-\frac{n}{2}}}{\Gamma\left(\frac{n}{2}\right)} \int_{x_{i1}}^{\infty} s^{n-1} e^{-\frac{x^2}{2}} \Phi\left(-\infty, \frac{sx_i}{\sqrt{n}}, R\right) ds$$

where

$$\Phi(x_i, R, m) = \frac{1}{\sqrt{|R|\,(2\pi)^m}} \int_{x_{i1}}^{\infty} \int_{x_{i2}}^{\infty} \ldots \int_{x_{iK}}^{\infty} e^{-\frac{1}{2}z'R^{-1}z} dz$$

For the noncentral cdf we have

$$T(x_i, R, n, m) = \frac{2^{1-\frac{n}{2}}}{\Gamma\left(\frac{n}{2}\right)} \int_0^{\infty} s^{n-1} e^{-\frac{x^2}{2}} \Phi\left(-\infty, \frac{sx_i}{\sqrt{n}} - m', R\right) ds$$

Source

```
cdfm.src
```

See Also

cdfMvte, cdfMvt2e, cdfMvnce

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts," *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.

2. Genz, A., "Numerical computation of multivariate normal probabilities," *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfMvt2e

Purpose

Computes multivariate Student's t cumulative distribution function with error management over [a,b].

Format

```
{y, err, retcode} = cdfMvt2e(ctl, a, b, R, m, n);
```

Input

ctl	instance of a **cdfmControl** structure with members.	
	ctl.maxEvaluations	scalar, maximum number of evaluations.
	ctl.absErrorTolerance	scalar absolute error tolerance.
	ctl.relErrorTolerance	tolerance.
a	NxK matrix, lower limits.	
b	NxK matrix, upper limits.	
R	KxK matrix, correlation matrix.	
m	Kx1 vector, noncentralities.	
n	scalar, degrees of freedom.	

Output

y	Lx1 vector, a $Pr(X > a \text{ and } X < b \mid r,m)$.	
err	Lx1 vector, estimates of absolute error.	
retcode	Lx1 vector, return codes.	
	0	normal completion with *err < ctl.absErrorTolerance*.
	1	*err > ctl.absErrorTolerance* and *ctl.maxEvaluations* exceeded; increase *ctl.maxEvaluations* to decrease error.

`2`	K > 100 or K < 1.
`3`	`R` not positive semi-definite.
`missing`	`R` not properly defined.

Remarks

The central multivariate Student's t cdf for the i-th row of `x` is defined by

$$T(x_i, R, n) = \frac{\Gamma\left(\frac{n+K}{2}\right)}{\Gamma\left(\frac{n}{2}\right)\sqrt{|R|\,(n\pi)^K}} \int_{x_{i1}}^{\infty}\int_{x_{i2}}^{\infty}\cdots\int_{x_{iK}}^{\infty}\left(1 + \frac{z'\Sigma^{-1}z}{n}\right)^{-\frac{n+K}{2}} dz$$

$$\equiv \frac{2^{1-\frac{n}{2}}}{\Gamma\left(\frac{n}{2}\right)} \int_0^{\infty} s^{n-1} e^{-\frac{x^2}{2}} \Phi\left(-\infty, \frac{sx_i}{\sqrt{n}}, R\right) ds$$

where

$$\Phi(x_i, R, m) = \frac{1}{\sqrt{|R|\,(2\pi)^m}} \int_{x_{i1}}^{\infty}\int_{x_{i2}}^{\infty}\cdots\int_{x_{iK}}^{\infty} e^{-\frac{1}{2}z'R^{-1}z} dz$$

For the noncentral cdf we have

$$T(x_i, R, n, m) = \frac{2^{1-\frac{n}{2}}}{\Gamma\left(\frac{n}{2}\right)} \int_0^{\infty} s^{n-1} e^{-\frac{x^2}{2}} \Phi\left(-\infty, \frac{sx_i}{\sqrt{n}} - m', R\right) ds$$

See Also

cdfMvte, cdfMvtce, cdfMvn2e

Source

`cdfm.src`

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts," *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.
2. Genz, A., "Numerical computation of multivariate normal probabilities," *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfN, cdfNc

Purpose

`cdfN` computes the cumulative distribution function (cdf) of the Normal distribution. **`cdfNc`** computes 1 minus the cdf of the Normal distribution.

Format

n = **`cdfN`**(*x*);
nc = **`cdfNc`**(*x*);

Input

x	NxK matrix.

Output

n	NxK matrix.
nc	NxK matrix.

Remarks

n is the integral from -∞ to *x* of the Normal density function, and *nc* is the integral from *x* to +∞.

Note that:

```
cdfN(x) + cdfNc(x) = 1
```

However, many applications expect **cdfN**(*x*) to approach 1, but never actually reach it. Because of this, we have capped the return value of **cdfN** at 1 - machine epsilon, or approximately 1 - 1.11e-16. As the relative error of **cdfN** is about ±5e-15 for **cdfN** (*x*) around 1, this does not invalidate the result. What it does mean is that for **abs**(*x*) > (approx.) 8.2924, the identity does not hold true. If you have a need for the uncapped value of **cdfN**, the following code will return it:

```
n = cdfN(x);
if n >= 1-eps;
   n = 1;
endif;
```

where the value of machine epsilon is obtained as follows:

```
x = 1;
do while 1-x /= 1;
   eps = x;
   x = x/2;
endo;
```

Note that this is an alternate definition of machine epsilon. Machine epsilon is usually defined as the smallest number such that 1 + machine epsilon > 1, which is about 2.23e-16. This defines machine epsilon as the smallest number such that 1 - machine epsilon < 1, or about 1.11e-16.

The **erf** and **erfc** functions are also provided, and may sometimes be more useful than **cdfN** and **cdfNc**.

Example

```
x = { -2 -1 0 1 2 };
n = cdfN(x);
nc = cdfNc(x);

x  = -2.0000000 -1.0000000 0.0000000 1.0000000 2.0000000
n  =  0.0227501 0.15865525 0.5000000 0.8413447 0.9772498
nc =  0.9772498 0.84134475 0.5000000 0.1586552 0.0227501
```

See Also

erf, erfc, cdfBeta, cdfChic, cdfTc, cdfFc, gamma

Technical Notes

For the integral from ∞ to x:

		x	<=	-37	**cdfN** underflows and 0.0 is returned
-36	<	x	<	-10	**cdfN** has a relative error of approx. ±5e-12

-10	<	x	<	0	**cdfN** has a relative error of approx. ±1e-13
0	<	x			**cdfN** has a relative error of approx. ±5e-15

For **cdfNc**, i.e., the integral from x to $+\infty$, use the above accuracies but change x to $-x$.

References

1. Adams, A.G. "Remark on Algorithm 304 Normal Curve Integral." *Comm. ACM*. Vol. 12, No. 10, Oct. 1969, 565-66.
2. Hill, I.D. and S.A. Joyce. "Algorithm 304 Normal Curve Integral." *Comm. ACM*. Vol. 10, No. 6, June 1967, 374-75.
3. Holmgren, B. "Remark on Algorithm 304 Normal Curve Integral." *Comm. ACM*. Vol. 13, No. 10, Oct. 1970.
4. Mardia, K.V. and P.J. Zemroch. *Tables of the F- and Related Distributions with Algorithms*. Academic Press, New York, 1978, ISBN 0-12-471140-5.

cdfNegBinomial

Purpose

Computes the cumulative distribution function for the negative binomial distribution.

Format

p = **cdfNegBinomial**(*f*,*s*,*prob*);

Input

f	NxK matrix, Nx1 vector or scalar. *0 < f*.
s	ExE conformable with *f*. *0 < s*.
prob	The probability of success on any given trial. ExE conformable with *f*. *0 < prob < 1*.

Output

p	NxK matrix, Nx1 vector or scalar. The probability of observing *f* failures before observing *s* s.

Remarks

For invalid inputs, **cdfNegBinomial** will return a scalar error code which, when its value is assessed by function **scalerr**, corresponds to the invalid input. If the first input is out of range, **scalerr** will return a 1; if the second is out of range, **scalerr** will return a 2; etc.

See Also

cdfBinomial, cdfBinomialInv, cdfNegBinomialInv

cdfNegBinomialInv

Purpose

Computes the quantile or inverse negative binomial cumulative distribution function.

Format

f = **cdfNegBinomialInv**(*p*, *s*, *prob*);

Input

p	NxK matrix, Nx1 vector or scalar. $0 < f < 1$.
s	ExE conformable with *p*. $0 < s$.
prob	The probability of success on any given trial. ExE conformable with *p*. $0 < prob < 1$.

Output

f	NxK matrix, Nx1 vector or scalar.

Remarks

For invalid inputs, **cdfNegBinomialInv** will return a scalar error code which, when its value is assessed by function **scalerr**, corresponds to the invalid input. If the first input is out of range, **scalerr** will return a 1; if the second is out of range, **scalerr** will return a 2; etc.

See Also

cdfBinomial, cdfBinomialInv, cdfNegBinomial

cdfN2

Purpose

Computes interval of Normal cumulative distribution function.

Format

y = **cdfN2**(*x*, *dx*);

Input

x	MxN matrix, abscissae.
dx	KxL matrix, ExE conformable to *x*, intervals.

Output

y	max(M,K) by max(N,L) matrix, the integral from *x* to *x*+ *dx* of the Normal distribution, i.e., $Pr(x < X < x + dx)$.

Remarks

The relative error is:

$\lvert x \rvert \leq 1$	and	$dx \leq 1$	$\pm 1e-14$
$1 < \lvert x \rvert < 37$	and	$\lvert dx \rvert < 1/\lvert x \rvert$	$\pm 1e-13$
$min(x, x+dx) > -37$	and	$y > 1e-300$	$\pm 1e-11$ or better

A relative error of ±1e-14 implies that the answer is accurate to better than ±1 in the 14th digit.

Example

```
print cdfN2(1,0.5);
```

```
9.1848052662599017e-02
```

```
print cdfN2(20,0.5);
```

```
2.7535164718736454e-89
```

```
print cdfN2(20,1e-2);
```

```
5.0038115018684521e-90
```

```
print cdfN2(-5,2);
```

```
1.3496113800582164e-03
```

```
print cdfN2(-5,0.15);
```

```
    3.3065580013000255e-07
```

Source

`lncdfn.src`

See Also

lncdfn2

cdfNi

Purpose

Computes the inverse of the cdf of the Normal distribution.

Format

x = **`cdfNi`**(*p*);

Input

p	NxK real matrix, Normal probability levels, `0 <= p <= 1`.

Output

x	NxK real matrix, Normal deviates, such that: `cdfN(x) = p.`

Remarks

cdfN(**cdfNi**(p)) = p to within the errors given below:

		p	<=	4.6e-308	-37.5 is returned
4.6e-308	<	p	<	5e-24	accurate to ±5 in 12th digit
5e-24	<	p	<	0.5	accurate to ±1 in 13th digit
0.5	<	p	<	1 - 2.22045e-16	accurate to ±5 in 15th digit
		p	≥	1 - 2.22045e-16	8.12589 is returned

See Also

cdfN

cdfPoisson

Purpose

Computes the Poisson cumulative distribution function.

Format

p = **cdfPoisson**(*x*, *lambda*);

Input

x	NxK matrix, Nx1 vector or scalar. *x* must be a positive whole number.
lambda	ExE conformable with *x*. The mean parameter.

Output

p	NxK matrix, Nx1 vector or scalar.

Remarks

For invalid inputs, **`cdfPoisson`** will return a scalar error code which, when its value is assessed by function **`scalerr`**, corresponds to the invalid input. If the first input is out of range, **`scalerr`** will return a 1; if the second is out of range, **`scalerr`** will return a 2; etc.

Example

Suppose that a hospital emergency department sees and average of 200 patients during the Friday evening shift. What is the probability that they will see fewer than 250 patients during any one Friday evening shift.

```
    p = cdfPoisson(250,200);
```

```
    p = 0.99971538 or 99.715%
```

See Also

cdfPoissonInv, cdfBinomial, cdfNegBinomial

cdfPoissonInv

Purpose

Computes the quantile or inverse Poisson cumulative distribution function.

Format

x = **cdfPoissonInv**(*p*, *lambda*);

Input

p	NxK matrix, Nx1 vector or scalar. *0 < p < 1*.
lambda	ExE conformable with *p*. The mean parameter.

Output

x	NxK matrix, Nx1 vector or scalar.

Example

Suppose that a hospital emergency department sees an average of 200 patients during the Friday evening shift. If the hospital wants to have enough staff on hand to handle the patient load on 95% of Friday evenings, how many patients do they need staff on hand for?

```
x = cdfPoissonInv(.95,200);
p = 224
```

The hospital should expect to see 224 or few patients on 95% of Friday evenings.

Remarks

For invalid inputs, **cdfPoissoninv** will return a scalar error code which, when its value is assessed by function **scalerr**, corresponds to the invalid input. If the first input is out of range, **scalerr** will return a 1; if the second is out of range,

scalerr will return a 2; etc.

See Also

cdfPoisson, cdfBinomial, cdfNegBinomial,

cdfRayleigh

Purpose

Computes the Rayleigh cumulative distribution function.

Format

y = **cdfRayleigh**(*x*,*b*);

Input

x	NxK matrix, an Nx1 vector or scalar. *x* must be greater than 0.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The Rayleigh cumulative distribution function is defined as

$$1 - exp\left(\frac{-x^2}{2\sigma^2}\right)$$

See Also

cdfRayleighInv, pdfRayleigh

cdfRayleighInv

Purpose

Computes the Rayleigh inverse cumulative distribution function.

Format

y = **cdfRayleighInv**(*p*,*b*);

Input

p	NxK matrix, Nx1 vector or scalar. *p* must be greater than 0 and less than 1.
b	Shape parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *p*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

See Also

pdfRayleigh, cdfRayleigh

cdfTc

Purpose

Computes the complement of the cdf of the Student's *t* distribution.

Format

```
y = cdfTc(x,  n);
```

Input

x	NxK matrix.
n	LxM matrix, ExE conformable with *x*.

Output

y	max(N,L) by max(K,M) matrix.

Remarks

y is the integral from *x* to ∞ of the *t* distribution with *n* degrees of freedom.

Allowable ranges for the arguments are:

$$-\infty \leq x \leq +\infty$$
$$n > 0$$

A -1 is returned for those elements with invalid inputs.

This equals:

$$1 - F(x, n)$$

where F is the t cdf with n degrees of freedom. Thus, to get the t cdf, subtract **cdfTc**(x, n) from 1. The complement of the cdf is computed because this is what is most commonly needed in statistical applications, and because it can be computed with fewer problems of roundoff error.

Example

```
    x = { .1, .2, .3, .4 };
    n = 3;
    y = cdfTc(x,n);

        0.46332617
        0.42713516
    y = 0.39188165
        0.35796758
```

See Also

cdfTci

Technical Notes

For results greater than 0.5e-30, the absolute error is approx. ±1e-14 and the relative error is approx. ±1e-12. If you multiply the relative error by the result, then take the minimum of that and the absolute error, you have the maximum actual error for any result. Thus, the actual error is approx. ±1e-14 for results greater than 0.01. For results less than 0.01, the actual error will be less. For example, for a result of 0.5e-30, the actual error is only ±0.5e-42.

References

1. Abramowitz, M. and I.A. Stegun, eds. *Handbook of Mathematical Functions*. 7th ed. Dover, New York, 1970. ISBN 0-486-61272-4.
2. Hill, G.W. "Algorithm 395 Student's t-Distribution." **Comm. ACM.** Vol. 13, No. 10, Oct. 1970.
3. Hill, G.W. "Reference Table: Student's t-Distribution Quantiles to 20D." *Division of Mathematical Statistics Technical Paper No. 35*. Commonwealth Scientific and Industrial Research Organization, Australia, 1972.

cdfTci

Purpose

Computes the inverse of the complement of the Student's *t* cdf.

Format

x = **cdfTci**(*p*, *n*);

Input

p	NxK real matrix, complementary Student's *t* probability levels, *0 <= p <= 1*.
n	LxM real matrix, degrees of freedom, *n > 1*, *n* need not be integral. ExE conformable with *p*.

Output

x	max(N,L) by max(K,M) real matrix, Student's *t* deviates, such that **cdfTc**(*x*, *n*) = *p*.

Remarks

cdfTc(**cdfTci**(*p*, *n*)) = *p* to within the errors given below:

0.5e-30	<	*p*	<	0.01	accurate to ±1 in 12th digit
0.01	<	*p*			accurate to ±1e-14

Extreme values of arguments can give rise to underflows, but no overflows are generated.

See Also

cdfTc

cdfTnc

Purpose

The integral under noncentral Student's *t* distribution, from -∞ to *x*. It can return a vector of values, but the degrees of freedom and noncentrality parameter must be the same for all values of *x*.

Format

```
y = cdfTnc(x, v, d);
```

Input

x	Nx1 vector, values of upper limits of integrals.
v	scalar, degrees of freedom, v > 0.
d	scalar, noncentrality parameter.
	This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. (See Scheffe, *The Analysis of Variance*, App. IV, 1959.)

Output

y	Nx1 vector, integrals from -∞ to *x* of noncentral *t*.

Remarks

cdfTc(*x*, *v*) = 1 - **cdfTnc**(*x*, *v*,0).

See Also

cdfFnc, cdfChinc

cdfTvn

Purpose

Computes the cumulative distribution function of the standardized trivariate Normal density (lower tail).

Format

```
c = cdfTvn(x1, x2, x3, rho12, rho23, rho13);
```

Input

x1	Nx1 vector of upper limits of integration for variable 1.
x2	Nx1 vector of upper limits of integration for variable 2.
x3	Nx1 vector of upper limits of integration for variable 3.
rho12	scalar or Nx1 vector of correlation coefficients between the two variables *x1* and *x2*.
rho23	scalar or Nx1 vector of correlation coefficients between the two variables *x2* and *x3*.
rho13	scalar or Nx1 vector of correlation coefficients between the two variables *x1* and *x3*.

Output

c	Nx1 vector containing the result of the triple integral from -∞ to *x1*, -∞ to *x2*, and -∞ to *x3* of the standardized trivariate Normal density.

Remarks

Allowable ranges for the arguments are:

$$-\infty > x1 > +\infty$$

$$-\infty > x2 > +\infty$$

$$-\infty > x3 > +\infty$$

$$-1 < rho12 < 1$$

$$-1 < rho23 < 1$$

$$-1 < rho14 < 1$$

In addition, `rho12`, `rho23` and `rho13` must come from a legitimate positive definite matrix. A -1 is returned for those rows with invalid inputs.

A separate integral is computed for each row of the inputs.

The first 3 arguments (`x1`, `x2`, `x3`) must be the same length, N. The second 3 arguments (`rho12`, `rho23`, `rho13`) must also be the same length, and this length must be N or 1. If it is 1, then these values will be expanded to apply to all values of `x1`, `x2`, `x3`. All inputs must be column vectors.

To find the integral under a general trivariate density, with `x1`, `x2`, and `x3` having nonzero means and any positive standard deviations, transform by subtracting the mean and dividing by the standard deviation. For example:

$$x1 = (\ x1 - meanc(x1)\ )\ /\ stdc(x1)$$

The absolute error for **`cdfTvn`** is approximately ±2.5e-8 for the entire range of arguments.

See Also

cdfN, cdfBvn

References

1. Daley, D.J. "Computation of Bi- and Tri-variate Normal Integral." *Appl. Statist.* Vol. 23,

No. 3, 1974, 435-38.

2. Steck, G.P. "A Table for Computing Trivariate Normal Probabilities." *Ann. Math. Statist.* Vol. 29, 780-800.

cdfWeibull

Purpose

Computes the cumulative distribution function for the Weibull distribution.

Format

```
y = cdfWeibull(x,k,lambda);
```

Input

x	NxK matrix, Nx1 vector or scalar. *x* must be greater than 0.
k	Shape parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *k* must be greater than 0.
lambda	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *lambda* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The Weibull cumulative distribution function is defined as:

```
f(x;k,λ) = 1 - e^-(x/λ)k
```

See Also

pdfWeibull, cdfWeibullInv

cdfWeibullInv

Purpose

Computes the Weibull inverse cumulative distribution function.

Format

```
y = cdfWeibullInv(p,k,lambda);
```

Input

p	NxK matrix, Nx1 vector or scalar. *p* must be greater than 0 and less than 1.
k	Shape parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *k* must be greater than 0.
lambda	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *lambda* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

See Also

pdfWeibull, cdfWeibull

cdir

Purpose

Returns the current directory.

Format

y = **`cdir`**(*s*);

Input

s	string, if the first character is 'A'-'Z' and the second character is a colon ':' then that drive will be used. If not, the current default drive will be used.

Output

y	string containing the drive and full path name of the current directory on the specified drive.

Remarks

If the current directory is the root directory, the returned string will end with a backslash, otherwise it will not.

A null string or scalar zero can be passed in as an argument to obtain the current drive and path name.

Example

If the current working directory is C:\gauss12:

```
x = cdir(0);
y = cdir("d:");
print x;
print y;
```

The code above will return:

```
C:\gauss12
d:
```

ceil

Purpose

Round up toward +∞.

Format

y = **ceil**(*x*);

Input

x	NxK matrix.

Output

y	NxK matrix.

Remarks

This rounds every element in the matrix *x* to an integer. The elements are rounded up toward +∞.

Example

```
x = 10*rndn(2,2);
y = ceil(x);
```

After the code above, the matrices *x* and *y* should hold values similar to below. Answers will vary due to the use of random numbers as the input to the **ceil** function.

```
x = 8.73383  -0.783488  y = 9.0000000  0.0000000
    13.1106   7.155113      14.000000  8.0000000
```

See Also

floor, trunc

C

ChangeDir

Purpose

Changes the working directory within a program.

Format

```
        d = ChangeDir(s);
```

Input

s	string, directory to change to.

Output

d	string, new working directory, or null string if change failed.

See Also

chdir, cdir

chdir

Purpose

Changes working directory in interactive mode..

Format

```
    chdir dirstr;
```

Input

dirstr	literal or ^string, directory to change to.

Remarks

This is for interactive use. Use **`ChangeDir`** in a program.

If the directory change fails, **`chdir`** prints an error message.

The working directory is listed in the status report on UNIX.

See Also

changedir, cdir

chiBarSquare

Purpose

Compute compute the probability for a chi-bar square statistic from an hypothesis involving parameters under constraints.

Format

```
SLprob = chiBarSquare(SL, H, a, b, c, d, bounds);
```

Input

SL	scalar, chi-bar square statistic
H	KxK matrix, positive covariance matrix
a	MxK matrix, linear equality constraint coefficients
b	Mx1 vector, linear equality constraint constants
	These arguments specify the linear equality constraints of the

following type:

```
a * X = b
```

where x is the Kx1 parameter vector.

c	MxK matrix, linear inequality constraint coefficients.
d	Mx1 vector, linear inequality constraint constants.

These arguments specify the linear inequality constraints of the following type:

```
c * X >= d
```

where x is the Kx1 parameter vector.

bounds	Kx2 matrix, bounds on parameters. The first column contains the lower bounds, and the second column the upper bounds.

Output

SLprob	scalar, probability of *SL*.

Remarks

See Silvapulle and Sen, *Constrained Statistical Inference*, page 75 for further details about this function. Let

$$Z_{px1} \quad N(0,V)$$

where V is a positive definite covariance matrix. Define

$$x^{-2}\left(V, C\right) = Z'V^{-1}Z - \min_{\theta\epsilon C}\ (Z-\theta)'\ V^{-1}(Z-\theta)$$

C is a closed convex cone describing a set of constraints. **ChiBarSquare** computes the probability of this statistic given *V* and *C*.

Example

```
V = { 0.0005255598 -0.0006871606 -0.0003191342,
     -0.0006871606 0.0037466205 0.0012285813,
     -0.0003191342 0.0012285813 0.0009081412 };

SL = 3.860509;

Bounds = { 0 200, 0 200, 0 200 };

vi = invpd(v);

SLprob = chiBarSquare(SL,Vi,0,0,0,0,bounds);

slprob = 0.10885000
```

Source

`hypotest.src`

chol

Purpose

Computes the Cholesky decomposition of a symmetric, positive definite square matrix.

Format

```
y = chol(x);
```

Input

x	NxN matrix.

Output

y	NxN matrix containing the Cholesky decomposition of *x*.

Remarks

y is the "square root" matrix of *x*. That is, it is an upper triangular matrix such that $x = y'y$.

`chol` does not check to see that the matrix is symmetric. **`chol`** will look only at the upper half of the matrix including the principal diagonal.

If the matrix *x* is symmetric but not positive definite, either an error message or an error code will be generated, depending on the lowest order bit of the trap flag:

trap 0	Print error message and terminate program.
trap 1	Return scalar error code 10.

See **`scalerr`** and `trap` for more details about error codes.

Example

```
    //'moment' calculates x'*x with options for handling missing data
```

```
x = moment(rndn(100,4),0);
y = chol(x);

//y'y is equivalent to y'*y
ypy = y'y;

    95.2801   8.6983   3.7248    1.5449       9.7612   0.8911   0.3816   0.1583
x =  8.6983  83.4547  -6.1455  -12.5551   y = 0.0000   9.0918  -0.7133  -1.3964
     3.7248  -6.1455  87.6666   -3.0284       0.0000   0.0000   9.3280  -0.4379
     1.5449 -12.5551  -3.0284   90.8311       0.0000   0.0000   0.0000   9.4162

      95.2801   8.6983   3.7248   1.5449
ypy = 8.6983  83.4547  -6.1455 -12.5551
      3.7248  -6.1455  87.6666  -3.0284
      1.5449 -12.5551  -3.0284  90.8311
```

See Also

crout, solpd

choldn

Purpose

Performs a Cholesky downdate of one or more rows on an upper triangular matrix.

Format

r = **choldn**(*C*, *x*);

Input

C	KxK upper triangular matrix.

c

x	NxK matrix, the rows to downdate *C* with.

Output

r	KxK upper triangular matrix, the downdated matrix.

Remarks

If **trap 1** is set, **choldn** returns scalar error code 60, otherwise it terminates the program with an error message.

C should be a Cholesky factorization.

```
    choldn(C, x);
```

is equivalent to

```
    chol(C'C - x'x);
```

but **choldn** is numerically much more stable.

WARNING: it is possible to render a Cholesky factorization non-positive definite with **choldn**. You should keep an eye on the ratio of the largest diagonal element of *r* to the smallest--if it gets very large, *r* may no longer be positive definite. This ratio is a rough estimate of the condition number of the matrix.

Example

```
    let C[3,3] = 20.16210005 16.50544413 9.86676135
                 0 11.16601462 2.97761666
                 0 0 11.65496052;
    let x[2,3] = 1.76644971 7.49445820 9.79114666
                 6.87691156 4.41961438 4.32476921;
    r = choldn(C,x);

        18.8706  15.3229   8.0495
    r =  0.0000   9.3068  -2.1201
         0.0000   0.0000   7.6288
```

See Also

cholup, chol

cholsol

Purpose

Solves a system of linear equations given the Cholesky factorization of the system.

Format

x = **cholsol**(*b*, *C*);

Input

b	NxK matrix.
C	NxN matrix.

Output

x	NxK matrix.

Remarks

C is the Cholesky factorization of a linear system of equations *A*. *x* is the solution for *Ax* = *b*. *b* can have more than one column. If so, the system is solved for each column, i.e., *A***x*[., *i*] = *b*[., *i*].

Since $A^{-1} = I/A$ and **eye**(*N*) creates an identity matrix of size *N*:

```
cholsol(eye(N), C);
```

is equivalent to:

```
invpd(A);
```

Thus, if you have the Cholesky factorization of *A*, **cholsol** is the most efficient way to obtain the inverse of *A*.

Example

```
//Assign the right-hand side 'b' and the Cholesky
//factorization 'C'
b = { 0.03177513, 0.41823100, 1.70129375 };
C = { 1.73351215 1.53201723 1.78102499,
               0 1.09926365 0.63230050,
               0          0 0.67015361 };

//Solve the system of equations
x = cholsol(b,C);

//Note: C'C is equivalent to C'*C
A = C'C;

//Solve the system of equations
x2 = b/A;
```

```
         -1.9440         -1.9440
     x = -1.5269   x2 = -1.5269
          3.2158          3.2158
```

See Also

chol

cholup

Purpose

Performs a Cholesky update of one or more rows on an upper triangular matrix.

Format

r = **cholup**(*C*, *x*);

Input

C	KxK upper triangular matrix.
x	NxK matrix, the rows to update *C* with.

Output

r	KxK upper triangular matrix, the updated matrix.

Remarks

C should be a Cholesky factorization.

cholup(*C*, *x*) is equivalent to **chol**($C'C + x'x$), but **cholup** is numerically much more stable.

Example

```
    let C[3,3] = 18.87055964 15.3229443  8.04947012
                           0 9.30682813 -2.12009339
                           0          0  7.62878355;
    let x[2,3] = 1.76644971 7.49445820 9.79114666
                 6.87691156 4.41961438 4.32476921;
    r = cholup(C,x);

        20.162100    16.505444    9.8667614
    r = 0.0000000    11.166015    2.9776167
        0.0000000    0.0000000    11.654961
```

See Also

choldn

chrs

Purpose

Converts a matrix of ASCII values into a string containing the appropriate characters.

Format

y = **chrs**(x);

Input

x	NxK matrix.

Output

y	string of length N*K containing the characters whose ASCII values are equal to the values in the elements of x.

Remarks

This function is useful for embedding control codes in strings and for creating variable length strings when formatting printouts, reports, etc.

Example

```
//42 is the ascii value for an asterisk '*'
print chrs(42);
```

The code above returns:

```
*
```

chrs can be used to create an interactive program in which the user is prompted for keyboard input which the code uses to make decisions.

```
//Print a string to prompt the user for input
print "Choose a parameter: Enter [a,b,c]";
```

```
//Wait for the user to enter a keystroke and assign the
//ASCII value of that key to 'param'
param = keyw;

//Convert the ASCII value to a string
paramString = chrs(param);

if paramString == "a";
   print "You have chosen:" "a";
   //execute code for this choice
elseif paramString == "b";
   print "You have chosen:" "b";
   //execute code for this choice
elseif paramString == "c";
   print "You have chosen:" "c";
   //execute code for this choice
endif;
```

See Also

vals, ftos, stof

clear

Purpose

Clears space in memory by setting matrices equal to scalar zero.

Format

`clear` *x*, *y*;

Remarks

If your program is running out of memory, or uses considerable system resources, using `clear` to deallocate large matrices after they are no longer needed may allow it to run more efficiently.

```
    clear x;
```

is equivalent to

```
    x = 0;
```

Matrix names are retained in the symbol table after they are cleared.

Matrices can be `clear`'ed even though they have not previously been defined. `clear` can be used to initialize matrices to scalar 0.

Example

```
    A = rndn(1000, 1000);
    //Code that uses 'A' would be here
    //Free memory holding 'A'
    clear A;
```

See Also

clearg, new, show, delete

clearg

Purpose

Clears global symbols by setting them equal to scalar zero.

Format

```
clearg a, b, c;
```

Output

a,b,c	scalar global matrices containing 0.

Remarks

It is considered a best practice to avoid using global variables inside of procedures when possible.

```
clearg x;
```

is equivalent to

```
x = 0;
```

where *x* is understood to be a global symbol. `clearg` can be used to initialize symbols not previously referenced. This command can be used inside of procedures to clear global matrices. It will ignore any locals by the same name.

Example

Let us suppose there is a procedure that takes in a large global matrix, but only uses the LU factorization for the majority of the calculation. If the computer is memory limited compared to the size of the data, you could do something like this:

```
//Create a 1000x1000 matrix of Cauchy random deviates
X = rndCauchy(1000, 1000, 0, 1);

//Call the procedure which is defined below
out = myProc(X);
```

```
proc (1) = myProc(A);
    local l, u, ans;

    //Calculate LU factors of 'A'
    { l, u } = lu(A);

    //Code no longer needs 'A', or global 'x', so free them
    clearg X;
    clear A;

    //Main work of proc would go here, including assignment
    //of 'ans'
retp(ans);
endp;
```

See Also

clear, delete, new, show, local

close

Purpose

Closes a **GAUSS** file.

Format

y = **close**(*handle*);

Input

handle	scalar, the file handle given to the file when it was opened with the `open`, `create`, or **`fopen`** command.

Output

y	scalar, 0 if successful, -1 if unsuccessful.

Remarks

handle is the scalar file handle created when the file was opened. It will contain an integer which can be used to refer to the file.

`close` will close the file specified by handle, and will return a 0 if successful and a -1 if not successful. The handle itself is not affected by **`close`** unless the return value of **`close`** is assigned to it.

If *f1* is a file handle and it contains the value 7, then after:

```
call close(f1);
```

the file will be closed but *f1* will still have the value 7. The best procedure is to do the following:

```
f1 = close(f1);
```

This will set *f1* to 0 upon a successful close.

It is important to set unused file handles to zero because both `open` and `create` check the value that is in a file handle before they proceed with the process of opening a file. During `open` or `create`, if the value that is in the file handle matches that of

c

an already open file, the process will be aborted and a File already open error message will be given. This gives you some protection against opening a second file with the same handle as a currently open file. If this happened, you would no longer be able to access the first file.

An advantage of the **close** function is that it returns a result which can be tested to see if there were problems in closing a file. The most common reason for having a problem in closing a file is that the disk on which the file is located is no longer in the disk drive--or the handle was invalid. In both of these cases, **close** will return a -1.

Files are not automatically closed when a program terminates. This allows users to run a program that opens files, and then access the files from interactive mode after the program has been run. Files are automatically closed when **GAUSS** exits to the operating system or when a program is terminated with the end statement. stop will terminate a program but not close files.

As a rule it is good practice to make end the last statement in a program, unless further access to the open files is desired from interactive mode. You should close files as soon as you are done writing to them to protect against data loss in the case of abnormal termination of the program due to a power or equipment failure.

The danger in not closing files is that anything written to the files may be lost. The disk directory will not reflect changes in the size of a file until the file is closed and system buffers may not be flushed.

Example

```
    open f1 = dat1 for append;
    y = writer(f1,x);
    f1 = close(f1);
```

See Also

closeall

closeall

Purpose

Closes all currently open **GAUSS** files.

Format

```
closeall;
closeall list_of_handles;
```

Remarks

`list_of_handles` is a comma-delimited list of file handles.

`closeall` with no specified list of handles will close all files. The file handles will not be affected. The main advantage of using `closeall` is ease of use; the file handles do not have to be specified, and one statement will close all files.

When a list of handles follows `closeall`, all files are closed and the file handles listed are set to scalar 0. This is safer than `closeall` without a list of handles because the handles are cleared.

It is important to set unused file handles to zero because both `open` and `create` check the value that is in a file handle before they proceed with the process of opening a file. During `open` or `create`, if the value that is in the file handle matches that of an already open file, the process will be aborted and a File already open error message will be given. This gives you some protection against opening a second file with the same handle as a currently open file. If this happened, you would no longer be able to access the first file.

Files are not automatically closed when a program terminates. This allows users to run a program that opens files, and then access the files from interactive mode after the program has been run. Files are automatically closed when **GAUSS** exits to the operating system or when a program is terminated with the `end` statement. `stop` will terminate a program but not close files.

As a rule it is good practice to make `end` the last statement in a program, unless further access to the open files is desired from interactive mode. You should close files as soon as you are done writing to them to protect against data loss in the case of abnormal termination of the program due to a power or equipment failure.

The danger in not closing files is that anything written to the files may be lost. The disk directory will not reflect changes in the size of a file until the file is closed and system buffers may not be flushed.

Example

```
    open f1 = dat1 for read;
    open f2 = dat1 for update;
    x = readr(f1,rowsf(f1));
    x = sqrt(x);
    call writer(f2,x);
    closeall f1,f2;
```

See Also

close, open

cls

Purpose

Clears the program input/output window.

Format

```
cls;
```

Remarks

This command clears the window and locates the cursor at the upper left hand corner of the window. It is sometimes useful to put a `cls` statement at the beginning of a program that prints a report to the screen so that you have fewer lines of data to look at.

See Also

locate

code

Purpose

Allows a new variable to be created (coded) with different values depending upon which one of a set of logical expressions is true.

Format

y = **code**(*e*, *v*);

Input

e	NxK matrix of 1's and 0's. Each column of this matrix is created by a logical expression using "dot" conditional and boolean operators. Each of these expressions should return a

	column vector result. The columns are horizontally concatenated to produce *e*. If more than one of these vectors contains a 1 in any given row, the **code** function will terminate with an error message.
v	(K+1)x1 vector containing the values to be assigned to the new variable.

Output

y	Nx1 vector containing the new values.

Remarks

If none of the K expressions is true, the new variable is assigned the default value, which is given by the last element of *v*.

Example

```
let x1 = 0 /* column vector of original values */
         5
        10
        15
        20;

let v = 1 /* column vector of new values */
        2
        3; /* the last element of v is the "default" */

e1 =(0 .lt x1) .and (x1 .le 5); /* expression 1 */
e2 =(5 .lt x1) .and (x1 .le 25); /* expression 2 */
```

```
e = e1~e2; /* concatenate e1 & e2 to make a 1,0 mask with
              one less column than the number of new values
              in v. */

y = code(e,v);
```

After the code above:

```
x1 = 0     v = 1     e =  0  0     y = 3
     5         2          1  0         1
    10         3          0  1         2
    15                    0  1         2
    20                    0  1         2
```

For every row in e, if a 1 is in the first column, the first element of v is used. If a 1 is in the second column, the second element of v is used, and so on. If there are only zeros in the row, the last element of v is used. This is the default value.

If there is more than one 1 in any row of e, the function will terminate with an error message.

Source

datatran.src

See Also

recode, substute

code (dataloop)

Purpose

Creates new variables with different values based on a set of logical expressions.

Format

```
code [[#]] [[$]]  var  [[default defval]] with
 val_1 for expression_1,
 val_2 for expression_2,
.
.
.
val_n  for  expression_n;
```

Input

var	literal, the new variable name.
defval	scalar, the default value if none of the expressions are TRUE.
val	scalar, value to be used if corresponding expression is TRUE.
expression	logical scalar-returning expression that returns nonzero TRUE or zero FALSE.

Remarks

If '$' is specified, the new variable will be considered a character variable. If '#' or nothing is specified, the new variable will be considered numeric.

The logical expressions must be mutually exclusive, i.e., only one may return TRUE for a given row (observation).

Any variables referenced must already exist, either as elements of the source data set, as externs, or as the result of a previous `make`, `vector`, or `code` statement.

If no default value is specified, 999 is used.

Example

```
code agecat default 5 with
 1 for age < 21,
 2 for age >= 21 and age < 35,
 3 for age >= 35 and age < 50,
 4 for age >= 50 and age < 65;
```

```
code $ sex with
"MALE" for gender == 1,
"FEMALE" for gender == 0;
```

See Also

recode (dataloop)

cols

Purpose

Returns the number of columns in a matrix.

Format

y = **cols**(*x*);

Input

x	NxK matrix or sparse matrix.

Output

y	number of columns in *x*.

Remarks

If *x* is an empty matrix, **rows**(*x*) and **cols**(*x*) both return 0.

Example

```
//Create a 100x3 matrix of uniform random numbers
x = rndu(100,3);

y = cols(x);
```

After the code above:

```
y = 3
```

See Also

rows, colsf, show

colsf

Purpose

Returns the number of columns in a **GAUSS** data (`.dat`) file or **GAUSS** matrix (`.fmt`) file.

Format

yf = **colsf**(*fh*);

Input

fh	file handle of an open file.

Output

yf	number of columns in the file that has the handle *fh*.

Remarks

In order to call **colsf** on a file, the file must be open.

Example

```
//Create a file with 10 columns
create fp = myfile with x,10,4;

//Calculate the number of rows of the file created above
```

```
    nCols = colsf(fp);
```

The result will be

```
    nCols = 10
```

See Also

rowsf, cols, show

combinate

Purpose

Computes combinations of *N* things taken *K* at a time.

Format

y = **combinate**(*N*, *K*);

Input

N	scalar.
K	scalar.

Output

y	MxK matrix, where M is the number of combinations of *N* things taken *K* at a time.

Remarks

"Things" are represented by a sequence of integers from 1 to *N*, and the integers in each row of *y* are the combinations of those integers taken *K* at a time.

Example

```
    //Calculate all combinations of 4 items chosen 2 at a time
    n = 4;
    k = 2;
    y = combinate(n,k);

    print y;
```

The code above will create the following output:

```
    1.0000 2.0000
    1.0000 3.0000
    1.0000 4.0000
    2.0000 3.0000
    2.0000 4.0000
    3.0000 4.0000
```

See Also

combinated, numCombinations

combinated

Purpose

Writes combinations of *N* things taken *K* at a time to a **GAUSS** data set.

Format

```
ret = combinated(fname, vnames, N, K);
```

Input

fname	string, file name.
vname	1x1 or Kx1 string array, names of columns in data set. If 1x1 string, names will have column number appended. If null string, names will be X1, X2, ...
N	scalar.
K	scalar.

Output

ret	scalar, if data set was successfully written, *ret* = number of rows written to data set. Otherwise, one of the following:

0	file already exists.
-1	data set couldn't be created.
-n	the (*n*-1)th write to the data set failed.

Remarks

The rows of the data set in *fname* contain sequences of the integers from 1 to *N* in combinations taken *K* at a time.

Example

```
//Note: The '$|' operator vertically concatenates strings
vnames = "Jim"$|"Harry"$|"Susan"$|"Wendy";

//Create a dataset file named 'couples', containing all
//combinations of the names in 'vnames' taken 2 at a time
k = 2;
m = combinated("couples",vnames, rows(vnames),k);

print m "rows were written to the dataset";
```

```
6.0000 rows were written to the dataset
```

Continuing from the code above:

```
//Open the file written above
open f0 = "couples";

//Read in m=6 rows of the dataset into 'y'
y = readr(f0,m);

//Get the variable names from the dataset and assign them
//to 'names'
names = getnamef(f0);
f0=close(f0);

for i(1, rows(y),1);
  print names[y[i,.]]';
endfor;
```

will produce the following output:

```
          1  2                       Jim Harry
          1  3                       Jim Susan
      y = 1  4   print output =  Jim Wendy
          2  3                       Harry Susan
          2  4                       Harry Wendy
          3  4                       Susan Wendy
```

The first row of the print output 'Jim Harry' is the first and second element of *vnames*, because the first row of *y* is equal to '1 2'. The fourth row of the `print` output is 'Harry Susan', because the fourth row of y is '2 3' and 'Harry' is the second element of *vnames* while 'Susan' is the third element.

See Also

combinate, numCombinations

comlog

Purpose

Controls logging of interactive mode commands to a disk file.

Format

`comlog [[file=`*`filename`*`]] [[on|off|reset]];`

Input

filename	literal or ^string.

Remarks

`comlog` on turns on command logging to the current file. If the file already exists, subsequent commands will be appended.

`comlog` off closes the log file and turns off command logging.

`comlog` reset turns on command logging to the current log file, resetting the log file by deleting any previous commands.

Interactive mode statements are always logged into the file specified in the *`log_file`* configuration variable, regardless of the state of `comlog`.

The command `comlog`*`file= filename`* selects the file but does not turn on logging.

The command `comlog` off will turn off logging. The filename will remain the same. A subsequent `comlog` on will cause logging to resume. A subsequent `comlog` reset will cause the existing contents of the log file to be destroyed and a new file created.

The command `comlog` by itself will cause the name and status of the current log file to be printed in the window.

compile

Purpose

Compiles a source file to a compiled code file. See also **Compiler**, Chapter 1.

Format

```
compile source fname;
```

Input

source	literal or ^string, the name of the file to be compiled.
fname	literal or ^string, optional, the name of the file to be created. If not given, the file will have the same filename and path as *source*. It will have a `.gcg` extension.

Remarks

The *source* file will be searched for in the *src_path* if the full path is not specified and it is not present in the current directory.

The *source* file is a regular text file containing a **GAUSS** program. There can be references to global symbols, **Run-Time Library** references, etc.

If there are `library` statements in *source*, they will be used during the compilation to locate various procedures and symbols used in the program. Since all of these library references are resolved at compile time, the `library` statements are not transferred to the compiled file. The compiled file can be run without activating any libraries.

If you do not want extraneous stuff saved in the compiled image, put a `new` at the top of the *source* file or execute a `new` in interactive mode before compiling.

The program saved in the compiled file can be run with the `run` command. If no extension is given, the `run` command will look for a file with the correct extension for the version of **GAUSS**. The *src_path* will be used to locate the file if the full path name is not given and it is not located on the current directory.

When the compiled file is `run`, all previous symbols and procedures are deleted before the program is loaded. It is therefore unnecessary to execute a `new` before `run`'ning a compiled file.

If you want line number records in the compiled file you can put a `#lineson` statement in the *source* file or turn line tracking on from the Options menu.

Don't try to include compiled files with `#include`.

Example

```
    compile qxy.e;
```

In this example, the *src_path* would be searched for qxy.e, which would be compiled to a file called `qxy.gcg` on the same subdirectory *qxy.e* was found.

```
    compile qxy.e xy;
```

In this example, the *src_path* would be searched for `qxy.e` which would be compiled to a file called `xy.gcg` on the current subdirectory.

See Also

run, use, saveall

complex

Purpose

Converts a pair of real matrices to a complex matrix.

Format

z = **complex**(*xr*, *xi*);

Input

xr	NxK real matrix, the real elements of *z*.
xi	NxK real matrix or scalar, the imaginary elements of *z*.

Output

z	NxK complex matrix.

Example

```
x = { 4  6,
      9  8 };

y = { 3  5,
      1  7 };

t = complex(x,y);
```

After the code above, *t* will be equal to:

```
4 + 3i  6 + 5i
9 + 1i  8 + 7i
```

See Also

imag, real

con

Purpose

Requests input from the keyboard (console), and returns it in a matrix.

Format

x = **con**(*r*, *c*);

Input

r	scalar, row dimension of matrix.
c	scalar, column dimension of matrix.

Output

x	*r* x *c* matrix.

Remarks

con gets input from the active window. **GAUSS** will not "see" any input until you press ENTER, so follow each entry with an ENTER.

r and *c* may be any scalar-valued expressions. Nonintegers will be truncated to an integer.

If *r* and *c* are both set to 1, **con** will cause a question mark to appear in the window, indicating that it is waiting for a scalar input.

Otherwise, **con** will cause the following prompt to appear in the window:

```
            - [1,1]
```

indicating that it is waiting for the [1,1] element of the matrix to be inputted. The **-** means that **con** will move horizontally through the matrix as you input the matrix elements. To change this or other options, or to move to another part of the matrix, use the following commands:

u	up one row	**U**	first row
d	down one row	**D**	last row
l	left one column	**L**	first column
r	right one column	**R**	last column
t	first element		
b	last element		
g #, #	goto element		
g #	goto element of vector		
h	move horizontally, default		
v	move vertically, default		
exttt\	move diagonally, default		
s	show size of matrix		
n	display element as numeric,		

	default
c	display element as character
e	**exp**(1)
p	pi
.	missing value
?	show help screen
x	exit

If the desired matrix is 1xN or Nx1, then **con** will automatically exit after the last element has been entered, allowing you to input the vector quickly.

If the desired matrix is NxK, you will need to type '*x*' to exit when you have finished entering the matrix data. If you exit before all elements have been entered, unspecified elements will be zeroed out.

Use a leading single quote for character input.

Example

```
     n = con(1,1);
     print rndn(n,n);
```

If you enter 2 at the **con** generated prompt:

```
      ? 2
```

the code above will return a 2x2 random matrix, similar to:

```
    -1.2505596        1.6322417
    -1.0894098       0.74763307
```

In this example, the **con** function is used to obtain the size of a square matrix of Normal random variables which is to be printed out.

See Also

cons, let, load

cond

Purpose

Computes the condition number of a matrix using the singular value decomposition.

Format

c = **cond**(*x*);

Input

x	NxK matrix.

Output

c	scalar, an estimate of the condition number of *x*. This equals the ratio of the largest singular value to the smallest. If the smallest singular value is zero or not all of the singular values

can be computed, the return value is 10^{300}.

Example

```
x = { 4 2 6,
      8 5 7,
      3 8 9 };

y = cond(x);
```

will assign *y* to equal:

```
y = 9.8436943
```

Source

`svd.src`

conj

Purpose

Returns the complex conjugate of a matrix.

Format

y = **conj**(*x*);

Input

x	NxK matrix.

Output

y	NxK matrix, the complex conjugate of *x*.

Remarks

Compare **conj** with the transpose (′) operator.

Example

```
    x = { 1+9i   2,
          4+4i   5i,
            7i 8-2i };
    y = conj(x);
```

```
        1 + 9i 2                1 - 9i 2
    x = 4 + 4i 0 + 5i     y = 4 - 4i 0 - 5i
        0 + 7i 8 - 2i           0 - 7i 8 + 2i
```

cons

Purpose

Retrieves a character string from the keyboard.

Format

x = **cons**();

Output

x	string, the characters entered from the keyboard

Remarks

x is assigned the value of a character string typed in at the keyboard. The program will pause to accept keyboard input. The maximum length of the string that can be entered is 254 characters. The program will resume execution when the ENTER key is pressed.

Example

```
    x = cons();
```

At the cursor enter:

```
    probability
```

Now *x* will be equal to:

```
    x = "probability";
```

See Also

con

ConScore

Purpose

Compute local score statistic and its probability for hypotheses involving parameters under constraints

Format

{ *SL*, *SLprob* } = **ConScore**(*H*, *G*, *grad*, *a*, *b*, *c*, *d*, *bounds*, *psi*);

Input

H	KxK matrix, Hessian of loglikelihood with respect to parameters.
G	KxK matrix, cross-product matrix of the first derivatives by observation. If not available set to *H*.
grad	Kx1 vector, gradient of loglikelihood with respect to parameters.
a	MxK matrix, linear equality constraint coefficients.
b	Mx1 vector, linear equality constraint constants.

These arguments specify the linear equality constraints of the following type:

```
    a * X = b
```

where *X* is the Kx1 parameter vector.

c	MxK matrix, linear inequality constraint coefficients.
d	Mx1 vector, linear inequality constraint constants. These arguments specify the linear inequality constraints of the following type: `c * X >= d` where *X* is the Kx1 parameter vector.
bounds	Kx2 matrix, bounds on parameters. The first column contains the lower bounds, and the second column the upper bounds.
psi	indices of the set of parameters in the hypothesis.

Output

SL	scalar, local score statistic of hypothesis.
SLprob	scalar, probability of *SL*.

Remarks

ConScore computes the local score statistic for the hypothesis $H(\Theta) = 0$ vs. $H(\Theta) \geq 0$, where Θ is the vector of estimated parameters, and $H()$ is a constraint function of the parameters.

First, the model with $H(\Theta) = 0$ is estimated, and the Hessian and optionally the cross-product of the derivatives is computed. Also, the gradient vector is computed.

Next, the constraint arguments are set to $H(\Theta) \geq 0$.

Example

This example is from Silvapulle and Sen, *Constrained Statistical Inference*, page 181-3. It computes the local score statistic and probability for an ARCH model. It tests the null hypothesis of no arch effects against the alternative of arch effects subject to their being constrained to be positive.

The Hessian, H, cross-product matrix, G, and the gradient vector, grad, are generated by an estimation using **sqpSolvemt** where the model is an ARCH model with the arch parameters constrained to be zero.

```
#include sqpsolvemt.sdf
/* data */
struct DS d0;
d0 = reshape(dsCreate,2,1);

load z0[] = aoi.asc;
z = packr(lagn(251*ln(trimr(z0,1,0)./trimr(z0,0,1)),0|1|2|3
|4));
d0[1].dataMatrix = z[.,1];
d0[2].dataMatrix = z[.,2:5];


/* control structure */
struct sqpsolvemtControl c0;
c0 = sqpSolveMTcontrolCreate;

/* constraints setting arch parameter equal
to zero for H(theta) = 0 */

c0.A = zeros(3,6) ~ eye(3);
c0.B = zeros(3,1);

c0.covType = 2; /* causes cross-product of
                   Jacobian to be computed which
```

C

```
                     is needed for ConScore */

    struct PV p0;
    p0 = pvPack(pvCreate,.08999, "constant");
    p0 = pvPack(p0,.25167|-.12599|.09164|.07517,
        "phi");
    p0 = pvPack(p0,3.22713, "omega");
    p0 = pvPack(p0,0|0|0, "arch");


    struct sqpsolvemtOut out0;
    out0 = sqpsolvemt(&lpr,p0,d0,c0);

    /* set up constraints for H(theta) >= 0 */
    bounds = { -1e256 1e256,
               -1e256 1e256,
               -1e256 1e256,
               -1e256 1e256,
               -1e256 1e256,
               -1e256 1e256,
                    0 1e256,
                    0 1e256,
                    0 1e256 };
    H = out0.hessian;
    G = out0.xproduct;
    // minus because -logl in log-likelihood
    grad = -out0.gradient;

    psi = { 7, 8, 9 };

    { SL, SLprob } = ConScore(H,G,grad,0,0,0,0,bounds,psi);
```

will assign the variables *SL* and *SLprob* as follows:

```
SL = 3.8605086

SLprob = 0.10410000
```

Source

hypotest.src

continue

Purpose

Jumps to the top of a `do` or `for` loop.

Format

```
continue;
```

Example

```
x = rndn(4,4);

//Loop through each row of 'x' using 'r' as the loop
//counter
for r(0, rows(x), 1);
   //Loop through each element in our current row
   for c(0, cols(x), 1); /* continue jumps here */
         //If we are on the diagonal skip the rest of the
         //inner loop
         if c == r;
            continue;
```

```
              endif;
           //Set the non-diagonal elements to 0
           x[r,c] = 0;
        endfor;
     endfor;
```

Before the loops, *x* looks like:

```
     -1.4400255      0.15389012     -0.90423208     -0.62402330
      2.1330276      0.95605712     -1.2353752       1.1276577
      1.1526412      0.36105374      1.1462596       1.1907549
      0.41986542       1.0603897     -0.19616276       2.8940323
```

After the loops above, *x* looks like:

```
     -1.4400255       0.0000000       0.0000000       0.0000000
      0.0000000      0.95605712       0.0000000       0.0000000
      0.0000000       0.0000000       1.1462596       0.0000000
      0.0000000       0.0000000       0.0000000       2.8940323
```

Remarks

This command works just as in **C**.

contour

Purpose

Graphs a matrix of contour data. Note: This function is for the deprecated PQG graphics.

Library

pgraph

Format

```
contour(x, y,  z);
```

Input

x	1xK vector, the X axis data. K must be odd.
y	Nx1 vector, the Y axis data. N must be odd.
z	NxK matrix, the matrix of height data to be plotted.

Global Input

_plev	Kx1 vector, user-defined contour levels for **contour**. Default 0.
_pzclr	Nx1 or Nx2 vector. This controls the Z level colors. See **surface** for a complete description of how to set this global.

Remarks

A vector of evenly spaced contour levels will be generated automatically from the *z* matrix data. Each contour level will be labeled. For unlabeled contours, use **ztics**.

To specify a vector of your own unequal contour levels, set the vector *_plev* before calling **contour**.

To specify your own evenly spaced contour levels, see **ztics**.

Source

`pcontour.src`

See Also

surface

conv

Purpose

Computes the convolution of two vectors.

Format

```
c = conv(b, x, f, l);
```

Input

b	Nx1 vector.
x	Lx1 vector.
f	scalar, the first convolution to compute.
l	scalar, the last convolution to compute.

Output

c	Qx1 result, where: $Q = (l - f + 1)$ If *f* is 0, the first to the *l*'th convolutions are computed. If *l* is 0, the *f*'th to the last convolutions are computed. If *f* and *l* are both zero, all the convolutions are computed.

Remarks

If *x* and *b* are vectors of polynomial coefficients, this is the same as multiplying the two polynomials.

See Also

polymult

convertsatostr

Purpose

Converts a 1x1 string array to a string.

Format

```
str = convertsatostr(sa);
```

Input

sa	1x1 string array.

Output

str	string, *sa* converted to a string.

See Also

convertstrtosa

convertstrtosa

Purpose

Converts a string to a 1x1 string array.

Format

sa = **convertstrtosa**(*str*);

Input

str	string.

Output

sa	1x1 string array, *str* converted to a string array.

Example

```
str = "This is a string";
z = convertstrtosa(str);
```

You can check the types of your variables by viewing them on the **GAUSS** data page, or by using the `show` command. If the code above was executed at startup, running the `show` command would return:

```
24 bytes      str           STRING                              1
6 char
```

```
40 bytes      z            STRING ARRAY
1,1
```

See Also

convertsatostr

corrm,corrvc,corrx

Purpose

Computes an unbiased estimate of a correlation matrix.

Format

cx = **corrm**(*m*);
cx = **corrvc**(*vc*);
cx = **corrx**(*x*);

Input

m	KxK moment (*x*'*x*) matrix. A constant term MUST have been the first variable when the moment matrix was computed.
vc	KxK variance-covariance matrix (of data or parameters).
x	NxK matrix of data.

Output

cx	PxP correlation matrix. For **corrm**, P = K-1. For **corrvc** and **corrx**, P = K.

Remarks

The correlation matrix is the standardized version of the unbiased estimator of the population variance-covariance matrix. It is computed using the moment matrix of deviations about the mean divided by the number of observations minus one *N - 1*. For the observed correlation/covariance matrix which uses *N* rather than *N - 1*, see **corrms** and **corrxs**.

Source

```
corr.src
```

See Also

momentd, corrms, corrxs

corrms,corrxs

Purpose

Computes the observed correlation matrix.

Format

cx = **corrms**(*m*);
cx = **corrxs**(*x*);

Input

m	KxK moment (*x*'*x*) matrix. A constant term MUST have been the first variable when the moment matrix was computed.
x	NxK matrix of data.

Output

cx	PxP correlation matrix. For **corrms**, P = K-1. For **corrxs**, P = K.

Remarks

The correlation matrix is the standardized version of the correlation/covariance matrix computed from the input data, that is, it divides the sample size, *N*, rather than *N - 1*. For an unbiased estimate correlation/covariance matrix which uses *N - 1*, use **corrm** or **corrx**.

Source

`corrs.src`

See Also

momentd, corrm, corrx

cos

Purpose

Returns the cosine of its argument.

Format

y = **cos**(*x*);

Input

x	NxK matrix.

Output

y	NxK matrix containing the cosines of the elements of *x*.

Remarks

For real matrices, *x* should contain angles measured in radians.

To convert degrees to radians, multiply the degrees by $\pi/180$.

Example

```
//Create a sequence starting at 0 and increasing by pi/4
x = seqa(0, pi/4, 5);
y = cos(x);
```

```
         0.0000      1.0000
         0.7854      0.7071
     x = 1.5708  y = 0.0000
         2.3562     -0.7071
         3.1416     -1.0000
```

See Also

atan, atan2, pi

cosh

Purpose

Computes the hyperbolic cosine.

Format

y = **cosh**(*x*);

Input

x	NxK matrix.

Output

y	NxK matrix containing the hyperbolic cosines of the elements of *x*.

Example

```
    x = { -0.5, -0.25, 0, 0.25, 0.5, 1 };
    x = x * pi;
    y = cosh(x);
```

```
       -1.5708       2.5092
       -0.7854       1.3246
    x = 0.0000   y = 1.0000
        0.7854       1.3246
        1.5708       2.5092
        3.1416      11.5920
```

Source

`trig.src`

counts

Purpose

Counts the numbers of elements of a vector that fall into specified ranges.

Format

c = **`counts`**(*x*, *v*);

Input

x	Nx1 vector containing the numbers to be counted.
v	Px1 vector containing breakpoints specifying the ranges

within which counts are to be made. The vector *v* MUST be sorted in ascending order.

Output

c — Px1 vector, the counts of the elements of *x* that fall into the regions:

```
              x ≤ v[1],
       v[1] < x ≤ v[2],
           .
           .
           .
     v[p-1] < x ≤ v[p]
```

Remarks

If the maximum value of *x* is greater than the last element (the maximum value) of *v*, the sum of the elements of the result, *c*, will be less than N, the total number of elements in *x*.

If

```
         1
         2
         3
         4      4
     x = 5  v = 5
         6      8
         7
         8
         9
```

then

```
          4
     c =  1
          3
```

The first category can be a missing value if you need to count missings directly. Also +∞ or -∞ are allowed as breakpoints. The missing value must be the first breakpoint if it is included as a breakpoint and infinities must be in the proper location depending on their sign. -∞ must be in the [2,1] element of the breakpoint vector if there is a missing value as a category as well, otherwise it has to be in the [1,1] element. If +∞ is included, it must be the last element of the breakpoint vector.

Example

Example 1

```
     x = { 1.5, 3, 5, 4, 1, 3 };
     v = { 0, 2, 4 };
     c = counts(x,v);
```

```
       1.5
          3        0        0
     x =  2   v =  2   c =  2
          4        4        3
          1
          3
```

Example 2

```
     x = { -3, -2.5, -2, 0, -9, 18 };
     v = __infn|-2|__infp;
     c = counts(x,v);
```

```
          -INF        0
     v =    -2   c =  4
          +INF        2
```

Example 3

```
x = { -2.3, -0.4, -3.1, -0.2, 1.9, -1.4 };

//Change all values in 'x' equal to -0.2 to be a miss
ing value
x = miss(x, -0.2);

v = { ., -2, 0, 2 };
c = counts(x,v);
```

```
    -2.3          .         1
    -0.4         -2         2
x = -3.1   v =    0    c =  2
       .          2         1
     1.9
    -1.4
```

countwts

Purpose

Returns a weighted count of the numbers of elements of a vector that fall into specified ranges.

Format

c = **countwts**(*x*, *v*, *w*);

Input

x	Nx1 vector, the numbers to be counted.
v	Px1 vector, the breakpoints specifying the ranges within

	which counts are to be made. This MUST be sorted in ascending order (lowest to highest).
w	Nx1 vector, containing weights.

Output

c Px1 vector containing the weighted counts of the elements of *x* that fall into the regions:

```
              x ≤ v[1],
       v[1] < x ≤ v[2],
           .
           .
           .
     v[p-1] < x ≤ v[p]
```

That is, when *x*[*i*] falls into region *j*, the weight *w*[*i*] is added to the *j*th counter.

Remarks

If any elements of *x* are greater than the last element of *v*, they will not be counted.

Missing values are not counted unless there is a missing in *v*. A missing value in *v* MUST be the first element in *v*.

Example

```
     x = { 1, 3, 2, 4, 1, 3 };
     w = { .25, 1, .333, .1, .25, 1 };
     v = { 0, 1, 2, 3, 4 };
```

```
    c = countwts(x,v,w);
```

```
        0.0000000
        0.5000000
    c = 0.3330000
        2.0000000
        0.1000000
```

create

Purpose

Creates and opens a **GAUSS**data set for subsequent writing.

Format

create [[*vflag*]] [[-w32]] [[complex]] *fh* = *filename* with *vnames*, *col*, *dtyp*, *vtyp*;

create [[*vflag*]] [[-w32]] [[complex]] *fh* = *filename* using *comfile*;

Input

vflag	literal, version flag.	
	-v89	obsoleted, use -*v96*.
	-v92	obsoleted, use-*v96*.
	-v96	supported on all platforms.
	For details on the various versions, see **Foreign Language**	

	Interface, Chapter 1. The default format can be specified in `gauss.cfg` by setting the *`dat_fmt_version`* configuration variable. The default, *`v96`*, should be used.
`filename`	literal or ^string *`filename`* is the name to be given to the file on the disk. The name can include a path if the directory to be used is not the current directory. This file will automatically be given the extension `.dat`. If an extension is specified, the `.dat` will be overridden. If the name of the file is to be taken from a string variable, the name of the string must be preceded by the ^ (caret) operator.

`create... with...`

`vnames`	literal or ^string or ^character matrix. *`vnames`* controls the names to be given to the columns of the data file. If the names are to be taken from a string or character matrix, the ^ (caret) operator must be placed before the name of the string or character matrix. The number of columns parameter, *`col`*, also has an effect on the way the names will be created. See below and see the examples for details on the ways names are assigned to a data file.
`col`	scalar expression. *`col`* is a scalar expression containing the number of columns in the data file. If *`col`* is 0, the number of columns will be controlled by the contents of *`vnames`*. If *`col`* is positive, the file will contain *`col`* columns and the names to be given each column will be created as necessary depending on the *`vnames`* parameter. See the examples.
`dtyp`	scalar expression.

dtyp is the precision used to store the data. This is a scalar expression containing 2, 4, or 8, which is the number of bytes per element.

2	signed integer
4	single precision
8	double precision

Data Type	Digits	Range		
integer	4	-32768	$< X <$	32768
single	6-7	8.43×10^{-37}	$< \lvert X \rvert <$	$3.37 \times 10^{+38}$
double	15-16	4.19×10^{-307}	$< \lvert X \rvert <$	1.67×10^{308}

If the integer type is specified, numbers will be rounded to the nearest integer as they are written to the data set. If the data to be written to the file contains character data, the precision must be 8 or the character information will be lost.

vtyp matrix, types of variables.

The types of the variables in the data set. If **rows**(*vtyp*) ***cols**(*vtyp*) < *col*, only the first element is used. Otherwise nonzero elements indicate a numeric variable and zero elements indicate character variables.

create... using...

comfile literal or ^string.

comfile is the name of a command file that contains the information needed to create the file. The default extension for the command file is `.gcf`, which can be overridden.

There are three possible commands in this file:

```
    numvar   n str;
    outvar   varlist;
    outtyp   dtyp;
```

`numvar` and **`outvar`** are alternate ways of specifying the number and names of the variables in the data set to be created.

When **`numvar`** is used, *`n`* is a constant which specifies the number of variables (columns) in the data file and *`str`* is a string literal specifying the prefix to be given to all the variables. Thus:

```
    numvar 10 xx;
```

says that there are 10 variables and that they are to be named *`xx01`* through *`xx10`*. The numeric part of the names will be padded on the left with zeros as necessary so the names will sort correctly:

xx1,	...	xx9	1-9 names
xx01,	...	xx10	10-99 names
xx001,	...	xx100	100-999 names
xx0001,	...	xx1000	1000-8100 names

If *`str`* is omitted, the variable prefix will be "*`X`*".

When **`outvar`** is used, *`varlist`* is a list of variable names, separated by spaces or commas. For instance:

```
    outvar x1, x2, zed;
```

C

specifies that there are to be 3 variables per row of the data set, and that they are to be named *X1*, *X2*, *ZED*, in that order.

outtyp specifies the precision. It can be a constant: 2, 4, or 8, or it can be a literal: I, F, or D. For an explanation of the available data types, see *dtyp* in `create... with...` previously.

The **outtyp** statement does not have to be included. If it is not, then all data will be stored in 4 bytes as single precision floating point numbers.

Output

fh	scalar. *fh* is the file handle which will be used by most commands to refer to the file within **GAUSS**. This file handle is actually a scalar containing an integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the `create` (or `open`) command is executed.

Remarks

If the *complex* flag is included, the new data set will be initialized to store complex number data. Complex data is stored a row at a time, with the real and imaginary halves interleaved, element by element.

The *-w32* flag is an optimization for Windows. It is ignored on all other platforms. **GAUSS** 7.0 and later use Windows system file write commands that support 64-bit file sizes. These commands are slower on Windows XP than the 32-bit file write commands that were used in **GAUSS** 6.0 and earlier. If you include the *-w32* flag, successive writes to the file indicated by *fh* will use 32-bit Windows write commands, which will be faster on Windows XP. Note, however, that the `-w32` flag does not support 64-bit file sizes.

Example

```
let vnames = age sex educat wage occ;
create f1 = simdat with ^vnames,0,8;

obs = 0; nr = 1000;
do while obs < 10000;
   data = rndn(nr,colsf(f1));
   if writer(f1,data) /= nr;
      print "Disk Full";
      end;
   endif;
   obs = obs+nr;
endo;

closeall f1;
```

This example uses `create... with...` to create a double precision data file called `simdat.dat` on the default drive with 5 columns. The **`writer`** command is used to write 10000 rows of Normal random numbers into the file. The variables (columns) will be named: *AGE*, *SEX*, *EDUCAT*, *WAGE*, *OCC*.

Here are some examples of the variable names that will result when using a character vector of names in the argument to the `create` function.

```
vnames = { AGE PAY SEX JOB };
typ = { 1, 1, 0, 0 };
create fp = mydata with ^vnames,0,8,typ;
```

The names in the this example will be: *AGE*, *PAY*, *SEX*, *JOB*.

AGE and *PAY* are numeric variables, *SEX* and *JOB* are character variables.

```
    create fp = mydata with ^vnames,3,2;
```

The names will be: *AGE*, *PAY*, *SEX*.

```
    create fp = mydata with ^vnames,8,2;
```

The names will now be: *AGE*, *PAY*, *SEX*, *JOB1*, *JOB2*, *JOB3*, *JOB4*, *JOB5*.

If a literal is used for the *vnames* parameter, the number of columns should be explicitly given in the *col* parameter and the names will be created as follows:

```
    create fp = mydata with var,4,2;
```

Giving the names: *VAR1*, *VAR2*, *VAR3*, *VAR4*.

The next example assumes a command file called `comd.gcf` containing the following lines, created using a text editor:

```
    outvar age, pay, sex;
    outtyp i;
```

Then the following program could be used to write 100 rows of random integers into a file called `smpl.dat` in the subdirectory called /gauss/data:

```
    filename = "/gauss/data/smpl";
    create fh = ^filename using comd;
    x = rndn(100,3)*10;
    if writer(fh,x) /= rows(x);
      print "Disk Full";
      end;
    endif;
    closeall fh;
```

For platforms using the backslash as a path separator, remember that two backslashes ("\\") are required to enter one backslash inside of double quotes. This is because a backslash is the escape character used to embed special characters in strings.

See Also

datacreate, datacreatecomplex, open, readr, writer, eof, close, output, iscplxf

crossprd

Purpose

Computes the cross-products (vector products) of sets of 3x1 vectors.

Format

z = **crossprd**(*x*, *y*);

Input

x	3xK matrix, each column is treated as a 3x1 vector.
y	3xK matrix, each column is treated as a 3x1 vector.

Output

z	3xK matrix, each column is the cross-product (sometimes called vector product) of the corresponding columns of *x* and *y*.

Remarks

The cross-product vector *z* is orthogonal to both *x* and *y*. **sumc**(*x* .* *z*) and **sumc**(*y* .* *z*) will be Kx1 vectors, all of whose elements are 0 (except for rounding error).

Example

```
    x = { 10  4,
          11 13,
         14 13 };
    y = { 3 11,
          5 12,
          7  9 };
    z = crossprd(x,y);

          7  -39
    z = -28  107
         17  -95
```

Source

`crossprd.src`

crout

Purpose

Computes the Crout decomposition of a square matrix without row pivoting, such that: $X = LU$.

Format

y = **`crout`**(x);

Input

x	NxN square nonsingular matrix.

Output

y	NxN matrix containing the lower (*L*) and upper (*U*) matrices of the Crout decomposition of *x*. The main diagonal of *y* is the main diagonal of the lower matrix *L*. The upper matrix has an implicit main diagonal of ones. Use **`lowmat`** and **`upmat1`** to extract the *L* and *U* matrices from *y*.

Remarks

Since it does not do row pivoting, it is intended primarily for teaching purposes. See **`croutp`** for a decomposition with pivoting.

Example

```
X = { 1 2 -1,
      2 3 -2,
      1 -2 1 };

//Perform crout decomposition
y = crout(x);

//Extract lower triangle of 'y' and assign it to 'L'
L = lowmat(y);

//Extract upper triangle of 'y', fill the diagonal with
//ones and assign it to 'L'
U = upmat1(y);
```

After the code above:

```
          1.0   2.0 -1.0          1.0   0.0   0.0          1.0   2.0 -1.0
    y = 2.0 -1.0   0.0    L = 2.0 -1.0   0.0    U = 0.0   1.0   0.0
          1.0 -4.0   2.0          1.0 -4.0   2.0          0.0   0.0   1.0
```

See Also

croutp, chol, lowmat, lowmat1, lu, upmat, upmat1

croutp

Purpose

Computes the Crout decomposition of a square matrix with partial (row) pivoting.

Format

y = **croutp**(*x*);

Input

x	NxN square nonsingular matrix.

Output

y	(N+1)xN matrix containing the lower (*L*) and upper (*U*) matrices of the Crout decomposition of a permuted *x*. The N+1 row of the matrix *y* gives the row order of the *y* matrix. The matrix must be reordered prior to extracting the *L* and *U* matrices. Use **lowmat** and **upmat1** to extract the *L* and

U matrices from the reordered *y* matrix.

Example

This example illustrates a procedure for extracting *L* and *U* of the permuted *x* matrix. It continues by sorting the result of *LU* to compare with the original matrix *x*.

```
X = { 1 2 -1,
      2 3 -2,
      1 -2 1 };

y = croutp(x);
```

If we view 'y', we will see:

```
    1.0000       0.50000       0.28571
y = 2.0000        1.5000      -1.0000
    1.0000       -3.5000      -0.57142
    2.0000        3.0000        1.0000
```

```
//This bottom row is the permutation index vector
//Calculate how many rows in 'y'
r = rows(y);

//Extract the index row and transpose it into a column
//vector
indx = y[r,.]';
```

Viewing 'indx' will reveal:

```
       2
indx = 3
       1
```

```
//Rearrange the rows of 'y' based upon the index vector
z = y[indx,.];

// obtain L and U of permuted matrix X
L = lowmat(z);
U = upmat1(z);

//Horizontally concatenate the index vector and the product
//of L*U then pass that result into the 'sortc' function
//which will sort this result based upon the first column
//(which is the index vector)
q = sortc(indx~(L*U),1);

//Remove the index vector, which we added by way of
//horizontal concatenation in the statement just above
x2 = q[.,2:cols(q)];
```

Now at the end of this example, *x2* is equal to *x*.

See Also

crout, chol, lowmat, lowmat1, upmat, upmat1

csrcol,csrlin

Purpose

Returns the position of the cursor.

Format

y = **csrcol**;
y = **csrlin**;

Output

y	scalar, row or column value.

Remarks

y will contain the current column or row position of the cursor on the screen. The upper left corner is (1,1).

csrcol returns the column position of the cursor. **csrlin** returns the row position.

The `locate` commmand allows the cursor to be positioned at a specific row and column.

csrcol returns the cursor column with respect to the current output line, i.e., it will return the same value whether the text is wrapped or not. **csrlin** returns the cursor line with respect to the top line in the window.

Example

```
r = csrlin;
c = csrcol;

//Clear the program input/output window
cls;

//Re-position the cursor to its location before the program
//input/output window was cleared
locate r,c;
```

In this example the screen is cleared without affecting the cursor position.

See Also

cls, locate

cumprodc

Purpose

Computes the cumulative products of the columns of a matrix.

Format

y = **cumprodc**(*x*);

Input

x	NxK matrix.

Output

y	NxK matrix containing the cumulative products of the columns of *x*.

Remarks

This is based on the recursive series **recsercp**. **recsercp** could be called directly as follows:

```
recsercp(x, zeros(1,cols(x)));
```

to accomplish the same thing.

Example

```
x = { 1 -3,
          2   2,
          3  -1 };
    y = cumprodc(x);
```

Now if you view *y*, you will see:

```
        1.000 -3.000
    y = 2.000 -6.000
        6.000  6.000
```

Source

`cumprodc.src`

See Also

cumsumc, recsercp, recserar

cumsumc

Purpose

Computes the cumulative sums of the columns of a matrix.

Format

y = **cumsumc**(*x*);

Input

x	NxK matrix.

Output

y	NxK matrix containing the cumulative sums of the columns of *x*.

Remarks

This is based on the recursive series function **recserar**. **recserar** could be called directly as follows:

```
    recserar(x, x[1,.],ones(1,cols(x)))
```

to accomplish the same thing.

Example

```
    x = { 1 -3,
          2  2,
          3 -1 };

    y = cumsumc(x);
```

Now if you view *y*, you will see:

```
        1.000 -3.000
    y = 3.000 -1.000
        6.000 -2.000
```

Source

cumsumc.src

See Also

cumprodc, recsercp, recserar

curve

Purpose

Computes a one-dimensional smoothing curve.

Format

{ *u*, *v* } = **curve**(*x*, *y*, *d*, *s*, *sigma*, *G*);

Input

x	Kx1 vector, x-abscissae (X-axis values).
y	Kx1 vector, y-ordinates (Y-axis values).
d	Kx1 vector or scalar, observation weights.
s	scalar, smoothing parameter. If *s* = 0, **curve** performs an interpolation. If *d* contains standard deviation estimates, a reasonable value for *s* is K.
sigma	scalar, tension factor.
G	scalar, grid size factor.

Output

u	(K*G)x1 vector, x-abscissae, regularly spaced.
v	(K*G)x1 vector, y-ordinates, regularly spaced.

Remarks

`sigma` contains the tension factor. This value indicates the curviness desired. If `sigma` is nearly zero (e.g. .001), the resulting curve is approximately the tensor product of cubic curves. If `sigma` is large, (e.g. 50.0) the resulting curve is approximately bi-linear. If `sigma` equals zero, tensor products of cubic curves result. A standard value for `sigma` is approximately 1.

G is the grid size factor. It determines the fineness of the output grid. For *G* = 1, the input and output vectors will be the same size. For *G* = 2, the output grid is twice as fine as the input grid, i.e., *u* and *v* will have twice as many rows as *x* and *y*.

Source

`spline.src`

cvtos

Purpose

Converts a character vector to a string.

Format

s = **cvtos**(*v*);

Input

v	Nx1 character vector, to be converted to a string.

Output

s	string, contains the contents of *v*.

Remarks

cvtos in effect appends the elements of *v* together into a single string.

cvtos was written to operate in conjunction with **stocv**. If you pass it a character vector that does not conform to the output of **stocv**, you may get unexpected results. For example, **cvtos** does NOT look for 0 terminating bytes in the elements of *v*; it assumes every element except the last is 8 characters long. If this is not true, there will be 0's in the middle of *s*.

If the last element of *v* does not have a terminating 0 byte, **cvtos** supplies one for *s*.

Example

```
    let v = { "Now is t" "he time " "for all " "good men"
     };
    s = cvtos(v);
```

Now the variable *s* is a string with the following contents.

```
    s = "Now is the time for all good men"
```

See Also

stocv, vget, vlist, vput, vread

datacreate

Purpose

Creates a real data set.

Format

```
fh = datacreate(filename, vnames, col, dtyp, vtyp);
```

Input

filename	string, name of data file.	
vnames	string or Nx1 string array, names of variables.	
col	scalar, number of variables.	
dtyp	scalar, data precision, one of the following:	
	2	2-byte, signed integer.
	4	4-byte, single precision.
	8	8-byte, double precision.
vtyp	scalar or Nx1 vector, types of variables, may contain one or both of the following:	
	0	character variable.

	1 numeric variable.

Output

fh	scalar, file handle.

Remarks

The file handle returned by **datacreate** is a scalar containing a positive integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the `create`, **datacreate**, **datacreatecomplex**, `open` or **dataopen** commands are executed. The file handle is used to reference the file in the commands **readr** and **writer**. If **datacreate** fails, it returns a -1.

If *filename* does not include a path, then the file is placed on the current directory. The file is given a `.dat` extension if no extension is specified.

If *col* is set to 0, then the number of columns in the data set is controlled by the contents of *vnames*. If *col* is positive, then the file will contain *col* columns.

If *vnames* contains *col* elements, then each column is given the name contained in the corresponding row of *vnames*. If *col* is positive and *vnames* is a string, then the columns are given the names *vnames*1, *vnames*2, ..., *vnames*N (or *vnames*01, *vnames*02, ..., *vnames*N), where N = *col*. The numbers appended to *vnames* are padded on the left with zeros to the same length as N.

The *dtyp* argument allows you to specify the precision to use when storing your data. Keep in mind the following range restrictions when selecting a value for *dtyp*:

Data Type	Digits	Range

integer	4	$-32768 < X < 32767$
single	6-7	$8.43 \times 10^{-37} < \|X\| \leq 3.37 \times 10^{+38}$
double	15-16	$4.19 \times 10^{-307} < \|X\| < 1.67 \times 10^{+308}$

If the integer type is specified, numbers are rounded to the nearest integer as they are written to the data set. If the data to be written to the file contains character data, the precision must be 8 or the character information will be lost.

If *vtyp* is a scalar, then the value in *vtyp* controls the types of all of the columns in the data set. If it is an Nx1 vector, then the type of each column is controlled by the value in the corresponding row of *vtyp*.

Example

```
     fh = datacreate("myfile.dat", "V",100,8,1);
     x = rndn(500,100);
     r = writer(fh,x);
     ret = close(fh);
```

This example creates a double precision data file called `myfile.dat`, which is placed in the current directory. The file contains 100 columns with 500 observations (rows), and the columns are given the names 'V001', 'V002', ..., 'V100'.

Source

`datafile.src`

See Also

datacreatecomplex, create, dataopen, writer

datacreatecomplex

Purpose

Creates a complex data set.

Format

```
fh = datacreatecomplex(filename, vnames, col, dtyp,
vtyp);
```

Input

filename	string, name of data file.	
vnames	string or Nx1 string array, names of variables.	
col	scalar, number of variables.	
dtyp	scalar, data precision, one of the following:	
	2	2-byte, signed integer.
	4	4-byte, single precision.
	8	8-byte, double precision.
vtyp	scalar or Nx1 vector, types of variables, may contain one or both of the following:	
	0	character variable.
	1	numeric variable.

Output

fh	scalar, file handle.

Remarks

The file handle returned by **datacreatecomplex** is a scalar containing a positive integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the create, **datacreate**, **datacreatecomplex**, open or **dataopen** commands are executed. The file handle is used to reference the file in the commands **readr** and **writer**. If **datacreatecomplex** fails, it returns a -1.

Complex data is stored a row at a time, with the real and imaginary halves interleaved, element by element. For columns containing character data, the imaginary parts are zeroed out.

If *filename* does not include a path, then the file is placed on the current directory. The file is given a `.dat` extension if no extension is specified.

If *col* is set to 0, then the number of columns in the data set is controlled by the contents of *vnames*. If *col* is positive, then the file will contain *col* columns.

If *vnames* contains *col* elements, then each column is given the name contained in the corresponding row of *vnames*. If *col* is positive and *vnames* is a string, then the columns are given the names *vnames*1, *vnames*2, ..., *vnames*N (or *vnames*01, *vnames*02, ..., *vnames*N), where N = *col*. The numbers appended to *vnames* are padded on the left with zeros to the same length as N.

The *dtyp* argument allows you to specify the precision to use when storing your data. Keep in mind the following range restrictions when selecting a value for *dtyp*:

Data Type	Digits	Range
integer	4	$-32768 < X < 32767$
single	6-7	$8.43 \times 10^{-37} < \lvert X \rvert \leq 3.37 \times 10^{+38}$
double	15-16	$4.19 \times 10^{-307} < \lvert X \rvert < 1.67 \times 10^{+308}$

If the integer type is specified, numbers are rounded to the nearest integer as they are written to the data set. If the data to be written to the file contains character data, the precision must be 8 or the character information will be lost.

If `vtyp` is a scalar, then the value in `vtyp` controls the types of all of the columns in the data set. If it is an Nx1 vector, then the type of each column is controlled by the value in the corresponding row of `vtyp`.

Example

```
    string vnames = { "random1", "random2" };
    fh = datacreatecomplex("myfilecplx.dat",vnames,2,8,1);
    x = complex(rndn(1000,2),rndn(1000,2));
    r = writer(fh,x);
    ret = close(fh);
```

This example creates a complex double precision data file called `myfilecplx.dat`, which is placed in the current directory. The file contains 2 columns with 1000 observations (rows), and the columns are given the names 'random1' and 'random2'.

Source

`datafile.src`

See Also

datacreate, create, dataopen, writer

datalist

Purpose

List selected variables from a data set.

Format

```
datalist dataset [[var 1 [[var 2 ...]]]];
```

Input

dataset	literal, name of the data set.
var#	literal, the names of the variables to list.

Global Input

__*range*	scalar, the range of rows to list. The default is all rows.	
__*miss*	scalar, controls handling of missing values.	
	0	display rows with missing values.
	1	do not display rows with missing values.
The default is 0.		
__*prec*	scalar, the number of digits to the right of the decimal point to display. The default is 3.	

Remarks

The variables are listed in an interactive mode. As many rows and columns as will fit on the screen are displayed. You can use the cursor keys to pan and scroll around in the listing.

Example

```
    datalist freq age sex pay;
```

This command will display the variables *age*, *sex*, and *pay* from the data set `freq.dat`.

Source

`datalist.src`

dataload

Purpose

Loads matrices, N-dimensional arrays, strings and string arrays from a disk file.

Format

y = **dataload**(*filename*);

Input

filename	string, name of data file.

Output

y	matrix, array, string or string array, data retrieved from the file.

Remarks

The proper extension must be included in the file name. Valid extensions are as follows:

`.fmt`	matrix file
	array file
`.fst`	string file
	string array file

See **Foreign Language Interface**, Chapter 1, for details on these file types.

Example

```
    y = dataload("myfile.fmt");
```

See Also

load, datasave

dataloop (dataloop)

Purpose

Specifies the beginning of a data loop.

Format

`dataloop` *`infile outfile`*;

Input

`infile`	string variable or literal, the name of the source data set.

Output

`outfile`	string variable or literal, the name of the output data set.

Remarks

The statements between the `dataloop`... `endata` commands are assumed to be metacode to be translated at compile time. The data from *`infile`* is manipulated by the specified statements, and stored to the data set *`outfile`*. Case is not significant within the `dataloop`... `endata` section, except for within quoted strings. Comments can be used as in any **GAUSS** code.

Example

```
src = "source";
dataloop ^src dest;
make newvar = x1 + x2 + log(x3);
```

```
x6 = sqrt(x4);
keep x6, x5, newvar;
endata;
```

Here, `src` is a string variable requiring the caret (^) operator, while `dest` is a string literal.

dataopen

Purpose

Opens a data set.

Format

fh = **dataopen**(*filename*, *mode*);

Input

filename	string, name of data file.	
mode	string containing one of the following:	
	read	open file for read.
	append	open file for append.
	update	open file for update.

Output

fh	scalar, file handle.

Remarks

The file must exist before it can be opened with the **dataopen** command (to create a new file, see **datacreate** or **datasave**).

The file handle returned by **dataopen** is a scalar containing a positive integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the `create`, **datacreate**, **datacreatecomplex**, `open` or **dataopen** commands are executed. The file handle is used to reference the file in the commands **readr** and **writer**. If **dataopen** fails, it returns a -1.

A file can be opened simultaneously under more than one handle. If the value that is in the file handle when the **dataopen** command begins to execute matches that of an already open file, the process will be aborted and a File already open error message will be given. This gives you some protection against opening a second file with the same handle as a currently open file. If this happens, you would no longer be able to access the first file.

It is important to set unused file handles to zero because both **dataopen** and **datacreate** check the value that is in a file handle to see if it matches that of an open file before they proceed with the process of opening a file. You may set unused file handles to zero with the **close** or `closeall` commands.

If *`filename`* does not have an extension, **dataopen** appends a `.dat` extension before searching for the file. If the file is an `.fmt` matrix file, the extension must be explicitly given. If no path information is included, then **dataopen** searches for the file in the current directory.

Files opened in *`read`* mode cannot be written to. The pointer is set to the beginning of the file and the **writer** function is disabled for files opened in this way. This is the only mode available for matrix files (`.fmt`), which are always written in one piece with the `save` command.

Files opened in *`append`* mode cannot be read. The pointer is set to the end of the file so that a subsequent write to the file with the **writer** function will add data to the end of the file without overwriting any of the existing data in the file. The **readr** function is disabled for files opened in this way. This mode is used to add additional rows to the end of a file.

Files opened in *update* mode can be read from and written to. The pointer is set to the beginning of the file. This mode is used to make changes in a file.

Example

```
    fh = dataopen("myfile.dat","read");
    y = readr(fh,100);
    ret = close(fh);
```

This example opens the data file `myfile.dat` in the current directory and reads 100 observations (rows) from the file into the global variable *y*.

Source

`datafile.src`

See Also

open, datacreate, writer, readr

datasave

Purpose

Saves matrices, N-dimensional arrays, strings and string arrays to a disk file.

Format

ret = **datasave**(*filename*, *x*);

Input

filename	string, name of data file.
x	matrix, array, string or string array, data to write to disk.

Output

ret	scalar, return code, 0 if successful, or -1 if it is unable to write the file.

Remarks

datasave can be used to save matrices, N-dimensional arrays, strings and string arrays. The following extensions are given to files that are saved with **datasave**:

matrix	`.fmt`
array	`.fmt`
string	`.fst`
string array	`.fst`

See **Foreign Language Interface**, Chapter 1, for details on these file types.

Use **dataload** to load a data file created with **datasave**.

Example

```
    x = rndn(1000,100);
```

```
ret = datasave("myfile.fmt",x);
```

See Also

save, dataload

date

Purpose

Returns the current date in a 4-element column vector, in the order: year, month, day, and hundredths of a second since midnight.

Format

y = **date**;

Remarks

The hundredths of a second since midnight can be accessed using **hsec**.

Example

```
print date;
```

```
   2012.0
      7.0
     16.0
4571524.7
```

See Also

time, timestr, ethsec, hsec, etstr

datestr

Purpose

Returns a date in a string.

Format

str = **datestr**(*d*);

Input

d	4x1 vector, like the **date** function returns. If this is 0, the **date** function will be called for the current system date.

Output

str	8 character string containing current date in the form: *mo/dy/yr*

Example

```
d = { 2012, 10, 09, 0 };
y = datestr(d);
print y;
```

produces the following output:

```
    10/09/12
```

Source

`time.src`

See Also

date, datestring, datestrymd, time, timestr, ethsec

datestring

Purpose

Returns a date in a string with a 4-digit year.

Format

str = **datestring**(*d*);

Input

d	4x1 vector, like the **date** function returns. If this is 0, the **date** function will be called for the current system date.

Output

str	10 character string containing current date in the form: *mm/dd/yyyy*

Example

```
    dt = { 2012, 12, 18, 0 };
    y = datestring(dt);
    print y;
```

produces the following output:

```
    12/18/2012
```

Source

`time.src`

See Also

date, datestr, datestrymd, time, timestr, ethsec

datestrymd

Purpose

Returns a date in a string in the form *yyyymmdd*.

Format

str = **datestrymd**(*d*);

Input

d	4x1 vector, like the **date** function returns. If this is 0, the

	`date` function will be called for the current system date.

Output

str	8 character string containing current date in the form: *yyyymmdd*

Example

```
    d = { 2012, 11, 16, 0 };
    y = datestrymd(d);
    print y;
```

returns:

```
    20121116
```

Source

time.src

See Also

date, datestr, datestring, time, timestr, ethsec

dayinyr

Purpose

Returns day number in the year of a given date.

Format

daynum = **dayinyr**(*dt*);

Input

dt	3x1 or 4x1 vector, date to check. The date should be in the form returned by **date**.

Output

daynum	scalar, the day number of that date in that year.

Example

```
x = { 1973, 8, 31, 0 };
y = dayinyr(x);
print y;
```

produces:

```
y = 243.00000
```

Source

time.src

Globals

_isleap

dayofweek

Purpose

Returns day of week.

Format

d = **dayofweek**(*a*);

Input

a	Nx1 vector, dates in DT format.

Output

d	Nx1 vector, integers indicating day of week of each date:

1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday
7	Saturday

Remarks

The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number

```
    20120401183207
```

represents 18:32:07 or 6:32:07 PM on April 4, 2012.

Source

`time.src`

dbAddDatabase

Purpose

Adds a database to the list of database connections using the driver type or a connection URL.

Format

db_id = **dbAddDatabase**(*driver_type*);
db_id = **dbAddDatabase**(*connection_url*);

Input

driver_type	string, supported options include: MYSQL, OCI, ODBC,

	PSQL,
	SQLITE
`connection_url`	string, with the following format:
	driver://username:password@hostname:port/database_name

Output

`db_id`	scalar, index into a table of all opened database connections, or 0 on failure.

Example

```
    db_id = dbAddDatabase("MYSQL");
```

```
    url = "mysql://webuser:pswd@localhost:3306/dev";
    db_id = dbAddDatabase(url);
```

Remarks

Before using the connection, it must be initialized. e.g., call some or all of **`dbSetDatabaseName()`**, **`dbSetUserName()`**, **`dbSetPassword()`**, **`dbSetHostName()`**, **`dbSetPort()`**, and **`dbSetConnectOptions()`**, and, finally, **`dbOpen()`**.

The exception to this is using a connection URL, since this performs the above mentioned steps. Omitting portions of the connection URL is allowed, but the syntax must remain the same. For example:

```
    id = dbAddDatabase("oci://root:@localhost:/testing");
```

is a valid connection URL, but will not set the password or port number fields.

dbClose

Purpose

Closes a database connection and destroys any remaining queries.

Format

`dbClose`(*`db_id`*);

Input

`db_id`	scalar, database connection index number.

Remarks

`dbClose()` does not remove the database connection from the list of available database connections. The connection can be opened again without repeating the database initialization and setup steps.

dbCommit

Purpose

Commits a transaction to the database if the driver supports transactions and a **`dbTransaction()`** has been started.

Format

```
ret = dbCommit(db_id);
```

Input

db_id	scalar, database connection index number.

Output

ret	scalar, 1 for success or 0 for failure.

Example

```
    db_id = dbAddDatabase("SQLITE");
    dbTransaction(db_id);
    dbExecQuery(db_id, "INSERT INTO PEOPLE
        (first, last) VALUES ('John', 'Doe');");
    dbCommit(db_id);
    dbClose(db_id);
```

Remarks

Note: For some databases, the commit will fail and return 0 if there is an active query using the database for a SELECT statement. Make the query inactive before doing the commit to resolve this problem.

Call **`dbGetLastErrorText()`** to get information about errors.

dbCreateQuery

Purpose

Process an SQL statement and prepare a query. If *`placeholders`* is present, these values are bound sequentially to ODBC style parameters.

Format

`qid` = **`dbCreateQuery`**(*`db_id`*);
`qid` = **`dbCreateQuery`**(*`db_id`*, *`query`*);
`qid` = **`dbCreateQuery`**(*`db_id`*, *`query`*, *`placeholders`*);

Input

`db_id`	scalar, database connection index number.
`query`	string, database query to construct.
`placeholders`	string, or string array containing bind value(s).

Output

`qid`	scalar, query id to be used for result retrieval.

Example

Example 1

```
qid = dbCreateQuery("SELECT * FROM GDP
      WHERE COUNTRY = ?", "USA");
dbQueryExecPrepared(qid);

// Results as a matrix
results = dbQueryFetchAllM(qid);
```

Example 2

```
qid = dbCreateQuery("INSERT INTO
      PEOPLE(id, fname, lname) VALUES
      (NULL, ?, ?);");
dbQueryAddBindValue(qid, "Joe");
dbQueryAddBindValue(qid, "Smith");
dbQueryExecPrepared(qid);
```

Remarks

If the `placeholders` parameter is passed in, the values are bound sequentially to ODBC style parameters.

See also

dbQueryPrepare

dbExecQuery

Purpose

Executes an SQL statement and creates a query. If *`placeholders`* is present, these values are bound sequentially to ODBC style parameters.

Format

`qid` = **`dbExecQuery`**(*`db_id`*, *`sql_statement`*);
`qid` = **`dbExecQuery`**(*`db_id`*, *`sql_statement`*, *`placeholders`*);

Input

`db_id`	scalar, database connection index number.
`sql_statement`	string containing a valid SQL statement.
`placeholders`	string (array) containing bind value(s).

Output

`qid`	scalar, query id to be used for result retrieval.

Example

In the examples below, *`db_id`* is a previously created database id.

d

Example 1

```
qid = dbExecQuery(db_id, "SELECT * FROM GDP
      WHERE COUNTRY = ?", "USA");

// Results as a matrix
results = dbQueryFetchAllM(qid);
```

Example 2

```
bd_vals = "Joe"$|"Smith";
qid = dbExecQuery(db_id, "INSERT INTO PEOPLE(id,
       fname, lname); VALUES (NULL, ?, ?);", bd_vals);
```

Example 3

```
qid = dbExecQuery("SELECT * FROM PEOPLE
      p WHERE p.FNAME = ?", "Joe");

// Results as a string array
results = dbQueryFetchAllSA(qid);
```

dbGetConnectOptions

Purpose

Returns the connection options string used for a database connection.

Format

options = **dbGetConnectOptions**(*db_id*);

Input

db_id	scalar, database connection index number.

Output

options	string, containing the connection options for the specified database connection.

See Also

dbSetConnectOptions

Remarks

If you have not set any connection options with **dbSetConnectOptions**, then this function will return an empty string. For a full list of options see **dbSetConnectOptions**.

dbGetDatabaseName

Purpose

Returns the name of the database.

Format

```
db_name = dbGetDatabaseName(db_id);
```

Input

db_id	scalar, database connection index number.

Output

db_name	string, name of the database.

dbGetDriverName

Purpose

Returns the name of the connection's database driver.

Format

```
driver_name = dbGetDriverName(db_id);
```

Input

db_id	scalar, database connection index number.

Output

driver_name	string, name of the database driver.

Example

```
    db_id = dbAddDatabase("SQLITE");
    print "Driver = " dbGetDriverName(db_id);
```

will print the following output

```
    Driver = SQLITE
```

dbGetDrivers();

Purpose

Returns a list of available database drivers.

Format

drivers = **dbGetDrivers**();

Output

drivers	Nx1 string array, list of available database drivers.

Example

```
    print dbGetDrivers();
```

```
        MYSQL
        OCI
        ODBC
        PSQL
        SQLITE
```

dbGetHostName

Purpose

Returns the database connection's host name

Format

host_name = **dbGetHostName**(*db_id*);

Input

db_id	scalar, database connection index number.

Output

host_name	string, name of database connection.

dbGetLastErrorNum

Purpose

Returns information about the last error that occurred on the database.

Format

```
last_error = dbGetLastErrorNum(db_id);
```

Input

db_id	scalar, database connection index number.

Output

last_error	scalar, number of last error on the specified database.

dbGetLastErrorText

Purpose

Returns information about the last error that occurred on the database.

Format

```
last_error = dbGetLastErrorText(db_id);
```

Input

db_id	scalar, database connection index number.

Output

last_error	string, details of last error on the specified database.

dbGetNumericalPrecPolicy

Purpose

Returns the default numerical precision policy for a specified database connection.

Format

prec_policy = **dbGetNumericalPrecPolicy**(*db_id*);

Input

db_id	scalar, database connection index number.

Output

prec_policy scalar:

DB_HIGH_PRECISION	strings will be used to preserve precision
DB_LOW_PRECISION_INT32	Force 32-bit integer values

DB_LOW_PRECISION_INT64	Force 64-bit integer values
DB_LOW_PRECISION_DOUBLE	Force double values. This is the default policy.

dbGetPassword

Purpose

Returns a connection's password.

Format

db_password = **dbGetPassword**(*db_id*);

Input

db_id	scalar, database connection index number.

Output

db_password	string containing the password for the specified database connection or a null string.

Remarks

`dbGetPassword()` will only return passwords set with **`dbSetPassword()`** or specified in the URL string for **`dbAddDatabase()`**.

dbGetPort

Purpose

Returns the database connection's port number if it has been set.

Format

`db_port` = **`dbGetPort`**(*`db_id`*);

Input

`db_id`	scalar, database connection index number.

Output

`db_port`	scalar, the port number of the specified database connection.

Remarks

`dbGetPort()` will only return the port number if it was previously set with **`dbSetPort()`**.

dbGetTableHeaders

Purpose

Returns a string array populated with the names of all the fields in a specified table (or view).

Format

field_names = **dbGetTableHeaders**(*db_id*, *table_name*);

Input

db_id	scalar, database connection index number.
table_name	string, name of table or view.

Output

field_names	string array containing the column names for the specified table or view.

Remarks

The order in which the fields appear in the record is undefined.

dbGetTables

Purpose

Returns the database's tables, system tables and views.

Format

```
tables = dbGetTables(db_id, type);
tables = dbGetTables(db_id);
```

Input

db_id	scalar, database connection index number.	
type	scalar:	
	DB_TABLES	All tables visible to the user. This is the default value.
	DB_SYSTEM_TABLES	Internal tables used by the database.
	DB_VIEWS	All views visible to the user.
	DB_ALL_TABLES	All of the above.

Output

tables	Nx1 string array containing the information specified by the 'type' parameter.

dbGetUserName

Purpose

Returns the database connection's user name.

Format

```
user_name = dbGetUserName(db_id);
```

Input

db_id	scalar, database connection index number.

Output

user_name	string containing the user name associated with the specified database connection.

See Also

dbSetUserName

dbHasFeature

Purpose

Returns a 1 if the database supports the specified feature.

Format

```
ret = dbHasFeature(db_id, feature);
```

Input

db_id scalar, database connection index number.

feature scalar or define:

Define	Scalar	Description
DB_TRANSACTIONS	0	Whether the driver supports SQL transactions.
DB_QUERY_SIZE	1	Whether the database is capable of reporting the size of a query. Note that some databases do not support returning the size (i.e. number

		of rows returned) of a query, in which case dbQueryRows() will return -1.
DB_BLOB	2	Whether the driver supports Binary Large Object fields.
DB_UNICODE	3	Whether the driver supports Unicode strings if the database server does.
DB_PREPARED_QUERIES	4	Whether the driver supports prepared query execution.
DB_NAMED_PLACEHOLDERS	5	Whether the driver supports the use of named

		placeholders.
DB_POSITIONAL_PLACEHOLDERS	6	Whether the driver supports the use of positional placeholders.
DB_LAST_INSERT_ID	7	Whether the driver supports returning the Id of the last touched row.
DB_BATCH_OPERATIONS	8	Whether the driver supports batched operations. (Not supported)
DB_SIMPLE_LOCKING	9	Whether the driver disallows a write lock on a table while other queries have a read lock on it.
DB_LOW_PRECISION_NUMBERS	10	Whether

		the driver allows fetching numerical values with low precision.
DB_EVENT_NOTIFICATIONS	11	Whether the driver supports database event notification s.
DB_FINISH_QUERY	12	Whether the driver can do any low-level resource cleanup when dbQueryFin ish() is called.
DB_MULTIPLE_RESULT_SETS	13	Whether the driver can access multiple result sets returned from batched statements

	or stored procedures. (Not supported)

Output

ret	scalar, 1 if the database supports the specified feature, or 0 if not.

Example

```
db_id = dbAddDatabase("MYSQL");

// Create empty query
qid = dbCreateQuery(db_id);

if dbHasFeature(db_id, DB_NAMED_PLACEHOLDERS);
     dbQueryPrepare(qid, "SELECT * FROM GDP
      WHERE COUNTRY = :country");
     dbQueryBindValue(qid, ":country", "USA");
else;
     dbQueryPrepare(qid, "SELECT * FROM GDP
      WHERE COUNTRY = ?");
     dbQueryBindValue(qid, 1, "USA");
endif;

dbQueryExecPrepared(qid);
```

Remarks

Note that some databases need to be opened with **`dbOpen()`** before this can be determined.

dbIsDriverAvailable

Purpose

Returns 1 if a specified database driver is available.

Format

`ret` = **`dbIsDriverAvailable`**(*`name`*);

Input

`name`	string, name of driver to check

Output

`ret`	scalar, 1 if the specified driver is available, or 0 if not.

dbIsOpen

Purpose

Reports whether a specified database connection is open.

Format

```
ret = dbIsOpen(db_id);
```

Input

db_id	scalar, database connection index number.

Output

ret	scalar, 1 if the connection is open or 0 if it is closed.

dbIsOpenError

Purpose

Reports whether an error occurred while attempting to open the database connection.

Format

```
ret = dbIsOpenError(db_id);
```

Input

db_id	scalar, database connection index number.

Output

ret	scalar, 1 if there was an error or 0 if not.

dbNumericalPrecPolicy

Purpose

Returns the default numerical precision policy for a specified database connection.

Format

```
prec_policy = dbNumericalPrecPolicy(db_id);
```

Input

db_id	scalar, database connection index number.

Output

prec_policy	scalar:	
	0	strings will be used to preserve precision
	1	Force 32-bit integer values
	2	Force 64-bit integer values
	4	Force double values. This is the default policy.

dbOpen

Purpose

Opens a specified database connection using the current connection values.

Format

```
ret = dbOpen(db_id, user_name, password);
ret = dbOpen(db_id);
```

Input

db_id	scalar, database connection index number.
user_name	string, user name for the database being connected to.
password	string, password associated with the specified user name for this database.

Output

ret	scalar, 1 for success.

Example

Set driver and host

```
db_id = dbAddDatabase("MYSQL");
dbSetHostName(db_id, "localhost");
```

then, either

```
    dbSetUserName(db_id, "test");
    dbSetPassword(db_id, "secret_passw0rd");
    ret = dbOpen(db_id);
```

or

```
    ret = dbOpen(db_id, "test", "secret_passw0rd");
```

dbQueryBindValue

Purpose

Set the placeholder placeholder to be bound to value val in the prepared statement. Note that the placeholder mark (e.g :) must be included when specifying the placeholder name.

Format

dbQueryBindValue(*qid*, *placeholder*, *val*);

Input

qid	scalar, query number.
placeholder	string, Oracle style (:value_name) or integer, index of ODBC style (?) placeholder.
val	valid type, the value to be bound.

Remarks

Values cannot be bound to multiple locations in the query.

Example

Oracle style

```
db_id = dbAddDatabase("MYSQL");
qid = dbCreateQuery(db_id);
dbQueryPrepare(qid, "SELECT * FROM
     PEOPLE WHERE FIRST = :fname AND
     LAST = :lname");
dbQueryBindValue(qid, ":fname", "John");
dbQueryBindValue(qid, ":lname", "Doe");
dbQueryExecPrepared(qid);
```

ODBC Style

```
db_id = dbAddDatabase("MYSQL");
qid = dbCreateQuery(db_id);
dbQueryPrepare(qid, "SELECT * FROM
     PEOPLE WHERE FIRST = ? AND
     LAST = ?");
dbQueryBindValue(qid, 1, "John");
dbQueryBindValue(qid, 2, "Doe");
dbQueryExecPrepared(qid);
```

dbQueryClear

Purpose

Clears the result set and releases any resources held by the query. Sets the query state to inactive.

Format

`dbQueryClear`(*`qid`*);

Input

`qid`	scalar, query number.

Remarks

You should rarely if ever need to call this function.

dbQueryCols

Purpose

Returns the number of fields in the record.

Format

`num_fields` = **`dbQueryCols`**(*`qid`*);

Input

qid	scalar, query number.

Output

num_fields	scalar, number of fields.

dbQueryExecPrepared

Purpose

Executes a previously created and prepared query.

Format

ret = **dbQueryExecPrepared**(*qid*);

Input

qid	scalar, query number.

Output

ret	scalar, 1 for success and 0 for failure.

Example

```
qid = dbCreateQuery(db_id);
dbQueryPrepare(qid, "SELECT * FROM
     USERS WHERE ID = :id");
dbQueryBindValue(qid, ":id", 5);
dbQueryExecPrepared(qid);

results = dbQueryFetchAllSA(qid);

dbQueryBindValue(qid, ":id", 10);

// Re-execute the query with new value
dbQueryExecPrepared(qid);

results = dbQueryFetchAllSA(qid);
```

dbQueryFetchAllM

Purpose

Returns the result set for the current query as a matrix.

Format

result = **dbQueryFetchAllM**(*qid*);
result = **dbQueryFetchAllM**(*qid*, *columns*);

Input

qid	scalar, query number.

columns	string or string array, specific columns to pull out from result matrix. Must be a subset of fields from SELECT statement.

Output

result	matrix, the result set; or if the result set is empty, a scalar error code.

Remarks

For string results, or to treat numerical results as strings, use **dbQueryFetchAllSA()** to return a string array.
This function retrieves all rows at once. You can process rows in an iterative manner by using the **dbQueryFetchOneM()** and **dbQueryFetchOneSA()** functions.

Example

Example 1

```
qid = dbExecQuery(db_id, "SELECT * FROM GDP");

gdp = dbQueryFetchAllM(qid);

// If 'gdp' is a scalar error code
if scalmiss(gdp);
    print "No results";
else;
    // do something with gdp
endif;
```

Example 2

```
qid = dbExecQuery(db_id, "SELECT * FROM
      PEOPLE WHERE COUNTRY = ?", "USA");

// specify zipcode as column of interest
zipcodes = dbQueryFetchAllM(qid, "ZIPCODE");

if not scalmiss(zipcodes);
    print "zip codes = " zipcodes;
endif;
```

See Also

dbQueryFetchAllSA, dbQueryFetchOneM, dbQueryFetchOneSA

dbQueryFetchAllSA

Purpose

Returns the result set for the current query as a string array.

Format

result = **dbQueryFetchAllSA**(*qid*);
result = **dbQueryFetchAllSA**(*qid*, *columns*);

Input

qid	scalar, query number.
columns	string or string array, specific columns to pull out from result

	matrix. Must be a subset of fields from SELECT statement.

Output

result	string array, containing the result set for the current query. If the result set is empty, a scalar error code is returned.

Remarks

For numerical only results, use **dbQueryFetchAllM()** to return a matrix. This function retrieves all rows at once. You can process rows in an iterative manner by using the **dbQueryFetchOneM()** and **dbQueryFetchOneSA()** functions.

Example

```
qid = dbExecQuery(db_id, "SELECT * FROM
      PEOPLE WHERE COUNTRY = ?", "USA");

// specify names as columns of interest
names = dbQueryFetchAllSA(qid,
     "FIRST_NAME"$|"LAST_NAME");

// If 'names' is not a scalar error code
if not scalmiss(names);
    print "People in the USA = " names;
endif;
```

See Also

dbQueryFetchAllM, dbQueryFetchOneSA, dbQueryFetchOneM

dbQueryFetchOneM

Purpose

Returns a single row as an Nx1 matrix where N is the column count of the SELECT statement containing the field information for the current query.

Format

record = **dbQueryFetchOneM**(*qid*);
record = **dbQueryFetchOneM**(*qid*, *columns*);

Input

qid	scalar, query number.
columns	string or string array, specific columns to pull from the result matrix. Must be a subset of fields from the SELECT statement.

Output

record	matrix, if the query points to a valid row (**dbQueryIsValid()** returns true), the record is populated with the row's values. An empty record (**scalmiss** *(record)* is true) is returned when there is no active query (**dbQueryIsActive()** returns false).

Remarks

This function is only useful in an iterative context. You can easily retrieve all the results at once by using the **dbQueryFetchAllM()** and **dbQueryFetchAllSA()** functions.

For string results, or to treat numerical results as a string, using **dbQueryFetchOneSA()** will return a string array.

Example

```
qid = dbExecQuery(db_id, "SELECT YTD,
      TOTAL FROM GDP");

do while dbQuerySeekNext(qid);
   record = dbQueryFetchOneM(qid);
   ytd = record[1];
   total = record[2];
endo;
```

See Also

dbQueryFetchOneSA, dbQueryFetchAllM, dbQueryFetchAllSA, dbQueryGetField

dbQueryFetchOneSA

Purpose

Returns a single row as a string vector containing the field information for the current query.

Format

record = **dbQueryFetchOneSA**(*qid*);
record = **dbQueryFetchOneSA**(*qid*, *columns*);

Input

qid	scalar, query number.
columns	string or string array, specific columns to pull from the result matrix. Must be a subset of fields from the SELECT statement.

Output

record	string array, if the query points to a valid row (**dbQueryIsValid()** returns true), the record is populated with the row's values. An empty record (**scalmiss** *(record)* is true) is returned when there is no active query **dbQueryIsActive()** returns false).

Remarks

This function is only useful in an iterative context. You can easily retrieve all the results at once by using the **dbQueryFetchAllM()** and **dbQueryFetchAllSA ()** functions.

For numerical only results, using **dbQueryFetchOneM()** will return a matrix instead of a string array.

Example

```
qid = dbExecQuery(db_id, "SELECT COUNTRY,
    TOTAL FROM GDP");
do while dbQuerySeekNext(qid);
    record = dbQueryFetchOneSA(qid);
```

```
        country = record[1];
        total = record[2];
    endo;
```

See Also

dbQueryFetchOneM, dbQueryFetchAllM, dbQueryFetchAllSA, dbQueryGetField

dbQueryFinish

Purpose

Instructs the database driver that no more data will be fetched from this query until it is re-executed.

Format

dbQueryFinish(*qid*);

Input

qid	scalar, query number.

Remarks

There is normally no need to call this function, but it may be helpful in order to free resources such as locks or cursors if you intend to re-use the query at a later time.

Sets the query to inactive. Bound values retain their values.

dbQueryGetBoundValue

Purpose

Returns the value for a placeholder in a query.

Format

val = **dbQueryGetBoundValue**(*qid*, *placeholder*);

Input

qid	scalar, query number.
placeholder	string, Oracle style (:value_name) or integer, index of ODBC style (?) placeholder.

Output

val	string, bound value if previously set.

Example

```
db_id = dbAddDatabase("MYSQL");
qid = dbCreateQuery(db_id);
dbQueryPrepare(qid, "SELECT * FROM
    PEOPLE WHERE FIRST = :fname AND
    LAST = :lname");
dbQueryBindValue(qid, ":fname", "John");
dbQueryBindValue(qid, ":lname", "Doe");
```

```
    print "Name = ";;
    print dbQueryGetBoundValue(qid, ":fname");;
    print dbQueryGetBoundValue(qid, ":lname");
```

or

```
    db_id = dbAddDatabase("MYSQL");
    args = "John"$|"Doe";
    qid = dbCreateQuery(db_id, "SELECT * FROM
        PEOPLE WHERE FIRST = ? AND LAST = ?", args);

    print "Name = ";;
    print dbQueryGetBoundValue(qid, 1);;
    print dbQueryGetBoundValue(qid, 2);
```

results in

```
    Name = John Doe
```

dbQueryGetBoundValues

Purpose

Returns an Nx2 string array containing the placeholders and their corresponding values in a query.

Format

bound_values = **dbQueryGetBoundValues**(*qid*);

Input

qid	scalar, query number.

Output

bound_values	Nx2 string array. The first column contains the placeholders and the second column contains the corresponding values.

Example

```
db_id = dbAddDatabase("MYSQL");
qid = dbCreateQuery(db_id);
dbQueryPrepare(qid, "SELECT * FROM
      PEOPLE WHERE FIRST = :fname AND
      LAST = :lname");
dbQueryBindValue(qid, ":fname", "John");
dbQueryBindValue(qid, ":lname", "Doe");

print "Vars = " dbQueryGetBoundValues(qid);
```

will print

```
Vars =
   :fname   John
   :lname   Doe
```

dbQueryGetField

Purpose

Returns the value of a specified field in the current record. An overloaded version that accepts a column name as input is available, but not as efficient.

Format

field_value = **dbQueryGetField**(*qid*, *idx*);
field_value = **dbQueryGetField**(*qid*, *name*);

Input

qid	scalar, query number.
idx	scalar, index of the field whose value should be returned.

Remarks

The fields are numbered from left to right using the text of the SELECT statement, e.g. in

```
qid = dbExecQuery("SELECT forename, surname FROM peopl
e");

do while dbQuerySeekNext(qid);
   forename = dbQueryGetField(qid, 1);
   // Using field index
   surname = dbQueryGetField(qid, 2);
   // Using field index
   forename = dbQueryGetField(qid, "forename");
```

```
        // Using field name
        surname = dbQueryGetField(qid, "surname");
        // Using field name
    endo;
```

Field 1 is *forename* and field 2 is *surname*. Using `SELECT *` is not recommended because the order of the fields in the query is undefined.

See Also

dbQueryFetchOneM, dbQueryFetchOneSA

dbQueryGetLastErrorNum

Purpose

Returns error information about the last error that occurred (if any) with the last executed query.

Format

err_num = **dbQueryGetLastErrorNum**();

Output

err_num	scalar, number of last error.

Remarks

Because a failed query will not have a valid handle (id), this function retrieves stored error information about the last executed query.

See also

dbQueryGetLastErrorText

dbQueryGetLastErrorText

Purpose

Returns error information about the last error that occurred (if any) with the last executed query.

Format

err_txt = **dbQueryGetLastErrorText**();

Output

err_txt	2x1 string array, database and driver text of last error.

Remarks

Because a failed query will not have a valid handle (id), this function retrieves stored error information about the last executed query.

See also

dbQueryGetLastErrorNum

dbQueryGetLastInsertID

Purpose

Returns the object ID of the most recent inserted row if supported by the database.

Format

last_insert = **dbQueryGetLastInsertID**(*qid*);

Input

qid	scalar, query number.

Output

last_insert	scalar, object id

Remarks

If more than one row was touched by the insert, the behavior is undefined.

For MySQL databases the row's auto-increment field will be returned.

With a PSQL database, the table must contain OID's which were not created by default. Check the *default_with_oids* configuration variable to be sure.

Example

```
// Given NAMES is an empty MySQL
// table with the *id* column
// auto-incrementing.
db_id = dbAddDatabase("MYSQL");
qid = dbCreateQuery(db_id, "INSERT
     INTO NAMES (first, last) VALUES
     ('John', 'Doe');");

if dbHasFeature(db_id, DB_LAST_INSERT_ID);
    last_id = dbQueryGetLastInsertID(qid);
endif;
```

See Also

dbHasFeature

dbQueryGetLastQuery

Purpose

Returns the text of the current query being used.

Format

query_string = **dbQueryGetLastQuery**(qid);

Input

qid	scalar, query number.

Output

query_string	string, text of the current query, or empty string if there is no current query.

dbQueryGetPosition

Purpose

Returns the current internal position of the query.

Format

```
index = dbQueryGetPosition(qid)
```

Input

qid	scalar, query number.

Output

index	scalar, query position

Remarks

The first record is at position zero. If the position is invalid, the function returns DB_BEFORE_FIRST_ROW or DB_AFTER_LAST_ROW, which are special negative values.

Example

```
    qid = dbCreateQuery(db_id, "SELECT *
         FROM PEOPLE");
    do while dbQuerySeekNext(qid);
        print "Current index = "
        dbQueryGetPosition(qid);
    endo;
```

dbQueryIsActive

Purpose

Returns 1 if the query is active.

Format

ret = **dbQueryIsActive**(*qid*);

Input

qid	scalar, query number.

Output

ret	scalar, 1 if the query is active or 0 if not.

Remarks

An active query is one that has been **dbQueryExecPrepared**()'d successfully, but not yet finished with. When you are finished with an active query, you can make the query inactive by calling **dbQueryFinish**() or **dbQueryClear**().

Note: Of particular interest is an active query that is a SELECT statement. For some databases that support transactions, an active query that is a SELECT statement can cause a **dbCommit**() or a **dbRollback**() to fail, so before committing or rolling back, you should make your active SELECT statement query inactive using one of the methods listed above.

Example

```
qid = dbCreateQuery(db_id);
dbQueryIsActive(qid); // False

dbQueryPrepare(qid, "INSERT INTO TEST
      (foo, bar) VALUES (1, 2);");
dbQueryIsActive(qid); // False

dbQueryExecPrepared(qid);
dbQueryIsActive(qid); // True

dbQueryFinish(qid);
dbQueryIsActive(qid); // False
```

dbQueryIsForwardOnly

Purpose

Reports whether you can only scroll forward through a result set.

Format

ret = **dbQueryIsForwardOnly**(*qid*);

Input

qid	scalar, query number.

Output

ret	scalar, 1 if the result set can only be scrolled through forward, otherwise a 0.

Remarks

Setting a query to "forward only" will usually improve performance. By default, queries are created with "forward only" off.

See Also

dbQuerySetForwardOnly, dbQuerySeekNext

dbQueryIsNull

Purpose

Returns 1 if the query is active, positioned on a valid record and the field is NULL; otherwise returns 0. Reports whether the current field pointed at by an active query positioned on a valid record is NULL.

Format

```
ret = dbQueryIsNull(qid, field);
```

Input

qid	scalar, query number.
field	scalar, index into result set.

Output

ret	scalar, 1 if the field is NULL or 0 otherwise.

Remarks

Note that for some drivers, **`dbQueryIsNull`**() will not return accurate information until after an attempt is made to retrieve data.

See Also

dbQueryIsActive, dbQueryIsValid

dbQueryIsSelect

Purpose

Reports whether the specified query is a SELECT statement.

Format

ret = **`dbQueryIsSelect`**(*qid*);

Input

qid	scalar, query number.

Output

ret	scalar, 1 if the query is a SELECT statement or 0 otherwise.

Example

```
qid = dbExecQuery(db_id, "SELECT *
    FROM PEOPLE");

dbQueryIsSelect(qid); // True

qid = dbExecQuery(db_id, "INSERT INTO
    PEOPLE (fname, lname) VALUES
    ('John', 'Doe');");

dbQueryIsSelect(qid); // False
```

dbQueryIsValid

Purpose

Reports whether the specified query is positioned on a valid record.

Format

ret = **dbQueryIsValid**(*qid*);

Input

qid	scalar, query number.

Output

ret	scalar, 1 if the query is positioned on a valid record or 0 otherwise.

Example

```
qid = dbExecQuery(db_id, "SELECT * FROM
      PEOPLE");

dbQueryIsValid(qid); // False
// Give it a valid position
dbQuerySeekFirst(qid);

// Iterate until no longer valid.
do while dbQueryIsValid(qid);
    // dbQueryIsValid = True
    dbQuerySeekNext(qid);
endo;

dbQueryIsValid(qid); // False
```

dbQueryPrepare

Purpose

Prepares a SQL query for execution.

Format

ret = **dbQueryPrepare**(*qid*, *query*);

Input

qid	scalar, query index number.
query	string, database query to prepare.

Output

ret	scalar, 1 for success and 0 for failure.

Example

```
db_id = dbAddDatabase("MYSQL");
qid = dbCreateQuery(db_id);
ret = dbQueryPrepare(qid, "SELECT *
    FROM STOCKS WHERE SYMBOL = :sym");
dbQueryBindValue(qid, ":sym", "GOOG");
ret = dbQueryExecPrepared(qid);
```

Remarks

The query may contain placeholders for binding values. Both Oracle style colon-name (e.g., :surname), and ODBC style (?) placeholders are supported; but they cannot be mixed in the same query.

Portability note: Some databases choose to delay preparing a query until it is executed the first time. In this case, preparing a syntactically incorrect query succeeds, but every consecutive **dbQueryExecPrepared**() will fail.

For SQLite, the query string can contain only one statement at a time. If more than one statement is given, the function returns 0.

See also

dbQueryBindValue

dbQueryRows

Purpose

Returns the size of the result (number of rows returned), or -1 if the size cannot be determined or if the database does not support reporting information about query sizes.

Format

result_size = **dbQueryRows**(*qid*);

Input

qid	scalar, query number.

Output

result_size	scalar, number of rows in the current result set of the active query. If the number of rows cannot be determined a -1 is

returned.

Remarks

Note that if the query is not active, a -1 is returned. This property can be checked with **`dbQueryIsActive()`**.

Example

```
    // Given a table with US States.
    qid = dbCreateQuery(db_id, "SELECT *
         FROM STATES");

    count = dbQueryRows(qid); // count = 50
```

dbQuerySeek

Purpose

Retrieves the record at a specified position, if available, and positions the query on the retrieved record.

Format

ret = **`dbQuerySeek`**(*`qid`*, *`idx`*, *`idx_type`*);
ret = **`dbQuerySeek`**(*`qid`*, *`idx`*);

Input

qid	scalar, query number.
idx	scalar, the index at which to place the cursor.
idx_type	scalar, 1 for relative position or 0 for absolute positioning. If not specified, absolute positioning is used.

Output

ret	scalar, 1 if successful.

Remarks

The first record is at position 1. Note that the query must be in an active state before calling this function. The state of the query may be verified with the function **dbQueryIsSelect()**.

If *idx_type* is 0 (the default), the following rules apply:

If *idx* is negative, the result is positioned before the first record and 0 is returned. Otherwise, an attempt is made to move to the record at position *idx*. If the record at position *idx* could not be retrieved, the result is positioned after the last record and 0 is returned. If the record is successfully retrieved, 1 is returned.

If *idx_type* is 1, the following rules apply:

If the result is currently positioned before the first record or on the first record, and *idx* is negative, there is no change, and 0 is returned.

If the result is currently located after the last record, and *idx* is positive, there is no change, and 0 is returned. If the result is currently located somewhere in the middle, and the relative offset `idx` moves the result below zero, the result is positioned before the first record and 0 is returned.

Otherwise, an attempt is made to move to the record `idx` records ahead of the current record (or `idx` records behind the current record if `idx` is negative).

If the record at offset `idx` could not be retrieved, the result is positioned after the last record if `idx` >= 0, (or before the first record if `idx` is negative), and 0 is returned. If the record is successfully retrieved, 1 is returned.

See Also

dbQuerySeekFirst, dbQuerySeekLast, dbQuerySeekNext, dbQuerySeekPrevious

dbQuerySeekFirst

Purpose

Retrieves the first record in the result, if available, and positions the query on the retrieved record.

Format

ret = **dbQuerySeekFirst**(*qid*);

Input

qid	scalar, query number.

Output

ret	1 if successful. If unsuccessful the query position is set to an

invalid position and 0 is returned.

Remarks

Note that the result must be in the active state or it will do nothing and return. This can be verified by calling the **dbQueryIsSelect**() function.

Example

```
qid = dbCreateQuery(db_id, "SELECT *
     FROM PEOPLE");

do while dbQuerySeekNext(qid);
    // iterate over results
endo;

// set back to start
dbQuerySeekFirst(qid);

do while dbQuerySeekNext(qid);
    // iterate over results AGAIN
endo;
```

dbQuerySeekLast

Purpose

Retrieves the last record in the result, if available, and positions the query on the retrieved record.

Format

```
ret = dbQuerySeekLast(qid);
```

Input

qid	scalar, query number.

Output

ret	scalar, returns 1 if successful. If unsuccessful the query position is set to an invalid position and 0 is returned.

Remarks

Note that the result must be in the active state and dbQueryIsSelect() must return 1 before calling this function or it will do nothing and return 0.

Example

```
    // Given STATES is a table with all
    // 50 states listed alphabetically
    qid = dbExecQuery(db_id, "SELECT name
        FROM STATES");

    // Move to last state
    ret = dbQuerySeekLast(qid);

    //If 'ret' is equal to 0
    if not  ret;
```

```
      print  "dbQuerySeekLast failed";
   else;
      // Print last state: Wyoming
      print dbQueryFetchOneSA(qid);
   endif;
```

See also

dbQuerySeekNext, dbQuerySeekPrevious, dbQuerySeekFirst, dbQuerySeek, dbQueryGetPosition

dbQuerySeekNext

Purpose

Retrieves the next record in the result, if available, and positions the query on the retrieved record.

Format

ret = **dbQuerySeekNext**(*qid*);

Input

qid	scalar, query number.

Output

ret	scalar, if the record could not be retrieved, the result is positioned after the last record and 0 is returned. If the record

is successfully retrieved, 1 is returned.

Remarks

Note that the result must be in the active state before calling this function or it will do nothing and return 0. You can verify the status of the query with **dbQueryIsSelect**().

The following rules apply:

If the result is currently located before the first record, e.g., immediately after a query is executed, an attempt is made to retrieve the first record.

If the result is currently located after the last record, there is no change and 0 is returned.

If the result is located somewhere in the middle, an attempt is made to retrieve the next record.

Example

```
    qid = dbCreateQuery(db_id, "SELECT *
          FROM PEOPLE");

    do while dbQuerySeekNext(qid);
         row = dbQueryFetchOneSA(qid);
         // Or dbQueryFetchOneM(qid) if data
          // is numeric
    endo;
```

See Also

dbQuerySeekFirst, dbQuerySeekLast, dbQuerySeekPrevious, dbQuerySeek, dbQueryGetPosition

dbQuerySeekPrevious

Purpose

Retrieves the previous record in the result, if available, and positions the query on the retrieved record.

Format

```
ret = dbQuerySeekPrevious(qid);
```

Input

qid	scalar, query number.

Output

ret	scalar, 1 if the record is successfully retrieved. If the record could not be retrieved, the result is positioned before the first record and 0 is returned.

Remarks

Note that the result must be in the active state before calling this function or it will do nothing and return false. The state of the query can be verified with **`dbQueryIsSelect()`**.

The following rules apply:

If the result is currently located before the first record, there is no change and 0 is returned.

If the result is currently located after the last record, an attempt is made to retrieve the last record.

If the result is somewhere in the middle, an attempt is made to retrieve the previous record.

See Also

dbQuerySeekFirst, dbQuerySeekLast, dbQuerySeekNext, dbQuerySeek, dbQueryGetPositino

dbQuerySetForwardOnly

Purpose

Sets forward only mode to forward. If forward is true, only **dbQuerySeekNext**() and **dbQuerySeek**() with positive values are allowed for navigating the results.

Format

dbQuerySetForwardOnly(*qid*, *forward*);

Input

qid	scalar, query number.
forward	scalar, 1 to set forward only or 0 to allow seeking in either direction.

Remarks

Forward only mode can be (depending on the driver) more memory efficient since results do not need to be cached. It will also improve performance on some databases. For this to be true, you must call **`dbQuerySetForwardOnly`**() before the query is prepared or executed.

Forward only mode is set to off by default.

Setting forward only to false is a suggestion to the database engine, which has the final say on whether a result set is forward only or scrollable.

`dbQueryIsForwardOnly`() will always return the correct status of the result set.

See Also

dbQueryIsForwardOnly

dbRemoveDatabase

Purpose

Removes a database connection from the list of open database connections. Frees all related resources.

Format

`dbRemoveDatabase`(*`db_id`*);

Input

`db_id`	scalar, database connection index number.

dbRollback

Purpose

Rolls back a transaction on the database.

Format

`ret` = **`dbRollback`**(*`db_id`*);

Input

`db_id`	scalar, database connection index number.

Output

`ret`	scalar, 1 to indicate success and a 0 if the rollback fails.

Remarks

A rollback is only possible if the SQL driver supports transactions and a **`dbTransaction`**() has been started.

Note: For some databases, the rollback will fail and return 0 if there is an active query using the database for a `SELECT`. Make the query inactive before doing the rollback.

Call **`dbGetLastErrorText`**() to get information about errors.

dbSetConnectOptions

Purpose

Sets database-specific options.

Format

`dbSetConnectOptions`(*`db_id`*, *`db_options`*);

Input

`db_id`	scalar, database connection index number.
`db_options`	string, a semi-colon separated list of option names or `option=value` pairs. Available options will depend upon the database being used.

Remarks

This must be done before the connection is opened or it has no effect (or you can **`dbClose`**() the connection, call this function and **`dbOpen`**() the connection again). The format of the options string is a semicolon separated list of option names or option=value pairs. The options depend on the database client used:

```
ODBC
SQL_ATTR_ACCESS_MODE
SQL_ATTR_LOGIN_TIMEOUT
SQL_ATTR_CONNECTION_TIMEOUT
SQL_ATTR_CURRENT_CATALOG
SQL_ATTR_METADATA_ID
SQL_ATTR_PACKET_SIZE
SQL_ATTR_TRACEFILE
SQL_ATTR_TRACE
SQL_ATTR_CONNECTION_POOLING
SQL_ATTR_ODBC_VERSION

MySQL
CLIENT_COMPRESS
CLIENT_FOUND_ROWS
CLIENT_IGNORE_SPACE
CLIENT_SSL
CLIENT_ODBC
CLIENT_NO_SCHEMA
CLIENT_INTERACTIVE
UNIX_SOCKET
MYSQL_OPT_RECONNECT

PostgreSQL
connect_timeout
options
```

```
tty
requiressl
service

DB2
SQL_ATTR_ACCESS_MODE
SQL_ATTR_LOGIN_TIMEOUT

OCI
OCI_ATTR_PREFETCH_ROWS
OCI_ATTR_PREFETCH_MEMORY

TDS
none

SQLite
QSQLITE_BUSY_TIMEOUT
QSQLITE_OPEN_READONLY
QSQLITE_ENABLE_SHARED_CACHE

Interbase
ISC_DPB_LC_CTYPE
ISC_DPB_SQL_ROLE_NAME
```

Example

```
    // MySQL connection
    // use an SSL connection to the server
    dbSetConnectOptions(db_id, "CLIENT_SSL=1;
        CLIENT_IGNORE_SPACE=1");
    if not dbOpen();
        // clears the connect option string
        dbSetConnectOptions(db_id, "");
        ...
    endif;
```
...
```
    // PostgreSQL connection
    // enable PostgreSQL SSL connections
    dbSetConnectOptions(db_id, "requiressl=1");
    if not dbOpen();
        // clear options
        dbSetConnectOptions(db_id, "");
        ...
    endif;
```
...
```
    // ODBC connection
    dbSetConnectOptions(db_id, "SQL_ATTR_ACCESS_MODE=
         SQL_MODE_READ_ONLY;
         SQL_ATTR_TRACE=
         SQL_OPT_TRACE_ON");
    // set ODBC options
    if not  dbOpen();
        // don't try to set this option
        dbSetConnectOptions(db_id, "");
        ...
    endif;
```

dbSetDatabaseName

Purpose

Sets the connection's database name to name. To have effect, the database name must be set before the connection is opened. Alternatively, you can **`dbClose()`** the connection, set the database name, and call **`dbOpen()`** again.

Format

`dbSetDatabaseName`(*`db_id, database_name`*);

Input

db_id	scalar, database connection index number.
database_name	string, database name to apply to specified database connection.

Remarks

For the OCI (Oracle) driver, the database name is the TNS Service Name.

For the ODBC driver, the name can either be a DSN, a DSN filename (in which case the file must have a .dsn extension), or a connection string.

For example, Microsoft Access users can use the following connection string to open an .mdb file directly, instead of having to create a DSN entry in the ODBC manager:

```
...
db_id = dbAddDatabase("ODBC");
dbSetDatabaseName(db_id, "DRIVER=
    {Microsoft Access Driver (*.mdb)};
    FIL={MS Access};
    DBQ=myaccessfile.mdb");
dbOpen(db_id);
...
```

See Also

dbGetDatabaseName

dbSetHostName

Purpose

Sets the specified database connection's host name.

Format

dbSetHostName(*db_id*, *host_name*);

Input

db_id	scalar, database connection index number.
host_name	string, the name to which the specified connection's host name should be assigned.

Remarks

For this function to have an effect, it must be called before the database connection is opened with **dbOpen()**.

dbSetPassword

Purpose

Sets the database connection's password.

Format

dbSetPassword(*db_id*, *pswd*);

Input

db_id	scalar, database connection index number.
pswd	string, password for database.

Remarks

This function must be called before the connection is opened with **dbOpen()** to have an effect.

See Also

dbGetPassword

dbSetPort

Purpose

Sets the specified database connection's port number.

Format

`dbSetPort`(*`db_id`*, *`port_num`*);

Input

`db_id`	scalar, database connection index number.
`port_num`	scalar, port number for database connection to use.

Remarks

This function must be called before the connection is opened with **`dbOpen()`** to have an effect.

See Also

dbGetPort

dbSetUserName

Purpose

Sets the specified database connection's user name.

Format

`dbSetUserName`(*`db_id`*, *`user_name`*);

Input

db_id	scalar, database connection index number.
user_name	string, user name to apply to specified database connection.

Remarks

This function must be called before the connection is opened with **dbOpen()** to have an effect.

See Also

dbGetUserName

dbTransaction

Purpose

Begins a transaction on the database.

Format

ret = **dbTransaction**(*db_id*);

Input

db_id	scalar, database connection index number.

Output

ret	scalar, 1 to indicate success and a 0 if the transaction fails.

Example

```
//If 'dbTransaction' succeeds
if dbTransaction(db_id);
    // All queries must succeed, or all fail.
    if not dbExecQuery(db_id,
        "INSERT INTO TEST...");
        dbRollback(db_id);
        errorlog("Query 1 failed");
        end;
    endif;

    if not dbExecQuery(db_id,
        "INSERT INTO TEST...");
        dbRollback(db_id);
        errorlog("Query 2 failed");
        end;
    endif;

    dbCommit(db_id);
endif;
```

Remarks

This function can only be used with databases that support transactions.

See Also

dbCommit, dbRollback

debug

Purpose

Runs a program under the source level debugger.

Format

```
debug filename;
```

Input

filename	Literal, name of file to debug.

Remarks

See **Debugging**, Section 1.1.

declare

Purpose

Initializes global variables at compile time.

Format

```
declare [[type]] symbol [[aop clist]];
```

Input

type	optional literal, specifying the type of the symbol.
	`matrix`
	`string`
	`array`
	`sparse matrix`
	`struct` *structure_type*
	if *type* is not specified, `matrix` is assumed. Set *type* to `string` to initialize a string or string array variable.

symbol	the name of the symbol being declared.
aop	the type of assignment to be made. `=` if not initialized, initialize. If already initialized, reinitialize. `!=` if not initialized, initialize. If already initialized, reinitialize. `:=` if not initialized, initialize. If already initialized, redefinition error. `?=` if not initialized, initialize. If already initialized, leave as is. If *aop* is specified, *clist* must be also.
clist	a list of constants to assign to *symbol*. If *aop clist* is not specified, *symbol* is initialized as a

scalar 0 or a null string.

Remarks

The `declare` syntax is similar to the `let` statement.

`declare` generates no executable code. This is strictly for compile time initialization. The data on the right-hand side of the equal sign must be constants. No expressions or variables are allowed.

`declare` statements are intended for initialization of global variables that are used by procedures in a library system.

It is best to place `declare` statements in a separate file from procedure definitions. This will prevent redefinition errors when rerunning the same program without clearing your workspace.

The optional *`aop`* and *`clist`* arguments are allowed only for declaring matrices, strings, and string arrays. When you `declare` an N-dimensional array, sparse matrix, or structure, they will be initialized as follows:

Variable Type	Initializes To
N-dimensional array	1-dimensional array of 1 containing 0
sparse matrix	empty sparse matrix
structure	structure containing empty and/or zeroed out members

Complex numbers can be entered by joining the real and imaginary parts with a sign (+ or -); there should be no spaces between the numbers and the sign. Numbers with no real part can be entered by appending an 'i' to the number.

There should be only one declaration for any symbol in a program. Multiple declarations of the same symbol should be considered a programming error. When **GAUSS** is looking through the library to reconcile a reference to a matrix or a string, it will quit looking as soon as a symbol with the correct name is found. If another symbol with the same name existed in another file, it would never be found. Only the first one in the search path would be available to programs.

Here are some of the possible uses of the three forms of declaration:

`!=`, `=`	Interactive programming or any situation where a global by the same name will probably be sitting in the symbol table when the file containing the `declare` statement is compiled. The symbol will be reset.
`:=`	Redefinition is treated as an error because you have probably just outsmarted yourself. This will keep you out of trouble because it won't allow you to zap one symbol with another value that you didn't know was getting mixed up in your program. You probably need to rename one of them.
`?=`	Interactive programming where some global defaults were set when you started and you don't want them reset for each successive run even if the file containing the `declare`'s gets recompiled. This can get you into trouble if you are not careful.

The `declare` statement warning level is a compile option. Call **`config`** in the command line version of **GAUSS** or select Preferences from the Configure menu in the graphical user interface to edit this option. If `declare` warnings are on, you will be warned whenever a `declare` statement encounters a symbol that is already initialized. Here's what happens when you declare a symbol that is already initialized when `declare` warnings are turned on:

`declare !=`	Reinitialize and warn.
`declare :=`	End program with fatal error.

declare ?= Leave as is and warn.

If `declare` warnings are off, no warnings are given for the *!=* and *?=* cases.

Example

```
declare matrix x,y,z;

x = 0   y = 0   z = 0
declare string x = "This string.";

x = "This string."

declare matrix x;

x = 0

//Initialize 'x' with the specified values and
//return a warning if 'x'already exists AND
//the 'Compile Options: declare warnings' is
//selected
declare matrix x != { 1 2 3, 4 5 6, 7 8 9 };

    1 2 3
x = 4 5 6
    7 8 9

declare matrix x[3,3] = 1 2 3 4 5 6 7 8 9;

    1 2 3
x = 4 5 6
    7 8 9

declare matrix x[3,3] = 1;
```

```
    1 1 1
x = 1 1 1
    1 1 1

declare matrix x[3,3];

    0 0 0
x = 0 0 0
    0 0 0

declare matrix x = 1 2 3 4 5 6 7 8 9;

    1
    2
    3
x = 4
    5
    6
    7
    8
    9

//Create a 2x1 character matrix
declare matrix x = alpha beta;

//To print character matrices, the '$' operator must
//be prepended to the variable name
print $x;
```

The code snippet directly above, produces:

```
ALPHA
 BETA

//Since this is declared as a matrix, the text in
```

```
//quotes will create a character vector, rather
//than a string array
declare matrix x = "mean" "variance";

print $x;
```

produces:

```
    mean
variance

declare array a;
```

a is a 1-dimensional array of 1 containing 0.

```
declare sparse matrix sm;
```

sm is an empty sparse matrix.

```
struct mystruct {
  matrix m;
  string s;
  string array sa;
  array a;
  sparse matrix sm;
 };

declare struct mystruct ms;
```

ms is a **mystruct** structure, with its members set as follows:

ms.m	empty matrix
ms.s	null string

ms.sa	1x1 string array containing a null string
ms.a	1-dimensional array of 1 containing 0
ms.sm	empty sparse matrix

See Also

let, external

delete

Purpose

Deletes global symbols from the symbol table.

Format

```
delete-flagssymbol_list;
deletesymbol_list;
```

Input

flags	specify the type(s) of symbols to be deleted	
	p	procedures
	k	keywords
	f	fn functions
	m	matrices

	s	strings
	g	only procedures with global references
	l	only procedures with all local references
	n	no pause for confirmation
symbol	literal, name of symbol to be deleted. If symbol ends in an asterisk, all symbols matching the leading characters will be deleted.	

Remarks

This completely and irrevocably deletes a symbol from **GAUSS**'s memory and workspace.

Flags must be preceded by a dash (e.g. *-pfk*). If the *n* (no pause) flag is used, you will not be asked for confirmation for each symbol.

This command is supported only from interactive level. Since the interpreter executes a compiled pseudo-code, this command would invalidate a previously compiled code image and therefore would destroy any program it was a part of. If any symbols are deleted, all procedures, keywords and functions with global references to those symbols will be deleted as well.

Example

```
    //Create a matrix 'x'
    x = { 1, 2, 3, 4 };

    //'show' returns information about active symbols
    show x;
```

This should return:

```
    32 bytes    x       MATRIX                     4,1

    delete -m x;
```

At the `Delete?[Yes No Previous Quit]` prompt, enter `y`.

```
    show x;
```

x no longer exists.

delete (dataloop)

Purpose

Removes specific rows in a data loop based on a logical expression.

Format

`delete`*`logical_expression`*;

Remarks

Deletes only those rows for which *`logical_expression`* is TRUE. Any variables referenced must already exist, either as elements of the source data set, as `extern`'s, or as the result of a previous `make`, `vector`, or **code** statement.

GAUSS expects *`logical_expression`* to return a row vector of 1's and 0's. The relational and other operators (e.g. <) are already interpreted in terms of their dot equivalents (`.`<), but it is up to the user to make sure that function calls within *`logical_expression`* result in a vector.

Example

```
    delete age < 40 or sex == 'FEMALE';
```

See Also

select (dataloop)

DeleteFile

Purpose

Deletes files.

Format

ret = **DeleteFile**(*name*);

Input

name	string or NxK string array, name of file or files to delete.

Output

ret	scalar or NxK matrix, 0 if successful.

Remarks

The return value, *ret*, is scalar if *name* is a string. If *name* is an NxK string array, `ret` will be an NxK matrix reflecting the success or failure of each separate file deletion.

`DeleteFile` calls the C library **`unlink`** function for each file. If **`unlink`** fails it sets the C library errno value. **`DeleteFile`** returns the value of errno if **`unlink`** fails, otherwise it returns zero. If you want detailed information about the reason for failure, consult the C library **`unlink`** documentation for your platform for details.

delif

Purpose

Deletes rows from a matrix. The rows deleted are those for which there is a 1 in the corresponding row of *e*.

Format

y = **`delif`**(*x*, *e*);

Input

x	NxK data matrix.
e	Nx1 logical vector (vector of 0's and 1's).

Output

y	MxK data matrix consisting of the rows of *y* for which there is a 0 in the corresponding row of *e*. If no rows remain, **`delif`** will return a scalar missing.

Remarks

The input *e* will usually be generated by a logical expression using dot operators. For instance:

```
    //Create a vector 'e' with a 1 for each row in which the
    //value in the second column of 'x' is less than 100,
    //otherwise a 0
    e = x[.,2] .> 100;

    y = delif(x, e);
```

Or the equivalent statement:

```
    y = delif(x, x[.,2] .> 100);
```

will delete all rows of *x* whose second element is greater than 100. The remaining rows of *x* will be assigned to *y*.

Example

```
    x = { 0 10 20,
         30 40 50,
         60 70 80 };

    /* logical vector */
    e =(x[.,1] .gt 0) .and (x[.,3] .lt 100);

    y = delif(x,e);
```

After the code above:

```
    y = 0 10 20
```

All rows for which the elements in column 1 are greater than 0 and the elements in column 3 are less than 100 are deleted.

See Also

selif

delrows

Purpose

Deletes rows from a matrix. The second argument contains the indices of the rows to be deleted.

Format

y = **`delrows`**(*x*, *r*);

Input

x	NxK data matrix.
r	Mx1 vector, indices of rows to delete.

Output

y	PxK matrix contaning the remaining rows of *x*. If no rows remain, *y* will be an empty matrix.

Remarks

If *r* is an empty matrix, the result will be unchanged. Negative values of *r* are counted from the end of the matrix, therefore:

```
    r = -1;

    y = delrows(x, r);
```

will delete last row of *x* . The remaining rows of *x* will be assigned to *y*.

Example

```
    x = { 0 10 20,
         30 40 50,
         32 42 52,
         35 45 55,
         60 70 80 };

    r = { 2,
          4 };

    y = delrows(x,r);
```

After the code above:

```
    y =   0 10 20
         32 42 52
         60 70 80
```

Rows 2 and 3 are deleted.

See Also

delif

denseToSp

Purpose

Converts a dense matrix to a sparse matrix.

Format

y = **denseToSp**(*x*, *eps*);

Input

x	MxN dense matrix.
eps	scalar, elements of *x* whose absolute values are less than or equal to *eps* will be treated as zero.

Output

y	MxN sparse matrix.

Remarks

A dense matrix is just a normal format matrix.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **denseToSp**.

Example

```
    //Declare 'y' as a sparse matrix
```

```
sparse matrix y;

x = { 0.01 0.00 0.01 1.00,
      0.00 4.00 0.02 0.00,
      0.00 0.01 0.00 0.00,
      0.02 0.00 -2 0.00 };

//Create a sparse matrix 'y' from 'x' and set all elements
//less than 0.04 equal to 0
y = denseToSp(x,0.04);
```

After the code above, *y* is equal to:

```
0.00   0.00   0.00   1.00
0.00   4.00   0.00   0.00
0.00   0.00   0.00   0.00
0.00   0.00  -2.00   0.00
```

See Also

spCreate, spDenseSubmat, spToDense

denseToSpRE

Purpose

Converts a dense matrix to a sparse matrix, using a relative epsilon.

Format

y = **denseToSpRE**(*x*, *reps*);

Input

x	MxN dense matrix.
reps	scalar, relative epsilon. Elements of *x* will be treated as zero if their absolute values are less than or equal to *reps* multiplied by the mean of the absolute values of the non-zero values in *x*.

Output

y	MxN sparse matrix.

Remarks

A dense matrix is just a normal format matrix.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **denseToSpRE**.

Example

```
sparse matrix y;
x = { -9   0   0    1,
       0   4   0    0,
       5   0   0    7,
       0   0  -2  2.2 };

y = denseToSpRE(x,0.5);
d = spToDense(y);
```

After the code above, *d* is equal to:

```
   -9.00    0.00    0.00    0.00
    0.00    4.00    0.00    0.00
    5.00    0.00    0.00    7.00
    0.00    0.00    0.00    2.20
```

You can calculate the mean of the non-zero elements of *x* like this:

```
//Create a matrix of 1's and 0's with a 1 where the
//corresponding element in 'x' is not equal to 0
mask = x ./= 0;

//Calculate the sum of 'mask', this is the number of
//non-zeros in 'x'
nnz = sumc(sumc(mask));

//Divide the sum of the absolute value of 'x' by the number
//of non-zeros
nzmean = sumc(sumc(abs(x)))/nnz;
```

```
nnz =         7
nzmean = 4.31
```

The call to **denseToSpRE** towards the start of this example, removed all non-zeros less than *0.5*nzmean*, or approximately 2.16.

See Also

denseToSp, spCreate, spToDense

denToZero

Purpose

Converts every denormal to a 0 in a matrix or array.

Format

y = **denToZero**(*x*);

Input

x	A matrix or an N-dimensional array.

Output

y	A matrix or an N-dimensional array with the same orders as the input. Every denormal in the input will be converted to 0 in the output.

Example

```
x = { 1, exp(-724.5), 3 };

//If 'x' contains any denormals set them to 0
if isden(x);
   x2 = denToZero(x);
endif;
```

After the first line above, *x* is equal to:

```
1.000e+000
2.902e-057
3.000e+000
```

At the end of the example, *x* is equal to:

```
    1.000e+000
    0.000e+000
    3.000e+000
```

design

Purpose

Creates a design matrix of 0's and 1's from a column vector of numbers specifying the columns in which the 1's should be placed.

Format

y = **design**(*x*);

Input

x	Nx1 vector.

Output

y	NxK matrix, where K = **maxc**(*x*); each row of *y* will contain a single 1, and the rest 0's. The one in the *i*th row will be in the **round**(*x*[*i*,1]) column.

Remarks

Note that *x* does not have to contain integers: it will be rounded to nearest if necessary.

Example

This example uses **design** to interchange the rows of a matrix.

```
//Suppress printing of digits after the decimal place
format /rd 6,0;

//Set the rng seed for repeatable random numbers
rndseed 345425235;

//Create a 4x4 matrix of random integers with a standard
//deviation of 10
x = round(10*rndn(4,4));
print x;
```

The code above returns:

```
 4      12      -1     -10
 5      -3      12       8
12      -2      21     -21
-7     -13       0      -1
```

Contintuing on with the example:

```
//The order of the rows we want
rowOrder = { 3, 1, 4, 2 };

//Create a permutation matrix from 'rowOrder'
p = design(rowOrder);
print p;
```

This section returns:

```
    0       0       1       0
    1       0       0       0
    0       0       0       1
    0       1       0       0

    //Create a permuted version of 'x' with our preferred row
    //order
    x2 = p*x;
    print x2;
```

This final section returns:

```
    12      -2      21     -21
     4      12      -1     -10
    -7     -13       0      -1
     5      -3      12       8
```

This last print statement shows us that we have indeed changed the order of the rows. In *x* the row order is 1, 2, 3, 4. However, in *x2*, the row order is 3, 1, 4, 2 (i.e. the third row is now first, the first row is now second, etc.)

Source

design.src

See Also

cumprodc, cumsumc, recserrc

det

Purpose

Returns the determinant of a square matrix.

Format

y = **`det`**(*x*);

Input

x	NxN square matrix or K-dimensional array where the last two dimensions are NxN .

Output

y	scalar or [K-2]-dimensional array, the determinant(s) of *x*.

Remarks

x may be any valid expression that returns a square matrix (number of rows equals number of columns) or a K-dimensional array where the last two dimensions are of equal size.

If *x* is a K-dimensional array, the result will be a [K-2]-dimensional array containing the determinants of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10x4x4 array, the result will be a 1-dimensional array of 10 elements containing the determinants of each of the 10 4x4 arrays contained in *x*.

`det` computes an LU decomposition.

`detl` can be much faster in many applications.

Example

```
    x = { 3 2 1,
          0 1 -2,
          1  3  4 };
    y = det(x);

    format /rd 3,0;
    print "The determinant of y =" y;
```

The code above, produces:

```
    The determinant of y = 25
```

See Also

detl

detl

Purpose

Returns the determinant of the last matrix that was passed to one of the intrinsic matrix decomposition routines.

Format

y = **detl**;

Remarks

Whenever one of the intrinsic matrix decomposition routines is executed, the determinant of the matrix is also computed and stored in a system variable. This function will return the value of that determinant and, because the value has been computed in a previous instruction, this will require no computation.

The following functions will set the system variable used by **detl**:

chol(*x*)	
crout(*x*)	
croutp(*x*)	
det(*x*)	
inv(*x*)	
invpd(*x*)	
solpd(*y*, *x*)	determinant of *x*

Example

If both the inverse and the determinant of the matrix are needed, the following two commands will return both with the minimum amount of computation:

```
xi = inv(x);
xd = detl;
```

The function **det**(*x*) returns the determinant of a matrix using the Crout decomposition. If you only want the determinant of a positive definite matrix, the following code will be the fastest for matrices larger than 10x10:

```
//The 'call' keyword tells GAUSS to ignore the values
//returned from chol
call chol(x);
xd = detl;
```

The Cholesky decomposition is computed and the result from that is discarded. The determinant saved during that instruction is retrieved using **`detl`**. This can execute up to 2.5 times faster than **`det`**(*x*) for large positive definite matrices.

See Also

det

dfft

Purpose

Computes a discrete Fourier transform.

Format

y = **`dfft`**(*x*);

Input

x	Nx1 vector.

Output

y	Nx1 vector.

Remarks

The transform is divided by N.

This uses a second-order Goertzel algorithm. It is considerably slower than **`fft`**, but it may have some advantages in some circumstances. For one thing, N does not have to be an even power of 2.

Source

`dfft.src`

See Also

dffti, fft, ffti

dffti

Purpose

Computes inverse discrete Fourier transform.

Format

y = **`dffti`**(*x*);

Input

x	Nx1 vector.

Output

y	Nx1 vector.

Remarks

The transform is divided by N.

This uses a second-order Goertzel algorithm. It is considerably slower than **`ffti`**, but it may have some advantages in some circumstances. For one thing, N does not have to be an even power of 2.

Source

`dffti.src`

See Also

fft, dffti, ffti

diag

Purpose

Creates a column vector from the diagonal of a matrix.

Format

y = **`diag`**(*x*);

Input

x	NxK matrix or L-dimensional array where the last two dimensions are NxK.

Output

y	min(N,K)x1 vector or L-dimensional array where the last two dimensions are min(N,K)x1.

Remarks

If *x* is a matrix, it need not be square. Otherwise, if *x* is an array, the last two dimensions need not be equal.

If *x* is an array, the result will be an array containing the diagonals of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10x4x4 array, the result will be a 10x4x1 array containing the diagonals of each of the 10 4x4 arrays contained in *x*.

diagrv reverses the procedure and puts a vector into the diagonal of a matrix.

Example

```
    x = rndu(3,3);
    y = diag(x);
```

```
        0.28330575     0.17602494     0.11521377      0.28330575
    x = 0.46050011     0.36578753     0.99670527
    y = 0.36578753
        0.58591859     0.39876576     0.94153871      0.94153871
```

```
    x = rndn(48,1);

    //Reshape the 48x1 vector into a 3x4x4 dimensional array
    x = reshape(x, 3|4|4);
    d = diag(x);
```

Now *x* is equal to:

```
    Plane [1,.,.]

      0.082720153    -0.49502230    -0.40613944      1.9283280
       0.23583965    -0.24230946    -0.66047073    -0.73098141
       -1.1187279    -0.27867822     -1.7846293    -0.44603382
      0.030071777     -1.0387861     0.23768949    0.019151917

    Plane [2,.,.]

       -1.7238416     0.17660645    -0.14798006    0.072065419
        1.3685721    -0.11216325    -0.12985589      1.1816008
       0.63154571     -1.4945397     -1.7276380    -0.28275797
      -0.71832623     -1.3193506    -0.53934998    -0.78348484

    Plane [3,.,.]

      -0.71111209    -0.30818842    -0.38982318     -2.7205066
       -1.5455077    -0.27131853     0.98686691     0.10870999
       0.57916876      1.8180884     0.76104693      1.1237605
        1.0727710     -1.1071168      1.7443178     -1.0684433
```

and d is a 3x4x1 array containing the diagonals from x above.

```
    Plane [1,.,.]

         0.082720153
         -0.24230946
          -1.7846293
         0.019151917

    Plane [2,.,.]

          -1.7238416
         -0.11216325
          -1.7276380
```

```
        -0.78348484

    Plane [3,.,.]

        -0.71111209
        -0.27131853
         0.76104693
         -1.0684433
```

See Also

diagrv

diagrv

Purpose

Inserts a vector into the diagonal of a matrix.

Format

y = **diagrv**(*x*, *v*);

Input

x	NxK matrix.
v	min(N,K)x1 vector.

Output

y	NxK matrix equal to *x* with its principal diagonal elements equal to those of *v*.

Remarks

diag reverses the procedure and pulls the diagonal out of a matrix.

Example

```
    x = rndu(3,3);
    v = ones(3,1);
    y = diagrv(x,v);
```

After the code above:

```
        0.614 0.686 0.633     1.000     1.000 0.686 0.633
    x = 0.802 0.185 0.707 v = 1.000 y = 0.802 1.000 0.707
        0.551 0.761 0.418     1.000     0.551 0.761 1.000
```

See Also

diag

digamma

Purpose

Computes the digamma function.

Format

y = **digamma**(*x*);

Input

x	MxN matrix or N-dimensional array.

Output

y	MxN matrix or N-dimensional array, digamma.

Remarks

The digamma function is the first derivative of the log of the gamma function with respect to its argument.

dlibrary

Purpose

Dynamically links and unlinks shared libraries.

Format

```
dlibrary lib1 [[lib2]]...;
dlibrary -alib1 [[lib2]]...;
dlibrary -d;
dlibrary;
```

Input

lib1 lib2...	literal, the base name of the library or the pathed name of the library. `dlibrary` takes two types of arguments, "base" names and file names. Arguments without any "/" path separators are assumed to be library base names, and are expanded by adding the suffix `.so`, `.dll` or `.dylib`, depending on the platform. They are searched for in the default dynamic library directory. Arguments that include "/" path separators are assumed to be file names, and are not expanded. Relatively pathed file names are assumed to be specified relative to the current working directory, not relative to the dynamic library directory.
-a	append flag, the shared libraries listed are added to the current set of shared libraries rather than replacing them. For search purposes, the new shared libraries follow the already active ones. Without the *-a* flag, any previously linked libraries are dumped.
-d	dump flag, ALL shared libraries are unlinked and the functions they contain are no longer available to your programs. If you use `dllcall` to call one of your functions after executing a `dlibrary -d` your program will terminate with an error.

Remarks

If no flags are used, the shared libraries listed are linked into **GAUSS** and any previously linked libraries are dumped. When you call `dllcall`, the shared libraries will be searched in the order listed for the specified function. The first instance of the function found will be called.

`dlibrary` with no arguments prints out a list of the currently linked shared libraries. The order in which they are listed is the order in which they are searched for functions.

`dlibrary` recognizes a default directory in which to look for dynamic libraries. You can specify this by setting the variable *`dlib_path`* in `gauss.cfg`. Set it to point to a single directory, not a sequence of directories. **`sysstate`**, case 24, may also be used to get and set this default.

GAUSS maintains its own shared libraries which are listed when you execute `dlibrary` with no arguments, and searched when you call `dllcall`. The default shared library or libraries are searched last. You can force them to be searched earlier by listing them explicitly in a `dlibrary` statement. They are always active and are not unlinked when you execute

```
    dlibrary -d
```

For more information, see **Foreign Language Interface**, Chapter 1.

See Also

dllcall, sysstate

dllcall

Purpose

Calls functions located in dynamic libraries.

Format

`dllcall` [-*r*] [-*v*] **func**(*arg1...argN*);

`dllcall` works in conjunction with `dlibrary`. `dlibrary` is used to link shared libraries into **GAUSS**; `dllcall` is used to access the functions contained in those shared libraries. `dllcall` searches the shared libraries (see `dlibrary` for an explanation of the search order) for a function named *func*, and calls the first instance it finds. The default shared libraries are searched last.

Input

func	the name of a function contained in a shared library (linked into **GAUSS** with `dlibrary`). If *func* is not specified or cannot be located in a shared library, `dllcall` will fail.
arg#	arguments to be passed to *func*, optional. These must be simple variable references; they cannot be expressions.
-*r*	optional flag. If -*r* is specified, `dllcall` examines the value returned by *func*, and fails if it is nonzero.
-*v*	optional flag. Normally, `dllcall` passes parameters to *func* in a list. If -*v* is specified, `dllcall` passes them in a vector. See below for more details.

Remarks

func should be written to:

1. Take 0 or more pointers to doubles as arguments.
2. Take arguments either in a list or a vector.
3. Return an integer.

In C syntax, *func* should take one of the following forms:

1. *int* **func**(void);
2. *int* **func**(double **arg1 [, arg2...argN]);*
3. *int* **func**(double **arg*[]);

`dllcall` can pass a list of up to 100 arguments to **func**; if it requires more arguments than that, you MUST write it to take a vector of arguments, and you MUST specify the *-v* flag when calling it. `dllcall` can pass up to 1000 arguments in vector format. In addition, in vector format `dllcall` appends a null pointer to the vector, so you can write *func* to take a variable number of arguments and just test for the null pointer.

Arguments are passed to *func* by reference. This means you can send back more than just the return value, which is usually just a success/failure code. (It also means that you need to be careful not to overwrite the contents of matrices or strings you want to preserve.) To return data from **func**, simply set up one or more of its arguments as return matrices (basically, by making them the size of what you intend to return), and inside **func** assign the results to them before returning.

For more information, see **Foreign Language Interface**, Chapter 1.

See Also

dlibrary, sysstate

do while,do until

Purpose

Executes a series of statements in a loop as long as a given expression is true (or false).

Format

```
    do while expression;
    or
    do until expression;
     .
     .
     .
      statements in loop
     .
     .
     .
    endo;
```

Remarks

expression is any expression that returns a scalar. It is TRUE if it is nonzero and FALSE if it is zero.

In a `do while` loop, execution of the loop will continue as long as the expression is TRUE.

In a `do until` loop, execution of the loop will continue as long as the expression is FALSE.

The condition is checked at the top of the loop. If execution can continue, the statements of the loop are executed until the `endo` is encountered. Then **GAUSS** returns to the top of the loop and checks the condition again.

The `do` loop does not automatically increment a counter. See the first example below.

`do` loops may be nested.

It is often possible to avoid using loops in **GAUSS** by using the appropriate matrix operator or function. It is almost always preferable to avoid loops when possible, since the corresponding matrix operations can be much faster.

Example

```
format /rdn 1,0;
space = "  ";
comma = ",";
i = 1;
do while i <= 4;
   j = 1;
   do while j <= 3;
      print space i comma j;;
      j = j+1;
   endo;
   i = i+1;
   print;
endo;
```

The code above prints the following output:

```
1,1  1,2  1,3
2,1  2,2  2,3
3,1  3,2  3,3
4,1  4,2  4,3
```

In the example above, two nested loops are executed and the loop counter values are printed out. Note that the inner loop counter must be reset inside of the outer loop before entering the inner loop. An empty `print` statement is used to print a carriage return/line feed sequence after the inner loop finishes.

The following are examples of simple loops that execute a predetermined number of times. These loops will both have the result shown.

First loop:

```
    format /rd 1,0;
    i = 1;
    do while i <= 10;
        print i;;
        i = i+1;
    endo;
```

produces:

```
    1 2 3 4 5 6 7 8 9 10
```

Second loop:

```
    format /rd 1,0;
       i = 1;
       do until i > 10;
           print i;;
           i = i+1;
      endo;
```

produces:

```
    1 2 3 4 5 6 7 8 9 10
```

See Also

continue, break

dos

Purpose

Provides access to the operating system from within **GAUSS**.

Format

```
    dos commd;
```

Input

commd	literal or ^string, the OS command to be executed.

Portability

UNIX/Linux

Control and output go to the controlling terminal, if there is one.

This function may be used in terminal mode.

Windows

The `dos` function opens a new terminal.

Running programs in the background is allowed on both of the aforementioned platforms.

Remarks

This allows all operating system commands to be used from within **GAUSS**. It allows other programs to be run even though **GAUSS** is still resident in memory.

If no operating system command (for instance, **`dir`** or **`copy`**) or program name is specified, then a shell of the operating system will be entered which can be used just like the base level OS. The **`exit`** command must be given from the shell to get back into **GAUSS**. If a command or program name is included, the return to **GAUSS** is automatic after the OS command has been executed.

All matrices are retained in memory when the OS is accessed in this way. This command allows the use of word processing, communications, and other programs from within **GAUSS**.

Do not execute programs that terminate and remain resident because they will be left resident inside of **GAUSS**'s workspace. Some examples are programs that create RAM disks or print spoolers.

If the command is to be taken from a string variable, the ^ (caret) must precede the string.

The shorthand ">" can be used in place of "dos".

Example

```
     cmdstr = "atog mycfile";
     dos ^cmdstr;
```

This will run the ATOG utility, using `mycfile.cmd` as the ATOG command file. For more information, see **ATOG**, Chapter 1.

```
     > dir *.prg;
```

This will use the DOS **`dir`** command to print a directory listing of all files with a `.prg` extension on Windows. When the listing is finished, control will be returned to **GAUSS**.

```
     > ls *.prg
```

This will perform the same operation on UNIX/Linux.

```
     dos;
```

This will cause a second level OS shell to be entered. The OS prompt will appear and OS commands or other programs can be executed. To return to **GAUSS**, type **`exit`**.

See Also

exec

doswin

Purpose

Opens the DOS compatibility window with default settings. NOTE: This function is no longer supported. This documentation is provided as a reference for understanding legacy code. In many cases, you may simply comment out calls to **`doswin`** and the program will run successfully in the program input/output window.

Format

```
doswin;
```

Portability

Windows only

Remarks

Calling **`doswin`** is equivalent to:

```
call DOSWinOpen("", error(0));
```

Source

`gauss.src`

DOSWinCloseall

Purpose

Closes the DOS compatibility window. NOTE: The DOS compatibility window is no longer supported. This documenation is provided as a reference for understanding legacy code.

Format

```
DOSWinCloseall;
```

Portability

Windows only

Remarks

Calling **DOSWinCloseall** closes the DOS window immediately, without asking for confirmation. If a program is running, its I/O reverts to the Command window.

Example

```
let attr = 50 50 7 0 7;
if not DOSWinOpen("Legacy Window", attr);
   errorlog "Failed to open DOS window, aborting";
   stop;
endif;
 .
 .
 .
DOSWinCloseall;
```

DOSWinOpen

Purpose

Opens the DOS compatibility window and gives it the specified title and attributes. NOTE: This function is no longer supported. This documentation is provided as a reference for understanding legacy code. In many cases, you may simply comment out calls to **`DOSWinOpen`** and the program will run successfully in the program input/output window.

Format

`ret` = **`DOSWinOpen`**(*`title`*, *`attr`*);

Input

`title`	string, window title.		
`attr`	5x1 vector or scalar missing, window attributes.		
	`[1]`	window x position	
	`[2]`	window y position	
	`[3]`	text foreground color	
	`[4]`	text background color	
	`[5]`	close action bit flags	
		bit 0 (1's bit)	issue dialog
		bit 1 (2's bit)	close window
		bit 2 (4's bit)	stop program

Output

ret	scalar, success flag, 1 if successful, 0 if not.

Portability

Windows only

Remarks

If *title* is a null string (""), the window will be titled "GAUSS-DOS".

Defaults are defined for the elements of *attr*. To use the default, set an element to a missing value. Set *attr* to a scalar missing to use all defaults. The defaults are defined as follows:

[1]	varies	use x position of previous DOS window
[2]	varies	use y position of previous DOS window
[3]	7	white foreground
[4]	0	black background
[5]	6	4+2: stop program and close window without confirming

If the DOS window is already open, the new *title* and *attr* will be applied to it. Elements of *attr* that are missing are not reset to the default values, but are left as is.

To set the close action flags value (*attr*[5]), just sum the desired bit values. For example:

stop program (4) + close window (2) + confirm close (1) = 7

The close action flags are only relevant when a user attempts to interactively close the DOS window while a program is running. If **GAUSS** is idle, the window will be closed immediately. Likewise, if a program calls **`DOSWinCloseall`**, the window is closed, but the program does not get terminated.

Example

```
let attr = 50 50 7 0 7;

if not DOSWinOpen("Legacy Window", attr);
   errorlog "Failed to open DOS window, aborting";
   stop;
endif;
```

This example opens the DOS window at screen location (50,50), with white text on a black background. The close action flags are 4 + 2 + 1 (stop program + close window + issue confirm dialog) = 7. Thus, if the user attempts to close the window while a program is running, he/she will be asked for confirmation. Upon confirmation, the window will be closed and the program terminated.

dotfeq,dotfge, dotfgt,dotfle,dotflt,dotfne

Purpose

Fuzzy comparison functions. These functions use *`_fcmptol`* to fuzz the comparison operations to allow for roundoff error.

Format

```
y = dotfeq(a, b);
y = dotfge(a, b);
y = dotfgt(a, b);
y = dotfle(a, b);
y = dotflt(a, b);
y = dotfne(a, b);
```

Input

a	NxK matrix, first matrix.
b	LxM matrix, second matrix, ExE compatible with *a*.

Global Input

_fcmptol	scalar, comparison tolerance. The default value is 1.0e-15.

Output

y	max(N,L) by max(K,M) matrix of 1's and 0's.

Remarks

The return value is 1 if TRUE and 0 if FALSE.

The statement:

```
    y = dotfeq(a,b);
```

is equivalent to:

```
    y = a .eq b;
```

The calling program can reset *_fcmptol* before calling these procedures:

```
    _fcmptol = 1e-12;
```

Example

```
    x = pi*ones(2,2);
    y = x;
    y[1,1] = 2*pi;

    //Test for elements where 'x' is > 'y'
    t = dotfge(x,y);

    x = 3.14 3.14  y = 6.28 3.14  t = 0.00 1.00
        3.14 3.14      3.14 3.14      1.00 1.00
```

Continuing with the data above:

```
    //Test for elements where 'x' is < 'y '
    t = dotflt(x,y);

    t = 1.00 0.00
        0.00 0.00
```

Source

fcompare.src

Globals

_fcmptol

See Also

feq-fne

dotfeqmt, dotfgemt,dotfgtmt,dotflemt,dotfltmt,dotfnemt

Purpose

Fuzzy comparison functions. These functions use the *fcmptol* argument to fuzz the comparison operations to allow for roundoff error.

Format

```
y = dotfeqmt(a, b, fcmptol);
y = dotfgemt(a, b, fcmptol);
y = dotfgtmt(a, b, fcmptol);
y = dotflemt(a, b, fcmptol);
y = dotfltmt(a, b, fcmptol);
y = dotfnemt(a, b, fcmptol);
```

Input

a	NxK matrix, first matrix.
b	LxM matrix, second matrix, ExE compatible with *a*.
fcmptol	scalar, comparison tolerance.

Output

y	max(N,L) by max(K,M) matrix of 1's and 0's.

Remarks

The return value is 1 if TRUE and 0 if FALSE.

The statement:

```
    y = dotfeqmt(a,b,1e-13);
```

is equivalent to:

```
    y = a .eq b;
```

Example

```
    x = rndu(2,2);
    y = x;
    y[1,1] = y[1,1] + 0.00000002;
    t = dotfgemt(x,y,1e-15);
```

```
    t = 0 1    x-y = -2e-8    0
        1 1              0    0
```

Source

`fcomparemt.src`

See Also

feqmt-fnemt

draw

Purpose

Graphs lines, symbols, and text using the PQG global variables. This procedure does not require actual X, Y, or Z data since its main purpose is to manually build graphs using *_pline*, *_pmsgctl*, *_psym*, *_paxes*, *_parrow* and other globals.

NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

draw;

Remarks

draw is especially useful when used in conjunction with transparent windows.

Example

```
library pgraph;
graphset;

begwind;
makewind(9,6.855,0,0,0); /* make full size window for
                            plot */
makewind(3,1,3,3,0);     /* make small overlapping window
```

```
                                          for text */
        setwind(1);
            x = seqa(.1,.1,100);
            y = sin(x);
            xy(x,y);                     /* plot data in first window */
        nextwind;
            _pbox = 15;
            _paxes = 0;
            _pnum = 0;
            _ptitlht = 1;
            margin(0,0,2,0);
            title("This is a text window.");
            draw;                         /* add a smaller text window */
        endwind;                           /* create graph */
```

Source

pdraw.src

See Also

window, makewind

drop (dataloop)

Purpose

Specifies columns to be dropped from the output data set in a data loop.

Format

```
drop variable_list;
```

Remarks

Commas are optional in *variable_list*.

Deletes the specified variables from the output data set. Any variables referenced must already exist, either as elements of the source data set, or as the result of a previous `make`, `vector`, or **code** statement.

If neither `keep` nor `drop` is used, the output data set will contain all variables from the source data set, as well as any defined variables. The effects of multiple `keep` and `drop` statements are cumulative.

Example

```
    drop age, pay, sex;
```

See Also

keep (dataloop)

dsCreate

Purpose

Creates an instance of a structure of type **DS** set to default values.

Include

`ds.sdf`

Format

```
    s = dsCreate;
```

Output

s	instance of structure of type **DS**.

Example

```
//Define 'DS' structure definition
#include ds.sdf;

//Declare 'myData' as instance of 'DS' structure
struct DS myData;

//Apply default settings
myData = dsCreate;
```

Source

ds.src

dstat

Purpose

Computes descriptive statistics.

Format

{ *vnam*, *mean*, *var*, *std*, *min*, *max*, *valid*, *mis* } = **dstat** (*dataset*, *vars*);

Input

dataset	string, name of data set. If *dataset* is null or 0, *vars* will be assumed to be a matrix containing the data.
vars	the variables. If *dataset* contains the name of a **GAUSS** data set, *vars* will be interpreted as: Kx1 character vector, names of variables. - or - Kx1 numeric vector, indices of variables. These can be any size subset of the variables in the data set and can be in any order. If a scalar 0 is passed, all columns of the data set will be used. If *dataset* is null or 0, *vars* will be interpreted as: NxK matrix, the data on which to compute the descriptive statistics.

Global Input

__*altnam*	matrix, default 0. This can be a Kx1 character vector of alternate variable names for the output.
__*maxbytes*	scalar, the maximum number of bytes to be read per iteration of the read loop. Default = 1e9.

`__maxvec`	scalar, the largest number of elements allowed in any one matrix. Default = 20000.	
`__miss`	scalar, default 0.	
	0	there are no missing values (fastest).
	1	listwise deletion, drop a row if any missings occur in it.
	2	pairwise deletion.
`__row`	scalar, the number of rows to read per iteration of the read loop.	
	if 0, (default) the number of rows will be calculated using `__maxbytes` and `__maxvec`.	
`__output`	scalar, controls output, default 1.	
	1	print output table.
	0	do not print output.

Output

`vnam`	Kx1 character vector, the names of the variables used in the statistics.
`mean`	Kx1 vector, means.
`var`	Kx1 vector, variance.
`std`	Kx1 vector, standard deviation.
`min`	Kx1 vector, minima.

max	Kx1 vector, maxima.
valid	Kx1 vector, the number of valid cases.
mis	Kx1 vector, the number of missing cases.

Example

```
//Calculate statistics on all variables in dataset: AGE, PA
Y, SEX and WT
vars = 0;
{ vnam, mean, var, std, min, max, valid, mis } = dstat("fr
eqdata.dat", vars);


//Calculate statistics on just AGE and PAY
vars = { AGE, PAY };
{ vnam, mean, var, std, min, max, valid, mis } = dstat("fr
eqdata.dat", vars);

//Calculate statistics on just AGE and PAY using numerical
indices
vars = { 1, 2 };
{ vnam, mean, var, std, min, max, valid, mis } = dstat("fr
eqdata.dat", vars);
```

Remarks

If pairwise deletion is used, the minima and maxima will be the true values for the valid data. The means and standard deviations will be computed using the correct number of valid observations for each variable.

Source

`dstat.src`

dstatmt

Purpose

Compute descriptive statistics.

Format

```
dout = dstatmt(dc0, dataset, vars);
```

Input

dc0	instance of a **dstatmtControl** structure containing the following members:	
	dc0.altnames	Kx1 string array of alternate variable names to be used if a matrix in memory is analyzed (i.e., *dataset* is a null string or 0). Default = "".
	dc0.maxbytes	scalar, the maximum number of bytes to be read per iteration of the read loop. Default = 1e9.
	dc0.vartype	Scalar, unused in **dstatmt**.
	dc0.miss	scalar, default 0.
		0 there are no missing values

(fastest).

1	listwise deletion, drop a row if any missings occur in it.
2	pairwise deletion.

dc0.row scalar, the number of rows to read per iteration of the read loop.

If 0, (default) the number of rows will be calculated using *dc0.maxbytes* and **maxvec**.

dc0.output scalar, controls output, default 1.

1	print output table.
0	do not print output.

dataset string, name of data set.

If *dataset* is null or 0, *vars* will be assumed to be a matrix containing the data.

vars the variables.

If *dataset* contains the name of a **GAUSS** data set, *vars* will be interpreted as:

Kx1 string array, names of variables.

- or -

Kx1 numeric vector, indices of variables.

These can be any size subset of the variables in the data set and can be in any order. If a scalar 0 is passed, all columns of the data set will be used.

If `dataset` is null or 0, `vars` will be interpreted as:

NxK matrix, the data on which to compute the descriptive statistics.

Output

dout	instance of a **dstatmtOut** structure containing the following members:	
	dout.vnames	Kx1 string array, the names of the variables used in the statistics.
	dout.mean	Kx1 vector, means.
	dout.var	Kx1 vector, variance.
	dout.std	Kx1 vector, standard deviation.
	dout.min	Kx1 vector, minima.
	dout.max	Kx1 vector, maxima.
	dout.valid	Kx1 vector, the number of valid cases.
	dout.missing	Kx1 vector, the number of missing cases.
	dout.errcode	scalar, error code, 0 if successful; otherwise, one of the following: 2 Can't open file. 7 Too many missings - no data left after packing.

9	`altnames` member of **dstatmtControl** structure wrong size.
10	`vartype` member of **dstatmtControl** structure wrong size.

Remarks

If pairwise deletion is used, the minima and maxima will be the true values for the valid data. The means and standard deviations will be computed using the correct number of valid observations for each variable.

Example

Example 1: Computing statistics on a GAUSS dataset

The `examples` directory contains a **GAUSS** dataset entitled `freqdata.dat`. This example will compute descriptive statistics on this file.

```
#include dstatmt.sdf
struct dstatmtControl d0;
struct dstatmtOut dout;

d0 = dstatmtControlCreate();

//Placing a '0' in for varnames will tell dstatmt to
//compute statistics for all variables in the dataset
dout = dstatmt(d0, "freqdata.dat", 0);
```

The above example will compute statistics for all variables in the dataset. If you run this code, you will see a printout of the statistics. You will also see that this dataset contains the variables AGE, PAY, SEX and WT. If you just wanted to compute statistics for the second variable, PAY, change the last line above to:

```
    dout = dstatmt(d0, "freqdata.dat", "PAY");
```

or:

```
    dout = dstatmt(d0, "freqdata.dat", 2);
```

If we wanted to compute statistics for AGE and WT, you could:

```
    //The '$|' operator performs vertical concatenation of
    //strings
    varnames = "AGE"$|"WT";
    dout = dstatmt(d0, "freqdata.dat", varnames);
```

Example 2: Computing statistics on a matrix

```
    #include dstatmt.sdf
    struct dstatmtControl d0;
    struct dstatmtOut dout;
    d0 = dstatmtControlCreate();

    //Create a random matrix on which to compute statistics
    A = rndn(10,3);

    //The empty string as the second input tells GAUSS to
    //compute statistics on a matrix rather than a dataset
    dout = dstatmt(d0, "", A);
```

You can specify custom variable names for the printout by setting `d0.altnames`. Continuing with the data from above:

```
    d0.altnames = "Alpha"$|"Beta"$|"Gamma";
    dout = dstatmt(d0, "", A);
```

Source

`dstatmt.src`

See Also

dstatmtControlCreate

dstatmtControlCreate

Purpose

Creates default **dstatmtControl** structure.

Include

```
dstatmt.sdf
```

Format

c = **dstatmtControlCreate**();

Output

c	instance of **dstatmtControl** structure with members set to default values.

Example

```
//Declare 'dsm' as an instance of a
//'dstatmtControl' structure
struct dstatmtControl dsm;

//Apply default values to 'dsm'
dsm = dstatmtControlCreate();
```

Source

`dstatmt.src`

See Also

dstatmt

dtdate

Purpose

Creates a matrix in DT scalar format.

Format

```
dt = dtdate(year, month, day, hour, minute, second);
```

Input

year	NxK matrix of years.
month	NxK matrix of months, 1-12.
day	NxK matrix of days, 1-31.
hour	NxK matrix of hours, 0-23.
minute	NxK matrix of minutes, 0-59.
second	NxK matrix of seconds, 0-59.

Output

dt	NxK matrix of DT scalar format dates.

Remarks

The arguments must be ExE conformable.

Source

`time.src`

See Also

dtday, dttime, utctodt, dttostr

dtday

Purpose

Creates a matrix in DT scalar format containing only the year, month and day. Time of day information is zeroed out.

Format

dt = **dtday**(*year*, *month*, *day*);

Input

year	NxK matrix of years.

month	NxK matrix of months, 1-12.
day	NxK matrix of days, 1-31.

Output

dt	NxK matrix of DT scalar format dates.

Remarks

This amounts to 00:00:00 or midnight on the given day. The arguments must be ExE conformable.

Source

`time.src`

See Also

dttime, dtdate, utctodt, dttostr

dttime

Purpose

Creates a matrix in DT scalar format containing only the hour, minute and second. The date information is zeroed out.

Format

dt = **dttime**(*hour*, *minute*, *second*);

Input

hour	NxK matrix of hours, 0-23.
minute	NxK matrix of minutes, 0-59.
second	NxK matrix of seconds, 0-59.

Output

dt	NxK matrix of DT scalar format times.

Remarks

The arguments must be ExE conformable.

Source

`time.src`

See Also

dtday, dtdate, utctodt, dttostr

dttodtv

Purpose

Converts DT scalar format to DTV vector format.

Format

```
dtv = dttodtv(dt);
```

Input

dt	Nx1 vector, DT scalar format.

Output

dtv	Nx8 matrix, DTV vector format.

Remarks

In DT scalar format, 15:10:55 on July 3, 2005 is 20050703151055.

Each row of `dtv`, in DTV vector format, contains:

`[N,1]`	Year
`[N,2]`	Month in Year, 1-12
`[N,3]`	Day of month, 1-31
`[N,4]`	Hours since midnight, 0-23
`[N,5]`	Minutes, 0-59
`[N,6]`	Seconds, 0-59
`[N,7]`	Day of week, 0-6, 0 = Sunday
`[N,8]`	Days since Jan 1 of current year, 0-365

Example

```
dt = 20100326110722;
print "dt = " dt;
```

```
20100326110722
```

```
dtv = dttodtv(dt);
print "dtv = " dtv;
```

```
2010 3 26 11 7 22 1 84
```

Source

`time.src`

See Also

dtvnormal, timeutc, utctodtv, dtvtodt, dttoutc, dtvtodt, strtodt, dttostr

dttostr

Purpose

Converts a matrix containing dates in DT scalar format to a string array.

Format

sa = **dttostr**(*x*, *fmt*);

Input

x	NxK matrix containing dates in DT scalar format.
fmt	string, or ExE conformable string array containing date/time format characters.

Output

sa	NxK string array.

Remarks

The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number

```
    20120703105031
```

represents 10:50:31 or 10:50:31 AM on July 3, 2012. **dttostr** converts a date in DT scalar format to a character string using the format string in *fmt*.

The following formats are supported:

YYYY	4 digit year
YR	Last two digits of year
QQ	Quarter of the year. This is calculated from the month number.
MO	Number of month, 01-12

`DD`	Day of month, 01-31
`HH`	Hour of day, 00-23
`MI`	Minute of hour, 00-59
`SS`	Second of minute, 00-59

A complete DT scalar format number will have 14 digits all to the left of the decimal point. However, **`dttostr`** will accept numbers with fewer digits. It will assume that the first four digits are the year, the next two the month and so on.

Example

Example 1

```
dt = 201202;
print dttostr(dt, "QQ-YYYY");
```

produces the output:

```
Q1-2012
```

Example 2

```
S0 = dttostr(utctodt(timeutc), "YYYY-MO-DD HH:MI:SS");
print ("Date and Time are: " $+ s0);
```

produces the output:

```
Date and time are: 2012-09-14 11:49:10
```

Example 3

```
print dttostr(utctodt(timeutc), "Today is DD-MO-YR");
```

produces the output:

```
     Today is 14-09-12
```

Example 4

```
    x = { 19120317060424, 19370904010928, 19510221031129 };
    s = dttostr(x, "YYYY-MO-DD");
```

produces s equal to:

```
    1912-03-17
    1937-09-04
    1951-02-21
```

Using the same x from above:

```
    s = dttostr(x, "DD/MO/YYYY");
```

produces s equal to:

```
    03/17/1912
    09/04/1937
    02/21/1951
```

Continuing with the same x from above:

```

    string fmt = { "YYYY-QQ", "YYYY-QQ-DD", "DD/MO/YYYY" };
    s = dttostr(x, fmt);
```

produces s equal to:

```
    1912-Q1
    1937-Q3-04
    21/02/1951
```

See Also

strtodt, dttoutc, utctodt

dttoutc

Purpose

Converts DT scalar format to UTC scalar format.

Format

utc = **dttoutc**(*dt*);

Input

dt	Nx1 vector, DT scalar format.

Output

utc	Nx1 vector, UTC scalar format.

Remarks

In DT scalar format, 10:50:31 on July 15, 2010 is 20100703105031. A UTC scalar gives the number of seconds since or before January 1, 1970 Greenwich Mean Time.

Example

```
    dt = 20010326085118;
```

```
tc = dttoutc(dt);

print "tc = " tc;
```

The above code produces the following output:

```
 tc = 985633642;
```

Source

time.src

See Also

dtvnormal, timeutc, utctodtv, dttodtv, dtvtodt, dtvtoutc, dtvtodt, strtodt, dttostr

dtvnormal

Purpose

Normalizes a date and time (DTV) vector.

Format

d = **dtvnormal**(*t*);

Input

t	1x8 date and time vector that has one or more elements outside the normal range.

Output

d	Normalized 1x8 date and time vector.

Remarks

The date and time vector is a 1x8 vector whose elements consist of:

`Year`	Year, four digit integer.
`Month`	1-12, Month in year.
`Day`	1-31, Day of month.
`Hour`	0-23, Hours since midnight.
`Min`	0-59, Minutes.
`Sec`	0-59, Seconds.
`DoW`	0-6, Day of week, 0 = Sunday.
`DiY`	0-365, Days since Jan 1 of year.

On input missing values are treated as zeros and the last two elements are ignored.

Example

```
format /rd 4,0;

dStart = { 2011 08 21 6 21 37 0 0 };
mnth = { 0 1 0 0 0 0 0 0 };
```

```
    //Add 6 months to 'dStart' which will give a 14 for the
    //month
    dEnd = dStart + 6*mnth;

    //Normalize the date vector
    dEnd2 = dtvnormal(dEnd);
```

After the code above:

```
    dEnd  = 2011   14   21    6   21   37    0    0
    dEnd2 = 2012    2   21    6   21   37    2   51
```

See Also

date, ethsec, etstr, time, timestr, timeutc, utctodtv

dtvtodt

Purpose

Converts DT vector format to DT scalar format.

Format

dt = **dtvtodt**(*dtv*);

Input

dtv	Nx8 matrix, DTV vector format.

Output

dt	Nx1 vector, DT scalar format.

Remarks

In DT scalar format, 11:06:47 on March 15, 2012 is 20120315110647.

Each row of *dtv*, in DTV vector format, contains:

[N,1]	Year
[N,2]	Month in Year, 1-12
[N,3]	Day of month, 1-31
[N,4]	Hours since midnight, 0-23
[N,5]	Minutes, 0-59
[N,6]	Seconds, 0-59
[N,7]	Day of week, 0-6, 0 = Sunday
[N,8]	Days since Jan 1 of current year, 0-365

Example

```
let dtv = { 2012 9 16 11 7 22 1 84 };
dt = dtvtodt(dtv);
```

The code above assigns *dt* as follows:

```
    20120916110722
```

Source

`time.src`

See Also

dtvnormal, timeutc, utctodtv, dttodtv, dttoutc, strtodt, dttostr

dtvtoutc

Purpose

Converts DTV vector format to UTC scalar format.

Format

utc = **dtvtoutc**(*dtv*);

Input

dtv	Nx8 matrix, DTV vector format.

Output

utc	Nx1 vector, UTC scalar format.

Remarks

A UTC scalar gives the number of seconds since or before January 1, 1970 Greenwich Mean Time.

Each row of `dtv`, in DTV vector format, contains:

[N,1]	Year
[N,2]	Month in Year, 1-12
[N,3]	Day of month, 1-31
[N,4]	Hours since midnight, 0-23
[N,5]	Minutes, 0-59
[N,6]	Seconds, 0-59
[N,7]	Day of week, 0-6, 0 = Sunday
[N,8]	Days since Jan 1 of current year, 0-365

Example

```
dtv = utctodtv(timeutc);
utc = dtvtoutc(dtv);
```

```
dtv = 2012     7    17    10    13    48     2   198
utc = 1342545228
```

See Also

dtvnormal, timeutc, utctodt, dttodtv, dttoutc, dtvtodt, dtvtoutc, strtodt, dttostr

dummy

Purpose

Creates a set of dummy (0/1) variables by breaking up a variable into specified categories. The highest (rightmost) category is unbounded on the right.

Format

y = **`dummy`**(x, v);

Input

x	Nx1 vector of data that is to be broken up into dummy variables.
v	(K-1)x1 vector specifying the K-1 breakpoints (these must be in ascending order) that determine the K categories to be used. These categories should not overlap.

Output

y	NxK matrix containing the K dummy variables.

Remarks

Missings are deleted before the dummy variables are created.

All categories are open on the left (i.e., do not contain their left boundaries) and all but the highest are closed on the right (i.e., do contain their right boundaries). The highest (rightmost) category is unbounded on the right. Thus, only K-1 breakpoints are required to specify K dummy variables.

The function **dummybr** is similar to **dummy**, but in that function the highest category is bounded on the right. The function **dummydn** is also similar to **dummy**, but in that function a specified column of dummies is dropped.

Example

```
    //Set seed for repeatable random numbers
    rndseed 135345;

    //Create uniform random integers between 1 and 9
    x = ceil(9*rndu(5,1));

    //Set the breakpoints
    v = { 1, 5, 7 };

    dm = dummy(x,v);
```

The code above produces four dummies based upon the breakpoints in the vector *v*:

```
        x < 1
    1 < x < 5
    5 < x < 7
    7 < x
```

which look like:

```
         0 1 0 0       2
         0 0 0 1       9
    dm = 0 1 0 0   x = 4
    0 0 1 0       7
    1 0 0 0       1
```

Source

datatran.src

See Also

dummybr, dummydn

dummybr

Purpose

Creates a set of dummy (0/1) variables. The highest (rightmost) category is bounded on the right.

Format

y = **dummybr**(*x*, *v*);

Input

x	Nx1 vector of data that is to be broken up into dummy variables.
v	Kx1 vector specifying the K breakpoints (these must be in ascending order) that determine the K categories to be used. These categories should not overlap.

Output

y	NxK matrix containing the K dummy variables. Each row will have a maximum of one 1.

Remarks

Missings are deleted before the dummy variables are created.

All categories are open on the left (i.e., do not contain their left boundaries) and are closed on the right (i.e., do contain their right boundaries). Thus, K breakpoints are required to specify K dummy variables.

The function **dummy** is similar to **dummybr**, but in that function the highest category is unbounded on the right.

Example

```
//Set seed for repeatable random numbers
rndseed 135345;

//Create uniform random integers between 1 and 9
x = ceil(9*rndu(5,1));

//Set the breakpoints
v = { 1, 5, 7 };

dm = dummybr(x,v);
```

The code above produces three dummies based upon the breakpoints in the vector *v*:

```
          x  <  1
     1  <  x  <  5
     5  <  x  <  7
```

which look like:

```
          0  1  0         2
          0  0  0         9
     dm = 0  1  0    x  = 4
          0  0  1         7
          1  0  0         1
```

Source

`datatran.src`

See Also

dummydn, dummy

dummydn

Purpose

Creates a set of dummy (0/1) variables by breaking up a variable into specified categories. The highest (rightmost) category is unbounded on the right, and a specified column of dummies is dropped.

Format

y = **dummydn**(*x*, *v*, *p*);

Input

x	Nx1 vector of data to be broken up into dummy variables.
v	(K-1)x1 vector specifying the K-1 breakpoints (these must be in ascending order) that determine the K categories to be used. These categories should not overlap.
p	positive integer in the range [1,K], specifying which column should be dropped in the matrix of dummy variables.

Output

y	Nx(K-1) matrix containing the K-1 dummy variables.

Remarks

This is just like the function **dummy**, except that the *p*th column of the matrix of dummies is dropped. This ensures that the columns of the matrix of dummies do not sum to 1, and so these variables will not be collinear with a vector of ones.

Missings are deleted before the dummy variables are created.

All categories are open on the left (i.e., do not contain their left boundaries) and all but the highest are closed on the right (i.e., do contain their right boundaries). The highest (rightmost) category is unbounded on the right. Thus, only K-1 breakpoints are required to specify K dummy variables.

Example

```
//Set seed for repeatable random numbers
rndseed 135345;
```

```
//Create uniform random integers between 1 and 9
x = ceil(9*rndu(5,1));

//Set the breakpoints
v = { 1, 5, 7 };

//Column to drop
p = 2;

dm = dummydn(x,v,p);
```

The code above produces four dummies based upon the breakpoints in the vector *v*:

```
    x < 1
1 < x < 5
5 < x < 7
7 < x
```

and then remove the *pth* column which will result in:

```
     0 0 0       2
     0 0 1       9
dm = 0 0 0   x = 4
     0 1 0       7
     1 0 0       1
```

Source

datatran.src

See Also

dummy, dummybr

e

ed

Purpose

Accesses an alternate editor.

Format

```
ed filename;
```

Input

filename	literal, the name of the file to be edited.

Remarks

The default name of the editor is set in `gauss.cfg`. To change the name of the editor used from within a **GAUSS** session, enter:

```
ed = editor_name flags;
```

or

```
ed = "editor_nameflags";
```

The flags are any command line flags you may want between the name of the editor and the filename when your editor is invoked. The quoted version will prevent the flags, if any, from being forced to uppercase.

This command can be placed in the startup file, so it will be set for you automatically when you start **GAUSS**.

See the `edit` command to open a file in the **GAUSS** editor from the command line.

edit

Purpose

Edits a disk file.

Format

```
edit filename;
```

Input

filename	literal, the name of the file to be edited.

This command loads a disk file in a **GAUSS** edit window. It is available only in the **GAUSS** graphical user interface.

Remarks

The edit command does not follow the *src_path* to locate files. You must specify the location in the *filename*. The default location is the current directory.

To edit the last run file, use `F7` or the Action List toolbar.

Example

```
edit test1.e;
```

See Also

run

erfInv,erfCInv

Purpose

Computes the inverse of the Gaussian error function (**`erfInv`**) and its complement (**`erfcInv`**).

Format

x = **`erfInv`**(*y*);
x = **`erfCInv`**(*y*);

Input

y	scalar or NxK matrix. -1 < y < 1.

Output

x	scalar or NxK matrix.

Example

```
x = seqa(.1,.1,10);
y = erf(x);
```

```
          0.1000       0.1125
          0.2000       0.2227
          0.3000       0.3286
          0.4000       0.4284
      x = 0.5000   y = 0.5205
          0.6000       0.6039
          0.7000       0.6778
          0.8000       0.7421
          0.9000       0.7969
          1.0000       0.8427
```

```
print erfInv(y);
```

```
          0.1000
          0.2000
          0.3000
          0.4000
          0.5000
          0.6000
          0.7000
          0.8000
          0.9000
          1.0000
```

See Also

erf, erfc, cdfN, cdfNC, cdfNi

eig

Purpose

Computes the eigenvalues of a general matrix.

Format

```
va = eig(x);
```

Input

x	NxN matrix or K-dimensional array where the last two dimensions are NxN.

Output

va	Nx1 vector or K-dimensional array where the last two dimensions are Nx1, the eigenvalues of *x*.

Remarks

If *x* is an array, the result will be an array containing the eigenvalues of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10x4x4 array, the result will be a 10x4x1 array containing the eigenvalues of each of the 10 4x4 arrays contained in *x*.

Errors

If the eigenvalues cannot all be determined, *va*[1] is set to an error code. Passing *va*[1] to the **scalerr** function will return the index of the eigenvalue that failed. The eigenvalues for indices **scalerr**(*va*[1])+1 to N should be correct.

Error handling is controlled with the low bit of the trap flag.

trap 0	set *va*[1] and terminate with message
trap 1	set *va*[1] and continue execution

Invalid inputs, such as an infinity, missing value or Nan will cause an error. If the **trap** is set to 1, *va* will be set to a scalar error code and program execution will continue. Passing this scalar error code to the **scalerr** function will return -1.

Eigenvalue ordering

The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.

Example

```
x = { 4 8 1,
      9 4 2,
      5 5 7 };

va = eig(x);

     14.4757
va = -4.4979
      5.0222
```

To calculate eigenvalues and eigenvectors see **eigv**.

See Also

eigh, eighv, eigv

eigh

Purpose

Computes the eigenvalues of a complex hermitian or real symmetric matrix.

Format

```
va = eigh(x);
```

Input

x	NxN matrix or K-dimensional array where the last two dimensions are NxN.

Output

va	Nx1 vector or K-dimensional array where the last two dimensions are Nx1, the eigenvalues of *x*.

Remarks

If *x* is an array, the result will be an array containing the eigenvalues of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10x4x4 array, the result will be a 10x4x1 array containing the eigenvalues of each of the 10 4x4 arrays contained in *x*.

Errors

If the eigenvalues cannot all be determined, *va*[1] is set to an error code. Passing *va*[1] to the **scalerr** function will return the index of the eigenvalue that failed. The eigenvalues for indices 1 to **scalerr**(*va*[1])-1 should be correct.

Error handling is controlled with the low bit of the trap flag.

trap 0	set *va*[1] and terminate with message
trap 1	set *va*[1] and continue execution

Invalid inputs, such as an infinity, missing value or Nan will cause an error. If the **trap** is set to 1, *va* will be set to a scalar error code and program execution will continue. Passing this scalar error code to the **scalerr** function will return -1.

Eigenvalue ordering

The eigenvalues are in ascending order.

The eigenvalues of a complex hermitian or real symmetric matrix are always real.

See Also

eig, eighv, eigv

eighv

Purpose

Computes eigenvalues and eigenvectors of a complex hermitian or real symmetric matrix.

Format

{ *va*, *ve* } = **eighv**(*x*);

Input

x	NxN matrix or K-dimensional array where the last two dimensions are NxN.

Output

va	Nx1 vector or K-dimensional array where the last two dimensions are Nx1, the eigenvalues of *x*.
ve	NxN matrix or K-dimensional array where the last two dimensions are NxN, the eigenvectors of *x*.

Remarks

If *x* is an array, *va* will be an array containing the eigenvalues of each 2-dimensional array described by the two trailing dimensions of *x*, and *ve* will be an array containing the corresponding eigenvectors. In other words, for a 10x4x4 array, *va* will be a 10x4x1 array containing the eigenvalues and *ve* a 10x4x4 array containing the eigenvectors of each of the 10 4x4 arrays contained in *x*.

Errors

If the eigenvalues cannot all be determined, *va*[1] is set to an error code. Passing *va*[1] to the **`scalerr`** function will return the index of the eigenvalue that failed. The eigenvalues for indices 1 to **`scalerr`**(*va*[1])-1 should be correct. The eigenvectors are not computed.

Error handling is controlled with the low bit of the trap flag.

trap 0	set *va*[1] and terminate with message
trap 1	set *va*[1] and continue execution

Invalid inputs, such as an infinity, missing value or Nan will cause an error. If the **trap** is set to 1, *va* will be set to a scalar error code and program execution will continue. Passing this scalar error code to the **`scalerr`** function will return -1.

Eigenvalue ordering

The eigenvalues are in ascending order. The columns of *ve* contain the eigenvectors of *x* in the same order as the eigenvalues. The eigenvectors are orthonormal.

The eigenvalues of a complex hermitian or real symmetric matrix are always real.

See Also

eig, eigh, eigv

eigv

Purpose

Computes eigenvalues and eigenvectors of a general matrix.

Format

```
{ va, ve } = eigv(x);
```

Input

x	NxN matrix or K-dimensional array where the last two dimensions are NxN.

Output

va	Nx1 vector or K-dimensional array where the last two dimensions are Nx1, the eigenvalues of *x*.
ve	NxN matrix or K-dimensional array where the last two dimensions are NxN, the eigenvectors of *x*.

Remarks

If *x* is an array, *va* will be an array containing the eigenvalues of each 2-dimensional array described by the two trailing dimensions of *x*, and *ve* will be an array containing the corresponding eigenvectors. In other words, for a 10x4x4 array, *va* will be a 10x4x1 array containing the eigenvalues and *ve* a 10x4x4 array containing the eigenvectors of each of the 10 4x4 arrays contained in *x*.

Errors

If the eigenvalues cannot all be determined, *va*[1] is set to an error code. Passing *va* [1] to the **scalerr** function will return the index of the eigenvalue that failed. The eigenvalues for indices **scalerr**(*va*[1])+1 to N should be correct. The eigenvectors are not computed.

Error handling is controlled with the low bit of the trap flag.

trap 0	set *va*[1] and terminate with message
trap 1	set *va*[1] and continue execution

Invalid inputs, such as an infinity, missing value or Nan will cause an error. If the **trap** is set to 1, *va* will be set to a scalar error code and program execution will continue. Passing this scalar error code to the **scalerr** function will return -1.

Eigenvalue ordering

The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first. The columns of *ve* contain the eigenvectors of *x* in the same order as the eigenvalues. The eigenvectors are not normalized.

Example

```
    x = { 4 8 1,
          9 4 2,
          5 5 7 };

    { y, n } = eigv(x);
```

```
        -4.4979          -0.6693    -0.6408    -0.4015
    y = 14.4757      n = 0.7134    -0.7249    -0.2605
         5.0222          -0.0192    -0.9134     1.6734
```

See Also

eig, eigh, eighv

elapsedTradingDays

Purpose

Computes number of trading days between two dates inclusively.

Format

n = **elapsedTradingDays**(*a*, *b*);

Input

a	scalar, date in DT scalar format.
b	scalar, date in DT scalar format.

Output

n	number of trading days between dates inclusively, that is,

elapsed time includes the dates *a* and *b*.

Remarks

A trading day is a weekday that is not a holiday as defined by the New York Stock Exchange from 1888 through 2013. Holidays are defined in `holidays.asc`. You may edit that file to modify or add holidays.

Example

```
//September 10, 2012
tStart = 20120910110231;

//September 14, 2012
tEnd = 20120914080722;

nDays = elapsedTradingDays(tStart, tEnd);
```

```
nDays = 4
```

Source

`finutils.src`

Globals

_fin_holidays

See Also

getNextTradingDay, getPreviousTradingDay, getNextWeekDay, getPreviousWeekDay

end

Purpose

Terminates a program.

Format

```
end;
```

Remarks

end causes **GAUSS** to revert to interactive mode, and closes all open files. end also closes the auxiliary output file and turns the window on. It is not necessary to put an end statement at the end of a program.

An end command can be placed above a label which begins a subroutine to make sure that a program does not enter a subroutine without a gosub.

stop also terminates a program but closes no files and leaves the window setting as it is.

Example

```
    output on;
    screen off;
    print x;
    end;
```

In this example, a matrix *x* is printed to the auxiliary output. The output to the window is turned off to speed up the printing. The end statement is used to terminate the program, so the output file will be closed and the window turned back on.

See Also

new, stop, system

endp

Purpose

Closes a procedure or keyword definition.

Format

```
endp;
```

Remarks

endp marks the end of a procedure definition that began with a proc or keyword statement. (For details on writing and using procedures, see **Procedures and Keywords**, Chapter 1.)

Example

```
proc regress(y,x);
   retp(inv(x'x)*x'y);
endp;

x = { 1 3 2, 7 4 9, 1 1 6, 3 3 2 };
y = { 3, 5, 2, 7 };

b = regress(y,x);
```

After executing the above code:

```
       0.1546
 b = 1.5028
    -0.1284
```

See Also

proc, keyword, retp

endwind

Purpose

Ends graphic panel manipulation; displays graphs with **rerun**. Note: This function is for use with the deprecated PQG graphics.

Library

pgraph

Format

endwind;

Remarks

This function uses **rerun** to display the most recently created .tkf file.

Source

pwindow.src

See Also

begwind, window, makewind, setwind, nextwind, getwind

envget

Purpose

Searches the environment table for a defined name.

Format

y = **envget**(*s*);

Input

s	string, the name to be searched for.

Output

y	string, the string that corresponds to that name in the environment table or a null string if it is not found.

Example

Example 1

```
//%USERPROFILE% is the user's home directory on most
//Windows systems
hmFld = envget("USERPROFILE");
```

Example 2

```
    proc dopen(file);
        local fname,fp;
        fname = envget("DPATH");
        //Check to see if DPATH is set or empty
        if fname $== "";
            fname = file;
        else;
            //Check to see if 'fname' ends with a
            //path separator
            if strsect(fname,strlen(fname),1) $== "\\";
                fname = fname $+ file;
            else;
                fname = fname $+ "\\" $+ file;
            endif;
        endif;
        open fp = ^fname;
        retp(fp);
    endp;
```

This is an example of a procedure that will open a data file using a path stored in an environment string called DPATH. The procedure returns the file handle and is called as follows:

```
    fp = dopen("myfile");
```

See Also

cdir

eof

Purpose

Tests if the end of a file has been reached.

Format

y = **eof**(*fh*);

Input

fh	scalar, file handle.

Output

y	scalar, 1 if end of file has been reached, else 0.

Remarks

This function is used with **readr** and the **fgets** *xxx* commands to test for the end of a file.

The **seekr** function can be used to set the pointer to a specific row position in a data set; the **fseek** function can be used to set the pointer to a specific byte offset in a file opened with **fopen**.

Example

```
    open f1 = dat1;
```

```
xx = 0;
do until eof(f1);
    xx = xx + moment(readr(f1,100),0);
endo;
```

In this example, the data file `dat1.dat` is opened and given the handle *f1*. Then the data are read from this data set and are used to create the moment matrix ($x'x$) of the data. On each iteration of the loop, 100 additional rows of data are read in, and the moment matrix for this set of rows is computed and added to the matrix *xx*. When all the data have been read, *xx* will contain the entire moment matrix for the data set.

GAUSS will keep reading until **eof**(*f1*) returns the value 1, which it will when the end of the data set has been reached. On the last iteration of the loop, all remaining observations are read in if there are 100 or fewer left.

See Also

open, readr, seekr

eqSolve

Purpose

Solves a system of nonlinear equations.

Format

{ *x*, *retcode* } = **eqSolve**(&*F*, *start*);

Input

start	Kx1 vector, starting values.

&F	scalar, a pointer to a procedure which computes the value at *x* of the equations to be solved.

Global Input

The following are set by **eqSolveSet**:

_eqs_JacobianProc	pointer to a procedure which computes the analytical Jacobian. By default, **eqSolve** will compute the Jacobian numerically.
_eqs_MaxIters	scalar, the maximum number of iterations. Default = 100.
_eqs_StepTol	scalar, the step tolerance. Default = *__macheps*$^{2/3}$.
_eqs_TypicalF	Kx1 vector of the typical **F**(*x*) values at a point not near a root, used for scaling. This becomes important when the magnitudes of the components of **F**(*x*) are expected to be very different. By default, function values are not scaled.
_eqs_TypicalX	Kx1 vector of the typical magnitude of *x*, used for scaling. This becomes important when the magnitudes of the components of *x* are expected to be very different. By default, variable values are not scaled.
_eqs_IterInfo	scalar, if nonzero, iteration information is printed. Default = 0.

The following are set by **gausset**:

__Tol	scalar, the tolerance of the scalar function $f = 0.5*\|\|F(x)\|\|^2$ required to terminate the algorithm. Default = 1e-5.

`__altnam`	Kx1 character vector of alternate names to be used by the printed output. By default, the names "X1, X2,X3..." or "X01,X02,X03..." (depending on how `__vpad` is set) will be used.
`__output`	scalar. If non-zero, final results are printed.
`__title`	string, a custom title to be printed at the top of the iterations report. By default, only a generic title will be printed.
`__vpad`	scalar. If `__altnam` is not set, variable names are automatically created. Two types of names can be created:
	0 — Variable names are not padded to give them equal length. For example, `X1, X2,...,X10,...`
	1 — Variable names are padded with zeros to give them an equal number of characters. For example, `X01,X02,...,X10,...` This is useful if you want the variable names to sort properly.

Output

x	Kx1 vector, solution.

Remarks

The equation procedure should return a column vector containing the result for each equation. For example:

```
    Equation 1:    x1² + x2² - 2 = 0
    Equation 2:    exp(x1-1) + x2³ - 2 = 0
```

```
proc f(var);
   local x1,x2,eqns;
   x1 = var[1];
   x2 = var[2];
   eqns[1] = x1^2 + x2^2 - 2;        /* Equation 1 */
   eqns[2] = exp(x1-1) + x2^3 - 2;  /* Equation 2 */
retp(eqns);
endp;
```

Example

```
eqSolveSet;

proc f(x);
   local f1,f2,f3;
   f1 = 3*x[1]^3 + 2*x[2]^2 + 5*x[3] - 10;
   f2 = -x[1]^3 - 3*x[2]^2 + x[3] + 5;
   f3 = 3*x[1]^3 + 2*x[2]^2 - 4*x[3];
retp(f1|f2|f3);
endp;

proc fjc(x);
   local fjc1,fjc2, fjc3;
   fjc1 = 9*x[1]^2 ~ 4*x[2] ~ 5;
   fjc2 = -3*x[1]^2 ~ -6*x[2] ~ 1;
   fjc3 = 9*x[1]^2 ~ 4*x[2] ~ -4;
retp(fjc1|fjc2|fjc3);
endp;

start = { -1, 12, -1 };

_eqs_JacobianProc = &fjc;

{ x,tcode } = eqSolve(&f,start);
```

produces:

```
==========================================================
 EqSolve Version 11.0.5              7/17/2012   5:47 pm
==========================================================

||F(X)|| at final solution:                   0.93699762
----------------------------------------------------------
Termination Code = 1:

Norm of the scaled function value is less than __Tol;
----------------------------------------------------------


----------------------------------------------------------
VARIABLE      START            ROOTS                F(ROOTS)
----------------------------------------------------------
X1           -1.00000        0.54144351      4.4175402e-006
X2           12.00000         1.4085912     -6.6263102e-006
X3           -1.00000         1.1111111      4.4175402e-006
----------------------------------------------------------
```

Source

`eqsolve.src`

eqSolvemt

Purpose

Solves a system of nonlinear equations.

Include

`eqsolvemt.sdf`

Format

```
out = eqSolvemt(&fct, par, data, c);
```

Input

&fct	pointer to a procedure that computes the function to be minimized. This procedure must have two input arguments, an instance of a **PV** structure containing the parameters, and an instance of a **DS** structure containing data, if any. And, one output argument, a column vector containing the result of each equation.
par	an instance of a **PV** structure. The *par* instance is passed to the user-provided procedure pointed to by &**fct**. *par* is constructed using the **pvPack** functions.
data	an array of instances of a **DS** structure. This array is passed to the user-provided procedure pointed to by &**fct** to be used in the objective function. **eqSolvemt** does not look at this structure. Each instance contains the the following members which can be set in whatever way that is convenient for computing the objective function:

data1 [i].dataMatrix	NxK matrix, data matrix.
data1 [i].dataArray	NxKxL... array, data array.
data1[i].vnames	string array, variable names (optional).
data1[i].dsname	string, data name (optional).
data1[i].type	scalar, type of data (optional).

c	an instance of an **eqSolvemtControl** structure. Normally an instance is initialized by calling **`eqSolvemtControlCreate`** and members of this instance can be set to other values by the user. For an instance named *c*, the members are:

c.jacobianProc	pointer to a procedure which computes the analytical Jacobian. By default, **`eqSolvemt`** will compute the Jacobian numerically.
c.maxIters	scalar, the maximum number of iterations. Default = 100.
c.stepTolerance	scalar, the step tolerance. Default = macheps$^{2/3}$.
c.typicalF	Kx1 vector of the typical **`fct`**(*x*) values at a point not near a root, used for scaling. This becomes important when the magnitudes of the components of **`fct`**(*x*) are expected to be very different. By default, function values are not scaled.
c.typicalX	Kx1 vector of the typical magnitude of *x*, used for scaling. This becomes important when the magnitudes of the components of *x* are expected to be very different. By default, variable values are not scaled.
c.printIters	scalar, if nonzero, iteration information is printed. Default = 0.
c.tolerance	scalar, the tolerance of the scalar function *f*

	`= 0.5*||fct(X)||`^{2} required to terminate the algorithm. That is, the condition that `|f(x)| <= c.tolerance` must be met before that algorithm can terminate successfully. Default = 1e-5.
c.altnam	Kx1 character vector of alternate names to be used by the printed output. By default, the names "X1,X2,X3..." will be used.
c.title	string, printed as a title in output.
c.output	scalar. If non-zero, final results are printed.

Output

out an instance of an **eqSolvemtOut** structure. For an instance named *out*, the members are:

out.par	an instance of a **PV** structure containing the parameter estimates.
out.fct	scalar, function evaluated at *x*
out.retcode	scalar, return code:

-1	Jacobian is singular.
1	Norm of the scaled function value is less than *c.tolerance*. *x* given is an approximate root of *fct*(*x*) (unless *c.tolerance* is too large).

2	The scaled distance between the last two steps is less than the step-tolerance (*c.stepTolerance*). *x* may be an approximate root of **fct**(*x*), but it is also possible that the algorithm is making very slow progress and is not near a root, or the step-tolerance is too large.
3	The last global step failed to decrease **norm2**(*fct*(*x*)) sufficiently; either *x* is close to a root of **fct**(*x*) and no more accuracy is possible, or an incorrectly coded analytic Jacobian is being used, or the secant approximation to the Jacobian is inaccurate, or the step-tolerance is too large.
4	Iteration limit exceeded.
5	Five consecutive steps of maximum step length have been taken; either **norm2** (*fct*(*x*)) asymptotes from above to a finite value in some direction or the maximum step length is too small.
6	*x* seems to be an approximate local minimizer of **norm2**(*fct*(*x*)) that is not a root of **fct**(*x*). To find a root of **fct**(*x*), restart **eqSolvemt** from a different region.

Remarks

The equation procedure should return a column vector containing the result for each equation.

If there is no data, you can pass an empty **DS** structure in the second argument:

```
call eqSolvemt(&fct,par,dsCreate(),c);
```

Example

```
Equation 1:   x1^2 + x2^2 - 2 = 0
Equation 2:   exp(x1-1) + x2^3 - 2 = 0
```

```
#include eqSolvemt.sdf
struct eqSolvemtControl c;
c = eqSolvemtControlCreate;

c.printIters = 1;

struct PV par;
par = pvPack(pvCreate,1, "x1");
par = pvPack(par,1, "x2");

struct eqSolvemtOut out1;
out1 = eqSolvemt(&fct,par,dsCreate(),c);

proc fct(struct PV p, struct DS d);
   local x1, x2, z;
   x1 = pvUnpack (p, "x1");
   x2 = pvUnpack (p, "x2");
   z = x1^2+x2^2-2 | exp(x1-1)+x2^3-2;
   retp(z);
endp;
```

Source

eqsolvemt.src

See Also

eqSolvemtControlCreate, eqSolvemtOutCreate

eqSolvemtControlCreate

Purpose

Creates default **eqSolvemtControl** structure.

Include

eqsolvemt.sdf

Format

c = **eqSolvemtControlCreate**();

Output

c	instance of **eqSolvemtControl** structure with members set to default values.

Example

Since structures are strongly typed in **GAUSS**, each structure must be declared before it can be used.

```
//declare c as an
//eqSolvemtControl structure
struct eqSolvemtControl c;

//initialize structure c
c = eqSolvemtControlCreate();
```

The members of an **eqSolvemtControl** structure and default values are described in the manual entry for **eqSolvemt**.

If you intend to just use the default values for the **eqSolvemtControl** structure, you can simply pass in the function **eqSolvemtControlCreate** as the final argument to **eqSolvemt**:

```
//set up data, function pointer, etc.

out = eqSolvemt(&fct, par, data, eqSolvemtControlCreate
() );
```

Source

eqsolvemt.src

See Also

eqSolvemt

eqSolvemtOutCreate

Purpose

Creates default **eqSolvemtOut** structure.

Include

`eqsolvemt.sdf`

Format

c = **`eqSolvemtOutCreate`**();

Output

c	instance of **eqSolvemtOut** structure with members set to default values.

Example

Since structures are strongly typed in **GAUSS**, each structure must be declared before it can be used.

```
//declare structure
struct eqSolvemtOut c;

//Initialize structure
c = eqSolvemtOutCreate();
```

The members of an **eqSolvemtOut** structure and default values are described in the manual entry for **`eqSolvemt`**.

Source

`eqsolvemt.src`

See Also

eqSolvemt

eqSolveSet

Purpose

Sets global input used by **`eqSolve`** to default values.

Format

```
eqSolveset;
```

Global Output

__eqs_TypicalX	Set to 0.
__eqs_TypicalF	Set to 0.
__eqs_IterInfo	Set to 0.
__eqs_JacobianProc	Set to 0.
__eqs_MaxIters	Set to 100.
__eqs_StepTol	Set to *__macheps*$^{2/3}$

erf,erfc

Purpose

Computes the Gaussian error function (**`erf`**) and its complement (**`erfc`**).

Format

y = **erf**(*x*);
y = **erfc**(*x*);

Input

x	NxK matrix.

Output

y	NxK matrix.

Remarks

The **erf** and **erfc** functions are closely related to the Normal distribution:

```
if x > 0
   cdfn(x) = 0.5 * (1 + erf(x / sqrt(2));

if x ≤ 0
   cdfn(x) = 0.5 * erfc(-x / sqrt(2));
```

Example

```
//Print 3 digits after the decimal point
format /rd 5,3;

x = { .5 .4 .3,
      .6 .8 .3 };
y = erf(x);
```

```
    yc = erfc(x);

    //The '~' operator performs horizontal concatenation
    //and causes this print statement to format 'x',
    //'y' and 'yc' as if they were one 2x9 matrix rather
    //than 3 2x3 matrices
    //This does not change the variable values, only
    //their appearance for this print statement
    print x~y~yc;
```

produces the following output:

```
    0.500 0.400 0.300 0.520 0.428 0.329 0.480 0.572 0.671
    0.600 0.800 0.300 0.604 0.742 0.329 0.396 0.258 0.671
```

See Also

cdfN, cdfNc

Technical Notes

erf and **erfc** are computed by summing the appropriate series and continued fractions. They are accurate to about 12 or more digits.

erfcplx,erfccplx

Purpose

Computes the Gaussian error function (**erfcplx**) and its complement (**erfccplx**) for complex inputs.

Format

```
f = erfcplx(z);
f = erfccplx(z);
```

Input

z	NxK complex matrix; *z* must be > 0.

Output

f	NxK complex matrix.

Technical Notes

Accuracy is better than 12 significant digits.

References

1. Abramowitz & Stegun, section 7.1, equations 7.1.9, 7.1.23, and 7.1.29
2. Main author Paul Godfrey
3. Small changes by Peter J. Acklam

error

Purpose

Allows the user to generate a user-defined error code which can be tested quickly with the **`scalerr`** function.

Format

```
y = error(x);
```

Input

x	scalar, in the range 0-65535.

Output

y	scalar error code which can be interpreted as an integer with the **scalerr** function.

Remarks

The user may assign any number in the range 0-65535 to denote particular error conditions. This number may be tested for as an error code by **scalerr**.

The **scalerr** function will return the value of the error code and so is the reverse of **error**. These user-generated error codes work in the same way as the intrinsic **GAUSS** error codes which are generated automatically when **trap 1** is on and certain **GAUSS** functions detect a numerical error such as a singular matrix.

```
    error(0);
```

is equal to the missing value code.

Example

```
    proc syminv(x);
       local oldtrap,y;
```

```
        if not x == x';
           retp(error(99));
        endif;
        oldtrap = trapchk(0xffff);
        trap 1;
        y = invpd(x);
        if scalerr(y);
           y = inv(x);
        endif;
        trap oldtrap,0xffff;
        retp(y);
     endp;
```

The procedure **syminv** returns error code 99 if the matrix is not symmetric. If **invpd** fails, it returns error code 20. If **inv** fails, it returns error code 50. The original trap state is restored before the procedure returns.

See Also

scalerr, trap, trapchk

errorlog

Purpose

Prints an error message to the window and error log file.

Format

```
errorlog str;
```

Input

str	string, the error message to print.

Remarks

This command enables you to do your own error handling in your **GAUSS** programs. To print an error message to the window and error log file along with file name and line number information, use `errorlogat`.

See Also

errorlogat

errorlogat

Purpose

Prints an error message to the window and error log file, along with the file name and line number at which the error occurred.

Format

```
errorlogat str;
```

Input

str	string, the error message to print.

Remarks

This command enables you to do your own error handling in your **GAUSS** programs. To print an error message to the window and error log file without file name and line number information, use `errorlog`.

See Also

errorlog

etdays

Purpose

Computes the difference between two times, as generated by the **`date`** command, in days.

Format

```
days = etdays(tstart, tend);
```

Input

tstart	3x1 or 4x1 vector, starting date, in the order: yr, mo, day. (Only the first 3 elements are used.)
tend	3x1 or 4x1 vector, ending date, in the order: yr, mo, day. (Only the first 3 elements are used.) MUST be later than *tstart*.

Output

days	scalar, elapsed time measured in days.

Remarks

This will work correctly across leap years and centuries. The assumptions are a Gregorian calendar with leap years on the years evenly divisible by 4 and not evenly divisible by 100, unless divisible by 400.

Example

```
    let date1 = 2008 1 2;
    let date2 = 2009 9 14;
    d = etdays(date1,date2);
```

After the code above, *d* is equal to:

```
    621
```

Source

time.src

See Also

dayinyr

ethsec

Purpose

Computes the difference between two times, as generated by the **date** command, in hundredths of a second.

Format

```
hs = ethsec(tstart, tend);
```

Input

tstart	4x1 vector, starting date, in the order: yr, mo, day, hundredths of a second.
tend	4x1 vector, ending date, in the order: yr, mo, day, hundredths of a second. MUST be later date than *tstart*.

Output

hs	scalar, elapsed time measured in hundredths of a second.

Remarks

This will work correctly across leap years and centuries. The assumptions are a Gregorian calendar with leap years on the years evenly divisible by 4 and not evenly divisible by 100, unless divisible by 400.

Example

```
let date1 = 2008 1 2 0;
let date2 = 2009 9 14 0;
t = ethsec(date1,date2);
```

After the code above, `t` is equal to:

```
5365440000
```

Source

`time.src`

See Also

dayinyr

etstr

Purpose

Formats an elapsed time measured in hundredths of a second to a string.

Format

str = **etstr**(*tothsecs*);

Input

tothsecs	scalar, an elapsed time measured in hundredths of a second, as given, for instance, by the **ethsec** function.

Output

str	string containing the elapsed time in the form: # days # hours # minutes #,## seconds

Example

```
    d1 = { 2012, 1, 2, 0 };
    d2 = { 2012, 1, 14, 815642 };
    t = ethsec(d1,d2);
    str = etstr(t);

    print "t   = " t;
    print "str = " str;
```

Output:

```
    t   = 104495642.000
    str = 12 days  2 hours  15 minutes  56.42 seconds
```

Source

time.src

See Also

ethsec

EuropeanBinomCall

Purpose

Prices European call options using binomial method.

Format

```
c = EuropeanBinomCall(S0, K, r, div, tau, sigma, N);
```

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

Output

c	Mx1 vector, call premiums.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach", *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2012);
c = EuropeanBinomCall(S0,K,r,0,tau,sigma,60);
print c;
```

produces:

```
17.1325
14.8599
12.6383
```

Source

finprocs.src

EuropeanBinomCall_Greeks

Purpose

Computes Delta, Gamma, Theta, Vega, and Rho for European call options using binomial method.

Format

```
{ d, g, t, v, rh } = EuropeanBinomCall_Greeks(S0, K, r,
div, tau, sigma, N);
```

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

Global Input

_fin_thetaType	scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.
_fin_epsilon	scalar, finite difference stepsize. Default = 1e-8.

Output

d	Mx1 vector, delta.
g	Mx1 vector, gamma.
t	Mx1 vector, theta.
v	Mx1 vector, vega.
rh	Mx1 vector, rho.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach", *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
    S0 = 305;
    K = 300;
    r = .08;
    sigma = .25;
    tau = .33;
    div = 0;
    print EuropeanBinomcall_Greeks(S0,K,r,0,tau,sigma,30);
```

produces:

```
      0.670
      0.000
    -38.426
```

```
65.170
56.677
```

Source

`finprocs.src`

See Also

EuropeanBinomCall_ImpVol, EuropeanBinomCall, EuropeanBinomPut_Greeks, EuropeanBSCall_Greeks

EuropeanBinomCall_ImpVol

Purpose

Computes implied volatilities for European call options using binomial method.

Format

sigma = **EuropeanBinomCall_ImpVol**(*c*, *S0*, *K*, *r*, *div*, *tau*, *N*);

Input

c	Mx1 vector, call premiums.
S0	scalar, current price.
K	Mx1 vector, strike prices.

r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
N	number of time segments.

Output

sigma	Mx1 vector, volatility.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach", *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
c = { 13.70, 11.90, 9.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0368;
div = 0;
t0 = dtday(2012, 1, 30);
t1 = dtday(2012, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2012);
sigma = EuropeanBinomCall_ImpVol(c,S0,K,r,0,tau,30);
print sigma;
```

produces:

```
    0.2027
    0.2081
    0.1989
```

Source

`finprocs.src`

EuropeanBinomPut

Purpose

Prices European put options using binomial method.

Format

c = **`EuropeanBinomPut`**(*`S0`*, *`K`*, *`r`*, *`div`*, *`tau`*, *`sigma`*, *`N`*);

Input

`S0`	scalar, current price.
`K`	Mx1 vector, strike prices.
`r`	scalar, risk free rate.
`div`	continuous dividend yield.
`tau`	scalar, elapsed time to exercise in annualized days of trading.
`sigma`	scalar, volatility.
`N`	number of time segments.

Output

c	Mx1 vector, put premiums.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach", *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0398;
sigma = .2493;
t0 = dtday(2012, 1, 30);
t1 = dtday(2012, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2012);
c = EuropeanBinomPut(S0,K,r,0,tau,sigma,60);print c;
```

produces:

```
16.872213
19.606098
22.390831
```

Source

finprocs.src

EuropeanBinomPut_Greeks

Purpose

Computes Delta, Gamma, Theta, Vega, and Rho for European put options using binomial method.

Format

{ *d*, *g*, *t*, *v*, *rh* } = **EuropeanBinomPut_Greeks**(*S0*, *K*, *r*, *div*, *tau*, *sigma*, *N*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

Global Input

_fin_thetaType	scalar, if 1, one day look ahead, else, infinitesimal. Default = 0.

_fin_epsilon	scalar, finite difference stepsize. Default = 1e-8.

Output

d	Mx1 vector, delta.
g	Mx1 vector, gamma.
t	Mx1 vector, theta.
v	Mx1 vector, vega.
rh	Mx1 vector, rho.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach", *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
S0 = 305;
K = 300;
r = .08;
div = 0;
sigma = .25;
tau = .33;
print EuropeanBinomPut_Greeks(S0,K,r,0,tau,sigma,60);
```

produces:

```
      -0.350
       0.001
       7.237
      65.432
     -39.652
```

Source

`finprocs.src`

See Also

EuropeanBinomPut_ImpVol, EuropeanBinomPut, EuropeanBinomCall_Greeks, EuropeanBSPut_Greeks

EuropeanBinomPut_ImpVol

Purpose

Computes implied volatilities for European put options using binomial method.

Format

sigma = **EuropeanBinomPut_ImpVol**(*c*, *S0*, *K*, *r*, *div*, *tau*, *N*);

Input

c	Mx1 vector, put premiums.
S0	scalar, current price.

K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
N	number of time segments.

Output

sigma	Mx1 vector, volatility.

Remarks

The binomial method of Cox, Ross, and Rubinstein ("Option pricing: a simplified approach", *Journal of Financial Economics*, 7:229:264) as described in *Options, Futures, and other Derivatives* by John C. Hull is the basis of this procedure.

Example

```
p = { 14.60, 17.10, 20.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0398;
div = 0;
t0 = dtday(2012, 1, 30);
t1 = dtday(2012, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2012);
sigma = EuropeanBinomPut_ImpVol(p,S0,K,r,0,tau,30);
print sigma;
```

produces:

```
    0.21609253
    0.21139494
    0.21407512
```

Source

`finprocs.src`

EuropeanBSCall

Purpose

Prices European call options using Black, Scholes and Merton method.

Format

c = **`EuropeanBSCall`**(*S0*, *K*, *r*, *div*, *tau*, *sigma*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

Output

c	Mx1 vector, call premiums.

Example

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2012, 1, 30);
t1 = dtday(2012, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2012);
c = EuropeanBSCall(S0,K,r,0,tau,sigma);
print c;
```

produces:

```
17.1351
14.7955
12.6860
```

Source

finprocs.src

EuropeanBSCall_Greeks

Purpose

Computes Delta, Gamma, Theta, Vega, and Rho for European call options using Black, Scholes, and Merton method.

Format

{ *d*, *g*, *t*, *v*, *rh* } = **EuropeanBSCall_Greeks**(*S0*, *K*, *r*, *div*, *tau*, *sigma*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

Global Input

_fin_thetaType	scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.
_fin_epsilon	scalar, finite difference stepsize. Default = 1e-8.

Output

d	Mx1 vector, delta.
g	Mx1 vector, gamma.
t	Mx1 vector, theta.
v	Mx1 vector, vega.
rh	Mx1 vector, rho.

Example

```
S0 = 305;
K = 300;
r = .08;
sigma = .25;
tau = .33;
print EuropeanBSCall_Greeks(S0,K,r,0,tau,sigma);
```

produce:

```
  0.6446
  0.0085
-38.5054
 65.2563
 56.8720
```

Source

finprocs.src

See Also

EuropeanBSCall_ImpVol, EuropeanBSCall, EuropeanBSPut_Greeks, EuropeanBinomCall_Greeks

EuropeanBSCall_ImpVol

Purpose

Computes implied volatilities for European call options using Black, Scholes, and Merton method.

Format

```
sigma = EuropeanBSCall_ImpVol(c, S0, K, r, div, tau);
```

Input

c	Mx1 vector, call premiums.
S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.

Output

sigma	Mx1 vector, volatility.

Example

```
c = { 13.70, 11.90, 9.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
t0 = dtday(2012, 1, 30);
t1 = dtday(2012, 2, 16);
tau = elapsedTradingDays(t0,t1) /
    annualTradingDays(2012);
sigma = EuropeanBSCall_ImpVol(c,S0,K,r,0,tau);
print sigma;
```

produces:

```
0.1986
0.2064
0.1951
```

Source

`finprocs.src`

EuropeanBSPut

Purpose

Prices European put options using Black, Scholes, and Merton method.

Format

c = **EuropeanBSPut**(*S0*, *K*, *r*, *div*, *tau*, *sigma*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

Output

c	Mx1 vector, put premiums.

Example

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2012, 1, 30);
t1 = dtday(2012, 2, 16);
tau = elapsedTradingDays(t0,t1) /
```

```
        annualTradingDays(2012);
    c = EuropeanBSPut(S0,K,r,0,tau,sigma);
    print c;
```

produces:

```
    16.6700
    19.3164
    22.1930
```

Source

finprocs.src

EuropeanBSPut_Greeks

Purpose

Computes Delta, Gamma, Theta, Vega, and Rho for European put options using Black, Scholes, and Merton method.

Format

{ *d*, *g*, *t*, *v*, *rh* } = **EuropeanBSPut_Greeks**(*S0*, *K*, *r*, *div*, *tau*, *sigma*);

Input

S0	scalar, current price.
K	Mx1 vector, strike prices.

r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

Global Input

_fin_thetaType	scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.
_fin_epsilon	scalar, finite difference stepsize. Default = 1e-8.

Output

d	Mx1 vector, delta.
g	Mx1 vector, gamma.
t	Mx1 vector, theta.
v	Mx1 vector, vega.
rh	Mx1 vector, rho.

Example

```
     S0 = 305;
     K = 300;
```

```
    r = .08;
    sigma = .25;
    tau = .33;
    print EuropeanBSPut_Greeks(S0,K,r,0,tau,sigma);
```

produces:

```
      -0.3554
       0.0085
    -15.1307
      65.2563
    -39.54861
```

Source

`finprocs.src`

See Also

EuropeanBSPut_ImpVol, EuropeanBSPut, EuropeanBSCall_Greeks, EuropeanBinomPut_Greeks

EuropeanBSPut_ImpVol

Purpose

Computes implied volatilities for European put options using Black, Scholes, and Merton method.

Format

sigma = **EuropeanBSPut_ImpVol**(*c*, *S0*, *K*, *r*, *div*, *tau*);

Input

c	Mx1 vector, put premiums
S0	scalar, current price.
K	Mx1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.

Output

sigma	Mx1 vector, volatility.

Example

```
    p = { 14.60, 17.10, 20.10 };
    S0 = 718.46;
    K = { 720, 725, 730 };
    r = .0498;
    t0 = dtday(2012, 1, 30);
    t1 = dtday(2012, 2, 16);
    tau = elapsedTradingDays(t0,t1) / annualTradingDays(201
    2);
    sigma = EuropeanBSPut_ImpVol(p,S0,K,r,0,tau);
    print sigma;
```

produce:

```
    0.2188
    0.2165
    0.2177
```

Source

`finprocs.src`

exctsmpl

Purpose

Computes a random subsample of a data set.

Format

n = **`exctsmpl`**(*`infile`*, *`outfile`*, *`percent`*);

Input

`infile`	string, the name of the original data set.
`outfile`	string, the name of the data set to be created.
`percent`	scalar, the percentage random sample to take. This must be in the range 0-100.

Output

n	scalar, number of rows in output data set.

Error returns are controlled by the low bit of the trap flag:

trap 0	terminate with error message	
trap 1	return scalar negative integer	
	-1	can't open input file
	-2	can't open output file
	-3	disk full

Remarks

Random sampling is done with replacement. Thus, an observation may be in the resulting sample more than once. If *percent* is 100, the resulting sample will not be identical to the original sample, though it will be the same size.

Example

```
    n = exctsmpl("freqdata.dat","rout",30);
```

`freqdata.dat` is an example data set provided with **GAUSS**. Switching to the `examples` subdirectory of your **GAUSS** installation directory will make it possible to do the above example as shown. Otherwise you will need to substitute another data set name for "`freqdata.dat`".

Source

`exctsmpl.src`

exec

Purpose

Executes an executable program and returns the exit code to **GAUSS**.

Format

```
y = exec(program, comline);
```

Input

program	string, the name of the program, including the extension, to be executed.
comline	string, the arguments to be placed on the command line of the program being executed.

Output

y scalar, the exit code returned by *program*.

If **exec** can't execute *program*, the error returns will be negative:

-1	file not found
-2	the file is not an executable file
-3	not enough memory
-4	command line too long

Example

```
y = exec("atog","comd1.cmd");

//If 'y' is nonzero
if y;
   errorlog"atog failed";
   end;
endif;
```

In this example the ATOG ASCII conversion utility is executed under the **exec** function. The name of the command file to be used, `comd1.cmd`, is passed to ATOG on its command line. The exit code *y* returned by **exec** is tested to see if ATOG was successful; if not, the program will be terminated after printing an error message. See **ATOG**, Section 1.

execbg

Purpose

Executes an executable program in the background and returns the process id to **GAUSS**.

Format

pid = **execbg**(*program*, *comline*);

Input

program	string, the name of the program, including the extension, to be executed.

comline	string, the arguments to be placed on the command line of the program being executed.

Output

pid scalar, the process id of the executable returned by program.

If **execbg** cannot execute program, the error returns will be negative:

-1	file not found
-2	the file is not an executable file
-3	not enough memory
-4	command line too long

Example

```
y = execbg("atog.exe","comd1.cmd");
if (y < 0);
   errorlog"atog failed";
   end;
endif;
```

In this example, the ATOG ASCII conversion utility is executed under the **execbg** function. The name of the command file to be used, `comd1.cmd`, is passed to ATOG on its command line. The returned value, *y*, is tested to see whether ATOG was successful. If not successful the program terminates after printing an error message. See **ATOG**, Chapter 1.

exp

Purpose

Calculates the exponential function.

Format

y = **exp**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing *e*, the base of natural logs, raised to the powers given by the elements of *x*.

Example

```
x = eye(3);
y = exp(x);
```

```
    1.000000   0.000000   0.000000
x = 0.000000   1.000000   0.000000
    0.000000   0.000000   1.000000
```

```
            2.718282  1.000000  1.000000
        y = 1.000000  2.718282  1.000000
            1.000000  1.000000  2.718282
```

This example creates a 3x3 identity matrix and computes the exponential function for each one of its elements. Note that **exp**(1) returns *e*, the base of natural logs.

See Also

ln

extern (dataloop)

Purpose

Allows access to matrices or strings in memory from inside a data loop.

Format

```
extern variable_list;
```

Remarks

Commas in *variable_list* are optional.

`extern` tells the translator not to generate local code for the listed variables, and not to assume that they are elements of the input data set.

`extern` statements should be placed before any reference to the symbols listed. The specified names should not exist in the input data set, or be used in a `make` statement.

Example

This example shows how to assign the contents of an external vector to a new variable in the data set, by iteratively assigning a range of elements to the variable. The reserved variable *x_x* contains the data read from the input data set on each iteration. The external vector must have at least as many rows as the data set.

```
base = 1;     /* used to index a range of */
              /* elements from exvec */
dataloop oldata newdata;
extern base, exvec;
make ndvar = exvec[seqa(base,1, rows(x_x))];
# base = base + rows(x_x); /* execute command */
                            /* literally */
endata;
```

external

Purpose

Lets the compiler know about symbols that are referenced above or in a separate file from their definitions.

Format

```
    external proc dog, cat;
    external keyword dog;
    external fn dog;
    external matrix x, y, z;
    external string mstr, cstr;
    external array a, b;
    external sparse matrix sma, smb;
    external struct structure_type sta, stb;
```

Remarks

See **Procedures and Keywords**, Chapter 1.

You may have several procedures in different files that reference the same global variable. By placing an `external` statement at the top of each file, you can let the compiler know what the type of the symbol is. If the symbol is listed and strongly typed in an active library, no `external` statement is needed.

If a matrix, string, N-dimensional array, sparse matrix, or structure appears in an `external` statement, it needs to appear once in a `declare` statement. If no declaration is found, an **`Undefined symbol`** error message will result.

Example

Let us suppose that you created a set of procedures defined in different files, which all set a global matrix *`_errcode`* to some scalar error code if errors were encountered.

You could use the following code to call one of the procedures in the set and check whether it succeeded:

```
    external matrix _errcode;

    x = rndn(10,5);
    y = myproc1(x);
    if _errcode;
       print "myproc1 failed";
       end;
    endif;
```

Without the `external` statement, the compiler would assume that *`_errcode`* was a procedure and incorrectly compile this program. The file containing the **`myproc1`** procedure must also contain an `external` statement that defines *`_errcode`* as a matrix, but this would not be encountered by the compiler until the `if` statement containing the reference to *`_errcode`* in the main program file had already been incorrectly compiled.

See Also

declare

eye

Purpose

Creates an identity matrix.

Format

y = **eye**(*n*);

Input

n	scalar, size of identity matrix to be created.

Output

y	*n*x*n* identity matrix.

Remarks

If *n* is not an integer, it will be truncated to an integer.

The matrix created will contain 1's down the diagonal and 0's everywhere else.

Example

```
    x = eye(3);
```

The code above assigns *x* to be equal to:

```
    1.0000 0.0000 0.0000
    0.0000 1.0000 0.0000
    0.0000 0.0000 1.0000
```

See Also

zeros, ones

f

fcheckerr

Purpose

Gets the error status of a file.

Format

err = **fcheckerr**(*f*);

Input

f	scalar, file handle of a file opened with **fopen**.

Output

err	scalar, error status.

Remarks

If there has been a read or write error on a file, **fcheckerr** returns 1, otherwise 0.

If you pass **fcheckerr** the handle of a file opened with open (i.e., a data set or matrix file), your program will terminate with a fatal error.

fclearerr

Purpose

Gets the error status of a file, then clears it.

Format

err = **fclearerr**(*f*);

Input

f	scalar, file handle of a file opened with **fopen**.

Output

err	scalar, error status.

Remarks

Each file has an error flag that gets set when there is an I/O error on the file. Typically, once this flag is set, you can no longer do I/O on the file, even if the error is a recoverable one. **fclearerr** clears the file's error flag, so you can attempt to continue using it.

If there has been a read or write error on a file, **fclearerr** returns 1, otherwise 0.

If you pass **fclearerr** the handle of a file opened with open (i.e., a data set or matrix file), your program will terminate with a fatal error.

The flag accessed by **fclearerr** is not the same as that accessed by **fstrerror**.

feq,fge,fgt,fle,flt,fne

Purpose

Fuzzy comparison functions. These functions use *_fcmptol* to fuzz the comparison operations to allow for roundoff error.

Format

```
y = feq(a, b);
y = fge(a, b);
y = fgt(a, b);
fle(a, b);
```

Input

a	NxK matrix, first matrix.
b	LxM matrix, second matrix, ExE compatible with *a*.

Global Input

_fcmptol	scalar, comparison tolerance. The default value is 1.0e-15.

Output

y	scalar, 1 (TRUE) or 0 (FALSE).

Remarks

The return value is TRUE if every comparison is TRUE.

The statement:

```
y = feq(a,b);
```

is equivalent to:

```
y = a eq b;
```

For the sake of efficiency, these functions are not written to handle missing values. If *a* and *b* contain missing values, use **missrv** to convert the missing values to something appropriate before calling a fuzzy comparison function.

The calling program can reset *_fcmptol* before calling these procedures:

```
_fcmptol = 1e-12;
```

Example

```
_fcmptol = 1e-12;

x = rndu(2,2);

y = x + 0.5*(_fcmptol);

if fge(x,y);
```

```
        print "each element of x is greater than";
        print "or equal to each element of y";
    else;
        print "at least one element of x is less";
        print "its corresponding element in y";
    endif;
```

Source

`fcompare.src`

See Also

dotfeq-dotfne

feqmt,fgemt,fgtmt,flemt,fltmt,fnemt

Purpose

Fuzzy comparison functions. These functions use the *fcmptol* argument to fuzz the comparison operations to allow for roundoff error.

Format

y = **feqmt**(*a*, *b*, *fcmptol*);
y = **fgemt**(*a*, *b*, *fcmptol*);
y = **fgtmt**(*a*, *b*, *fcmptol*);
y = **flemt**(*a*, *b*, *fcmptol*);
y = **fltmt**(*a*, *b*, *fcmptol*);
y = **fnemt**(*a*, *b*, *fcmptol*);

Input

a	NxK matrix, first matrix.
b	LxM matrix, second matrix, ExE compatible with *a*.
fcmptol	scalar, comparison tolerance.

Output

y	scalar, 1 (TRUE) or 0 (FALSE).

Remarks

The return value is TRUE if every comparison is TRUE.

The statement:

```
    y = feqmt(a,b,1e-15);
```

is equivalent to:

```
    y = a eq b;
```

For the sake of efficiency, these functions are not written to handle missing values. If *a* and *b* contain missing values, use **missrv** to convert the missing values to something appropriate before calling a fuzzy comparison function.

Example

```
    tol = 1e-12;
    x = rndu(2,2);

    y = x + 0.5*(tol);

    iffgemt(x,y,tol);
       print "each element of x is greater than";
       print "or equal to each element of y";
    else;
       print "at least one element of x is less";
       print "its corresponding element in y";
    endif;
```

Source

fcomparemt.src

See Also

dotfeqmt-dotfnemt

fflush

Purpose

Flushes a file's output buffer.

Format

```
ret = fflush(f);
```

Input

f	scalar, file handle of a file opened with **`fopen`**.

Output

ret	scalar, 0 if successful, -1 if not.

Remarks

If **`fflush`** fails, you can call **`fstrerror`** to find out why.

If you pass **`fflush`** the handle of a file opened with `open` (i.e., a data set or matrix file), your program will terminate with a fatal error.

fft

Purpose

Computes a 1- or 2-D Fast Fourier transform.

Format

y = **`fft`**(*x*);

Input

x	NxK matrix.

Output

y	LxM matrix, where L and M are the smallest powers of 2 greater than or equal to N and K, respectively.

Remarks

This computes the FFT of *x*, scaled by 1/N.

This uses a Temperton Fast Fourier algorithm.

If N or K is not a power of 2, *x* will be padded out with zeros before computing the transform.

See Also

ffti, rfft, rffti

ffti

Purpose

Computes an inverse 1- or 2-D Fast Fourier transform.

Format

y = **`ffti`**(*x*);

Input

x	NxK matrix.

Output

y	LxM matrix, where L and M are the smallest prime factor products greater than or equal to N and K, respectively.

Remarks

Computes the inverse FFT of *x*, scaled by 1/N.

This uses a Temperton prime factor Fast Fourier algorithm.

See Also

fft, rfft, rffti

fftm

Purpose

Computes a multi-dimensional FFT.

Format

y = **fftm**(*x*, *dim*);

Input

x	Mx1 vector, data.
dim	Kx1 vector, size of each dimension.

Output

y	Lx1 vector, FFT of *x*.

Remarks

The multi-dimensional data are laid out in a recursive or heirarchical fashion in the vector *x*. That is to say, the elements of any given dimension are stored in sequence left to right within the vector, with each element containing a sequence of elements of the next smaller dimension. In abstract terms, a 4-dimensional 2x2x2x2 hypercubic *x* would consist of two cubes in sequence, each cube containing two matrices in sequence, each matrix containing two rows in sequence, and each row containing two columns in sequence. Visually, *x* would look something like this:

$$X_{hyper} = X_{cube1} | X_{cube2}$$
$$X_{cube1} = X_{mat1} | X_{mat2}$$
$$X_{mat1} = X_{row1} | X_{row2}$$

Or, in an extended **GAUSS** notation, *x* would be:

```
    Xhyper = x[1,.,.,.] | x[2,.,.,.];
    Xcube1 = x[1,1,.,.] | x[1,2,.,.];
    Xmat1  = x[1,1,1,.] | x[1,1,2,.];
    Xrow1  = x[1,1,1,1] | x[1,1,1,2];
```

To be explicit, *x* would be laid out like this:

```
    x[1,1,1,1] x[1,1,1,2] x[1,1,2,1] x[1,1,2,2]
    x[1,2,1,1] x[1,2,1,2] x[1,2,2,1] x[1,2,2,2]
    x[2,1,1,1] x[2,1,1,2] x[2,1,2,1] x[2,1,2,2]
    x[2,2,1,1] x[2,2,1,2] x[2,2,2,1] x[2,2,2,2]
```

If you look at the last diagram for the layout of *x*, you'll notice that each line actually constitutes the elements of an ordinary matrix in normal row-major order. This is easy to achieve with **vecr**. Further, each pair of lines or "matrices" constitutes one of the desired cubes, again with all the elements in the correct order. And finally, the two cubes combine to form the hypercube. So, the process of construction is simply a sequence of concatenations of column vectors, with a **vecr** step if necessary to get started.

Here's an example, this time working with a 2x3x2x3 hypercube.

```
    let dim = 2 3 2 3;
    let x1[2,3] = 1 2 3 4 5 6;
    let x2[2,3] = 6 5 4 3 2 1;
    let x3[2,3] = 1 2 3 5 7 11;
    xc1 = vecr(x1)|vecr(x2)|vecr(x3); /* cube 1 */
    let x1 = 1 1 2 3 5 8;
    let x2 = 1 2 6 24 120 720;
    let x3 = 13 17 19 23 29 31;
    xc2 = x1|x2|x3;                  /* cube 2 */

    xh = xc1|xc2;                    /* hypercube */
    xhfft = fftm(xh,dim);

    let dimi = 2 4 2 4;
    xhffti = fftmi(xhfft,dimi);
```

We left out the **vecr** step for the 2nd cube. It's not really necessary when you're constructing the matrices with `let` statements.

dim contains the dimensions of *x*, beginning with the highest dimension. The last element of *dim* is the number of columns, the next to the last element of *dim* is the number of rows, and so on. Thus

```
    dim = { 2, 3, 3 };
```

indicates that the data in *x* is a 2x3x3 three-dimensional array, i.e., two 3x3 matrices of data. Suppose that *x1* is the first 3x3 matrix and *x2* the second 3x3 matrix, then:

```
    x = vecr(x1)|vecr(x2)
```

The size of *dim* tells you how many dimensions *x* has.

The arrays have to be padded in each dimension to the nearest power of two. Thus the output array can be larger than the input array. In the 2x3x2x3 hypercube example, *x* would be padded from 2x3x2x3 out to 2x4x2x4. The input vector would contain 36 elements, while the output vector would contain 64 elements. You may have noticed that we used a *dimi* with padded values at the end of the example to check our answer.

Source

`fftm.src`

See Also

fftmi, fft, ffti, fftn

fftmi

Purpose

Computes a multi-dimensional inverse FFT.

Format

y = **fftmi**(*x*, *dim*);

Input

x	Mx1 vector, data.

dim	Kx1 vector, size of each dimension.

Output

y	Lx1 vector, inverse FFT of *x*.

Remarks

The multi-dimensional data are laid out in a recursive or heirarchical fashion in the vector *x*. That is to say, the elements of any given dimension are stored in sequence left to right within the vector, with each element containing a sequence of elements of the next smaller dimension. In abstract terms, a 4-dimensional 2x2x2x2 hypercubic *x* would consist of two cubes in sequence, each cube containing two matrices in sequence, each matrix containing two rows in sequence, and each row containing two columns in sequence. Visually, *x* would look something like this:

$$X_{hyper} = X_{cube1} | X_{cube2}$$
$$X_{cube1} = X_{mat1} | X_{mat2}$$
$$X_{mat1} = X_{row1} | X_{row2}$$

Or, in an extended **GAUSS** notation, *x* would be:

```
Xhyper = x[1,.,.,.] | x[2,.,.,.];
Xcube1 = x[1,1,.,.] | x[1,2,.,.];
Xmat1 = x[1,1,1,.] | x[1,1,2,.];
Xrow1 = x[1,1,1,1] | x[1,1,1,2];
```

To be explicit, *x* would be laid out like this:

```
x[1,1,1,1] x[1,1,1,2] x[1,1,2,1] x[1,1,2,2]
x[1,2,1,1] x[1,2,1,2] x[1,2,2,1] x[1,2,2,2]
```

```
x[2,1,1,1] x[2,1,1,2] x[2,1,2,1] x[2,1,2,2]
x[2,2,1,1] x[2,2,1,2] x[2,2,2,1] x[2,2,2,2]
```

If you look at the last diagram for the layout of *x*, you'll notice that each line actually constitutes the elements of an ordinary matrix in normal row-major order. This is easy to achieve with **vecr**. Further, each pair of lines or "matrices" constitutes one of the desired cubes, again with all the elements in the correct order. And finally, the two cubes combine to form the hypercube. So, the process of construction is simply a sequence of concatenations of column vectors, with a **vecr** step if necessary to get started.

Here's an example, this time working with a 2x3x2x3 hypercube.

```
    let dim = 2 3 2 3;

    let x1[2,3] = 1 2 3 4 5 6;
    let x2[2,3] = 6 5 4 3 2 1;
    let x3[2,3] = 1 2 3 5 7 11;
    xc1 = vecr(x1)|vecr(x2)|vecr(x3); /* cube 1 */
    let x1 = 1 1 2 3 5 8;
    let x2 = 1 2 6 24 120 720;
    let x3 = 13 17 19 23 29 31;
    xc2 = x1|x2|x3;                  /* cube 2 */

    xh = xc1|xc2;                    /* hypercube */
    xhffti = fftmi(xh,dim);
```

We left out the **vecr** step for the 2nd cube. It's not really necessary when you're constructing the matrices with `let` statements.

`dim` contains the dimensions of *x*, beginning with the highest dimension. The last element of *`dim`* is the number of columns, the next to the last element of *`dim`* is the number of rows, and so on. Thus

```
    dim = { 2, 3, 3 };
```

indicates that the data in *x* is a 2x3x3 three-dimensional array, i.e., two 3x3 matrices of data. Suppose that *x1* is the first 3x3 matrix and *x2* the second 3x3 matrix, then

```
    x = vecr(x1)|vecr(x2)
```

The size of *dim* tells you how many dimensions *x* has.

The arrays have to be padded in each dimension to the nearest power of two. Thus the output array can be larger than the input array. In the 2x3x2x3 hypercube example, *x* would be padded from 2x3x2x3 out to 2x4x2x4. The input vector would contain 36 elements, while the output vector would contain 64 elements.

Source

`fftm.src`

See Also

fft, ffti, fftn

fftn

Purpose

Computes a complex 1- or 2-D FFT.

Format

y = **fftn**(*x*);

Input

x	NxK matrix.

Output

y	LxM matrix, where L and M are the smallest prime factor products greater than or equal to N and K, respectively.

Remarks

`fftn` uses the Temperton prime factor FFT algorithm. This algorithm can compute the FFT of any vector or matrix whose dimensions can be expressed as the product of selected prime number factors. **GAUSS** implements the Temperton algorithm for any power of 2, 3, and 5, and one factor of 7. Thus, **`fftn`** can handle any matrix whose dimensions can be expressed as

$$2^p \times 3^q \times 5^r \times 7^s$$

where *p*, *q* and *r* are nonnegative integers and *s* is equal to 0 or 1.

If a dimension of *x* does not meet this requirement, it will be padded with zeros to the next allowable size before the FFT is computed.

`fftn` pads matrices to the next allowable dimensions; however, it generally runs faster for matrices whose dimensions are highly composite numbers, i.e., products of several factors (to various powers), rather than powers of a single factor. For example, even though it is bigger, a 33600x1 vector can compute as much as 20% faster than a 32768x1 vector, because 33600 is a highly composite number, 2^6x3x5^2x7, whereas 32768 is a simple power of 2, 2^{15}. For this reason, you may want to hand-pad matrices to optimum dimensions before passing them to **`fftn`**. The **Run-Time Library** includes a routine, **`optn`**, for determining optimum dimensions.

The **Run-Time Library** also includes the **`nextn`** routine, for determining allowable dimensions for a matrix. (You can use this to see the dimensions to which **`fftn`** would pad a matrix.)

`fftn` scales the computed FFT by 1/(L*M).

See Also

fft, ffti, fftm, fftmi, rfft, rffti, rfftip, rfftn, rfftnp, rfftp

fgets

Purpose

Reads a line of text from a file.

Format

str = **fgets**(*f*, *maxsize*);

Input

f	scalar, file handle of a file opened with **fopen**.
maxsize	scalar, maximum size of string to read in, including the terminating null byte.

Output

str	string.

Remarks

fgets reads text from a file into a string. It reads up to a newline, the end of the file, or *maxsize*-1 characters. The result is placed in *str*, which is then terminated with a null byte. The newline, if present, is retained.

If the file is already at end-of-file when you call **`fgets`**, your program will terminate with an error. Use **`eof`** in conjunction with **`fgets`** to avoid this.

If the file was opened for update (see **`fopen`**) and you are switching from writing to reading, don't forget to call **`fseek`** or **`fflush`** first, to flush the file's buffer.

If you pass **`fgets`** the handle of a file opened with `open` (i.e., a data set or matrix file), your program will terminate with a fatal error.

See Also

fgetst, fgetsa, fopen

fgetsa

Purpose

Reads lines of text from a file into a string array.

Format

sa = **fgetsa**(*f*, *numl*);

Input

f	scalar, file handle of a file opened with **`fopen`**.
numl	scalar, number of lines to read.

Output

sa	Nx1 string array, N <= *numl*.

Remarks

fgetsa reads up to *numl* lines of text. If **fgetsa** reaches the end of the file before reading *numl* lines, *sa* will be shortened. Lines are read in the same manner as **fgets**, except that no limit is placed on the size of a line. Thus, **fgetsa** always returns complete lines of text. Newlines are retained. If *numl* is 1, **fgetsa** returns a string. (This is one way to read a line from a file without placing a limit on the length of the line.)

If the file is already at end-of-file when you call **fgetsa**, your program will terminate with an error. Use **eof** in conjunction with **fgetsa** to avoid this. If the file was opened for update (see **fopen**) and you are switching from writing to reading, don't forget to call **fseek** or **fflush** first, to flush the file's buffer.

If you pass **fgetsa** the handle of a file opened with `open` (i.e., a data set or matrix file), your program will terminate with a fatal error.

See Also

fgetsat, fgets, fopen

fgetsat

Purpose

Reads lines of text from a file into a string array.

Format

sa = **fgetsat**(*f*, *numl*);

Input

f	scalar, file handle of a file opened with **fopen**.
numl	scalar, number of lines to read.

Output

sa	Nx1 string array, N <= *numl*.

Remarks

fgetsat operates identically to **fgetsa**, except that newlines are not retained as text is read into *sa*.

In general, you don't want to use **fgetsat** on files opened in binary mode (see **fopen**). **fgetsat** drops the newlines, but it does NOT drop the carriage returns that precede them on some platforms. Printing out such a string array can produce unexpected results.

See Also

fgetsa, fgetst, fopen

fgetst

Purpose

Reads a line of text from a file.

Format

```
str = fgetst(f, maxsize);
```

Input

f	scalar, file handle of a file opened with **fopen**.
maxsize	scalar, maximum size of string to read in, including the null terminating byte.

Output

str	string.

Remarks

fgetst operates identically to **fgets**, except that the newline is not retained in the string.

In general, you don't want to use **fgetst** on files opened in binary mode (see **fopen**). **fgetst** drops the newline, but it does NOT drop the preceding carriage return used on some platforms. Printing out such a string can produce unexpected results.

See Also

fgets, fgetsat, fopen

fileinfo

Purpose

Returns names and information for files that match a specification.

Format

{ *fnames*, *finfo* } = **fileinfo**(*fspec*);

Input

fspec	string, file specification. Can include path. Wildcards are allowed in *fspec*.

Output

fnames	Nx1 string array of all file names that match, null string if none are found.
finfo	Nx13 matrix, information about matching files.

UNIX/Linux

[N, 1]	filesystem ID
[N, 2]	inode number
[N, 3]	mode bit mask
[N, 4]	number of links
[N, 5]	user ID

`[N, 6]`	group ID
`[N, 7]`	device ID (char/block special files only)
`[N, 8]`	size in bytes
`[N, 9]`	last access time
`[N,10]`	last data modification time
`[N,11]`	last file status change time
`[N,12]`	preferred I/O block size
`[N,13]`	number of 512-byte blocks allocated

Windows

`[N, 1]`	drive number (A = 0, B = 1, etc.)
`[N, 2]`	n/a, 0
`[N, 3]`	mode bit mask
`[N, 4]`	number of links, always 1
`[N, 5]`	n/a, 0
`[N, 6]`	n/a, 0
`[N, 7]`	n/a, 0
`[N, 8]`	size in bytes
`[N, 9]`	last access time
`[N,10]`	last data modification time
`[N,11]`	creation time

`[N,12]`	n/a, 0
`[N,13]`	n/a, 0

`finfo` will be a scalar zero if no matches are found.

Remarks

`fnames` will contain file names only; any path information that was passed is dropped.

The time stamp fields (`finfo`[N,9:11]) are expressed as the number of seconds since midnight, Jan. 1, 1970, Coordinated Universal Time (UTC).

See Also

filesa

filesa

Purpose

Returns a string array of file names.

Format

y = **filesa**(*n*);

Input

n	string, file specification to search for. Can include path. Wildcards are allowed in *n*.

Output

y	Nx1 string array of all file names that match, or null string if none are found.

Remarks

y will contain file names only; any path information that was passed is dropped.

Example

```
    y = filesa("ch*");
```

In this example all files listed in the current directory that begin with "ch" will be returned.

```
    proc exist(filename);
       retp(not filesa(filename) $== "");
    endp;
```

This procedure will return 1 if the file exists or 0 if not.

See Also

fileinfo, shell

floor

Purpose

Round down toward -∞.

Format

y = **floor**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing the elements of *x* rounded down.

Remarks

This rounds every element in *x* down to the nearest integer.

Example

```
//Set the seed for repeatable random numbers
rndseed 9072345;

//Create random normal numbers with a standard
//deviation of 100
x = 100*rndn(2,2);

//Round the numbers down
f = floor(x);

//Format so numbers will print in decimal form rather than
```

```
    //scientific notation) and will show 2 digits after the
    //decimal point
    format /rd 8,2;

    print "************************";
    print "After running this code:";
    print "************************\n";
    print "x = " x;
    print "";
    print "and, f = " f;
```

produces:

```
    ************************
    After running this code:
    ************************


    x =
        0.11   314.05
      -80.87   103.73

    and, f =
        0.00   314.00
      -81.00   103.00
```

Notice in the code above, how the $\backslash n$ at the end of the statement printing the line of asterisks, inserts a newline.

See Also

ceil, round, trunc

fmod

Purpose

Computes the floating-point remainder of *x*/*y*.

Format

r = **fmod**(*x*, *y*);

Input

x	NxK matrix.
y	LxM matrix, ExE conformable with *x*.

Output

r	max(N,L) by max(K,M) matrix.

Remarks

Returns the floating-point remainder *r* of *x*/*y* such that $x = iy + r$, where *i* is an integer, *r* has the same sign as *x* and $|r| < |y|$.

Compare this with %, the modulo division operator. (See **Operators**, Chapter 1.)

Example

This example extracts all of the years which are evenly divisible by four, from a vector with all of the years between 1900 and 2000.

```
//Create a vector with all years from 1900 to 2000
//i.e. 1900, 1901, 1902...2000
yrs = seqa(1900, 1, 101);

//Create an empty matrix into which we can put our output
y4 = {};

//Loop through each element in yrs
for i(1, rows(yrs), 1);
   //If the 'i'th element of 'yrs' is evenly divisible by
   //4, vertically concatenate it on to the bottom of 'y4'
   if not fmod(yrs[i], 4);
      y4 = y4|yrs[i];
   endif;
endfor;

//No digits after the decimal place
format /rd 8,0;

//Split 'y4' into two columns, each with half of the data
//and print the columns next to each other
print y4[1:13]~y4[14:26];
```

produces:

```
         1900      1952
         1904      1956
         1908      1960
         1912      1964
         1916      1968
         1920      1972
         1924      1976
         1928      1980
         1932      1984
         1936      1988
```

```
        1940      1992
        1944      1996
        1948      2000
```

fn

Purpose

Allows user to create one-line functions.

Format

```
fn fn_name(args) = code_for_function;
```

Remarks

Functions can be called in the same way as other procedures.

Example

```
fn area(r) = pi*r*r;

a = area(4);
```

After the code above:

```
a = 50.625
```

fonts

Purpose

Loads fonts to be used in the graph. Note: this function is for the deprecated PQG graphics.

Library

`pgraph`

Format

`fonts`(*`str`*);

Input

`str`	string or character vector containing the names of fonts to be used in the plot. The following fonts are available:	
	Simplex	standard sans serif font.
	Simgrma	Simplex greek, math.
	Microb	bold and boxy.
	Complex	standard font with serif.

Remarks

The first font specified will be used for the axes numbers.

If *`str`* is a null string, or **`fonts`** is not called, Simplex is loaded by default.

For more information on how to select fonts within a text string, see **Publication Quality Graphics**, Chapter 1.

Source

`pgraph.src`

See Also

title, xlabel, ylabel, zlabel

fopen

Purpose

Opens a file.

Format

f = **fopen**(*filename*, *omode*);

Input

filename	string, name of file to open.
omode	string, file I/O mode. (See Remarks, below.)

Output

f	scalar, file handle.

Portability

UNIX

Carriage return-linefeed conversion for files opened in text mode is unnecessary, because in UNIX a newline is simply a linefeed.

Remarks

`filename` can contain a path specification.

`omode` is a sequence of characters that specify the mode in which to open the file. The first character must be one of:

`r`	Open an existing file for reading. If the file does not exist, **`fopen`** fails.
`w`	Open or create a file for writing. If the file already exists, its current contents will be destroyed.
`a`	Open or create a file for appending. All output is appended to the end of the file.

To this can be appended a *+* and/or a *b*. The *+* indicates the file is to opened for reading and writing, or update, as follows:

`r+`	Open an existing file for update. You can read from or write to any location in the file. If the file does not exist, **`fopen`** fails.
`w+`	Open or create a file for update. You can read from or write to any location in the file. If the file already exists, its current contents will be destroyed.
`a+`	Open or create a file for update. You can read from any location in the file, but all output will be appended to the end of the file.

Finally, the `b` indicates whether the file is to be opened in text or binary mode. If the file is opened in binary mode, the contents of the file are read verbatim; likewise, anything output to the file is written verbatim. In text mode (the default), carriage return-linefeed sequences are converted on input to linefeeds, or newlines. Likewise on output, newlines are converted to carriage return-linefeeds. Also in text mode, if a `CTRL+Z` (char 26) is encountered during a read, it is interpreted as an end-of-file character, and reading ceases. In binary mode, `CTRL+Z` is read in uninterpreted.

The order of *+* and `b` is not significant; *`rb+`* and *`r+b`* mean the same thing.

You can both read from and write to a file opened for update. However, before switching from one to the other, you must make an **`fseek`** or **`fflush`** call, to flush the file's buffer.

If **`fopen`** fails, it returns a 0.

Use **`close`** and `closeall` to close files opened with **`fopen`**.

See Also

fseek, close, closeall

for

Purpose

Begins a `for` loop.

Format

```
for i (start, stop, step);
.
.
.
endfor;
```

Input

i	literal, the name of the counter variable.
start	scalar expression, the initial value of the counter.
stop	scalar expression, the final value of the counter.
step	scalar expression, the increment value.

Remarks

The counter is strictly local to the loop. The expressions, *start*, *stop* and *step* are evaluated only once when the loop initializes and are stored local to the loop.

The `for` loop is optimized for speed and much faster than a `do` loop.

The commands `break` and `continue` are supported. The `continue` command steps the counter and jumps to the top of the loop. The `break` command terminates the current loop.

The loop terminates when the value of *i* exceeds *stop*. If `break` is used to terminate the loop and you want the final value of the counter, you need to assign it to a variable before the `break` statement (see the third example, following).

Example

Example 1

```
x = zeros(10, 5);
for i (1, rows(x), 1);
   for j (1, cols(x), 1);
      x[i,j] = i*j;
    endfor;
endfor;
```

Example 2

```
x = rndn(3,3);
y = rndn(3,3);

for i (1, rows(x), 1);
   for j (1, cols(x), 1);
      if x[i,j] >= y[i,j];
         continue;
      endif;
      temp = x[i,j];
      x[i,j] = y[i,j];
      y[i,j] = temp;
   endfor;
endfor;
```

Example 3

```
li = 0;
x = rndn(100,1);
y = rndn(100,1);

for i (1, rows(x), 1);
   if x[i] /= y[i];
```

```
        li = i;
        break;
     endif;
  endfor;

  if li;
     print "Compare failed on row " li;
  endif;
```

format

Purpose

Controls the format of matrices and numbers printed out with `print` statements.

Format

`format` [[/*typ*]] [[/*fmted*]] [[/*mf*]] [[/*jnt*]] [[*f*,*p*]]

Input

/*typ*	literal, symbol type flag(s). Indicate which symbol types you are setting the output format for.	
	/*mat*, /*sa*, /*str*	Formatting parameters are maintained separately for matrices and arrays (/*mat*), sstring arrays (/*sa*), and strings (/*str*). You can specify more than one /*typ* flag; the format will be set forall types indicated. If no /*typ* flag is listed, `format` assumes /*mat*.

`/fmted`	literal, enable formatting flag.	
	`/on`, `/off`	Enable/disable formatting. When formatting is disabled, the contents of a variable are dumped to the screen in a "raw" format. `/off` is currently supported only for strings. "Raw" format for strings means that the entire string is printed, starting at the current cursor position. When formatting is enabled for strings, they are handled the same as string arrays. This shouldn't be too surprising, since a string is actually a 1x1 string array.
`/mf`	literal, matrix row format flag.	
	`/m0`	no delimiters before or after rows when printing out matrices.
	`/m1 or /mb1`	print 1 carriage return/line feed pair before each row of a matrix with more than 1 row.
	`/m2 or /mb2`	print 2 carriage return/line feed pairs before each row of a matrix with more than 1 row.
	`/m3 or /mb3`	print "Row 1", "Row 2"... before each row of a matrix with more than one row.
	`/ma1`	print 1 carriage return/line feed pair after each row of a matrix with more than 1 row.
	`/ma2`	print 2 carriage return/line feed pairs after each row of a matrix with more than 1 row.
	`/a1`	print 1 carriage return/line feed pair after each row of a matrix.

	`/a2`	print 2 carriage return/line feed pairs after each row of a matrix.
	`/b1`	print 1 carriage return/line feed pair before each row of a matrix.
	`/b2`	print 2 carriage return/line feed pairs before each row of a matrix.
	`/b3`	print "Row 1", "Row 2"... before each row of a matrix.

/jnt literal, matrix element format flag - controls justification, notation and trailing character.

Right-Justified

`/rd`	Signed decimal number in the form `####.####`, where `>####` is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed.
`/re`	Signed number in the form `#.##E±###`, where `#` is one decimal digit, `##` is one or more decimal digits depending on the precision, and `###` is three decimal digits. If precision is 0, the form will be `[-]#E±###` with no decimal point printed.
`/ro`	This will give a format like `/rd` or `/re` depending on which is most compact for the

	number being printed. A format like `/re` will be used only if the exponent value is less than -4 or greater than the precision. If a `/re` format is used, a decimal point will always appear. The precision signifies the number of significant digits displayed.
`/rz`	This will give a format like `/rd` or `/re` depending on which is most compact for the number being printed. A format like `/re` will be used only if the exponent value is less than -4 or greater than the precision. If a `/re` format is used, trailing zeros will be supressed and a decimal point will appear only if one or more digits follow it. The precision signifies the number of significant digits displayed.

Left-Justified

`/ld`	Signed decimal number in the form `[-] ####.####`, where `####` is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed. If the number is positive, a space character will replace the leading minus sign.
`/le`	Signed number in the form `[-]#.##E±###`, where `#` is one decimal digit, `##` is one or more decimal digits depending on the precision, and

`###` is three decimal digits. If precision is 0, the form will be `[-]#E±###` with no decimal point printed. If the number is positive, a space character will replace the leading minus sign.

`/lo`	This will give a format like `/ld` or `/le` depending on which is most compact for the number being printed. A format like `/le` will be used only if the exponent value is less than -4 or greater than the precision. If a `/le` format is used, a decimal point will always appear. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.
`/lz`	This will give a format like `/ld` or `/le` depending on which is most compact for the number being printed. A format like `/le` will be used only if the exponent value is less than -4 or greater than the precision. If a `/le` format is used, trailing zeros will be supressed and a decimal point will appear only if one or more digits follow it. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.

Trailing Character

The following characters can be added to the `/jnt` parameters above to control the trailing character if any:

```
    format /rdn 1,3;
```

s	The number will be followed immediately by a space character. This is the default.
c	The number will be followed immediately by a comma.
t	The number will be followed immediately by a tab character.
n	No trailing character.
f	scalar expression, controls the field width.
p	scalar expression, controls the precision.

Remarks

If character elements are to be printed, the precision should be at least 8 or the elements will be truncated. This does not affect the string data type.

For numeric values in matrices, *p* sets the number of significant digits to be printed. For string arrays, strings, and character elements in matrices, *p* sets the number of characters to be printed. If a string is shorter than the specified precision, the entire string is printed. For string arrays and strings, *p* = -1 means print the entire string, regardless of its length. *p* = -1 is illegal for matrices; setting *p*>= 8 means the same thing for character elements.

The */xxx* slash parameters are optional. Field and precision are optional also, but if one is included, then both must be included.

Slash parameters, if present, must precede the field and precision parameters.

A `format` statement stays in effect until it is overridden by a new `format` statement. The slash parameters may be used in a `print` statement to override the current default.

f and *p* may be any legal expressions that return scalars. Nonintegers will be truncated to integers.

The total width of field will be overridden if the number is too big to fit into the space allotted. For instance, **format** */rds 1,0* can be used to print integers with a single space between them, regardless of the magnitudes of the integers.

Complex numbers are printed with the sign of the imaginary half separating them and an "*i*" appended to the imaginary half. Also, the field parameter refers to the width of field for each half of the number, so a complex number printed with a field of 8 will actually take (at least) 20 spaces to print. The character printed after the imaginary part can be changed (for example, to a "*j*") with the **sysstate** function, case 9.

The default when **GAUSS** is first started is:

```
format /mb1 /ros 16,8;
```

Example

This code:

```
x = rndn(3,3);

format /m1 /rd 16,8;
print x;
```

produces:

```
 2.25240104       0.53724423      -0.67744907
-0.16183998       1.57152099       1.33836836
 0.00666162      -1.24948147      -0.77987532
```

This code:

```
format /m1 /rzs 1,10;
print x;
```

produces:

```
2.252401038 0.5372442301 -0.6774490661
-0.1618399808 1.571520994 1.338368355
0.00666161784 -1.24948147 -0.7798753222
```

This code:

```
format /m3 /rdn 16,4;
print x;
```

produces:

```
print x;

Row 1
          2.2524          0.5372          -0.6774
Row 2
         -0.1618          1.5715           1.3384
Row 3
          0.0067         -1.2495          -0.7799
```

This code:

```
format /m1 /ldn 16,4;
print x;
```

produces:

```
 2.2524          0.5372          -0.6774
-0.1618          1.5715           1.3384
 0.0067         -1.2495          -0.7799
```

This code:

```
format /m1 /res 12,4;
print x;
```

produces:

```
     2.2524e+000  5.3724e-001 -6.7745e-001
    -1.6184e-001  1.5715e+000  1.3384e+000
     6.6616e-003 -1.2495e+000 -7.7988e-001
```

See Also

formatcv, formatnv, print, output

formatcv

Purpose

Sets the character data format used by **`printfmt`**.

Format

`oldfmt` = **`formatcv`**(*`newfmt`*);

Input

`newfmt`	1x3 vector, the new format specification.

Output

`oldfmt`	1x3 vector, the old format specification.

Remarks

See **`printfm`** for details on the format vector.

Example

This example saves the old format, sets the format desired for printing *x*, prints *x*, then restores the old format. This code:

```
x = { A 1, B 2, C 3 };
oldfmt = formatcv("*.*s" ~ 3 ~ 3);
call printfmt(x,0~1);
call formatcv(oldfmt);
```

produces:

```
 A 1
 B 2
 C 3
```

Source

`gauss.src`

Globals

__fmtcv

See Also

formatnv, printfm, printfmt

formatnv

Purpose

Sets the numeric data format used by **`printfmt`**.

Format

```
oldfmt = formatnv(newfmt);
```

Input

newfmt	1x3 vector, the new format specification.

Output

oldfmt	1x3 vector, the old format specification.

Remarks

See **printfm** for details on the format vector.

Example

This example saves the old format, sets the format desired for printing *x*, prints *x*, then restores the old format. This code:

```
    x = { A 1, B 2, C 3 };
    oldfmt = formatnv("*.*lf" ~ 8 ~ 4);
    call printfmt(x,0~1);
    call formatnv(oldfmt);
```

produces:

```
     A 1.0000
     B 2.0000
     C 3.0000
```

Source

`gauss.src`

Globals

__fmtnv

See Also

formatcv, printfm, printfmt

fputs

Purpose

Writes strings to a file.

Format

numl = **`fputs`**(*f*, *sa*);

Input

f	scalar, file handle of a file opened with **`fopen`**.
sa	string or string array.

Output

numl	scalar, the number of lines written to the file.

Portability

UNIX

Carriage return-linefeed conversion for files opened in text mode is unnecessary, because in UNIX a newline is simply a linefeed.

Remarks

`fputs` writes the contents of each string in *`sa`*, minus the null terminating byte, to the file specified. If the file was opened in text mode (see **`fopen`**), any newlines present in the strings are converted to carriage return-linefeed sequences on output. If *`numl`* is not equal to the number of elements in *`sa`*, there may have been an I/O error while writing the file. You can use **`fcheckerr`** or **`fclearerr`** to check this. If there was an error, you can call **`fstrerror`** to find out what it was. If the file was opened for update (see **`fopen`**) and you are switching from reading to writing, don't forget to call **`fseek`** or **`fflush`** first, to flush the file's buffer. If you pass **`fputs`** the handle of a file opened with `open` (i.e., a data set or matrix file), your program will terminate with a fatal error.

See Also

fputst, fopen

fputst

Purpose

Writes strings to a file.

Format

`numl` = **`fputst`**(*`f`*, *`sa`*);

Input

f	scalar, file handle of a file opened with **fopen**.
sa	string or string array.

Output

numl	scalar, the number of lines written to the file.

Portability

UNIX

Carriage return-linefeed conversion for files opened in text mode is unnecessary, because in UNIX a newline is simply a linefeed.

Remarks

fputst works identically to **fputs**, except that a newline is appended to each string that is written to the file. If the file was opened in text mode (see **fopen**), these newlines are also converted to carriage return-linefeed sequences on output.

See Also

fputs, fopen

fseek

Purpose

Positions the file pointer in a file.

Format

```
ret = fseek(f, offs, base);
```

Input

f	scalar, file handle of a file opened with **fopen**.
offs	scalar, offset (in bytes).
base	scalar, base position.
	0 beginning of file.
	1 current position of file pointer.
	2 end of file.

Output

ret	scalar, 0 if successful, 1 if not.

Portability

UNIX

Carriage return-linefeed conversion for files opened in text mode is unnecessary, because in UNIX a newline is simply a linefeed.

Remarks

fseek moves the file pointer *offs* bytes from the specified *base* position. *offs* can be positive or negative. The call may fail if the file buffer needs to be flushed (see **fflush**).

If **fseek** fails, you can call **fstrerror** to find out why.

For files opened for update (see **fopen**), the next operation can be a read or a write.

fseek is not reliable when used on files opened in text mode (see **fopen**). This has to do with the conversion of carriage return-linefeed sequences to newlines. In particular, an **fseek** that follows one of the **fget***xxx* or **fput***xxx* commands may not produce the expected result. For example:

```
    p = ftell(f);
    s = fgetsa(f,7);
    call fseek(f,p,0);
```

is not reliable. We have found that the best results are obtained by **fseek**'ing to the beginning of the file and then **fseek**'ing to the desired location, as in

```
    p = ftell(f);
    s = fgetsa(f,7);
    call fseek(f,0,0);
    call fseek(f,p,0);
```

If you pass **fseek** the handle of a file opened with `open` (i.e., a data set or matrix file), your program will terminate with a fatal error.

See Also

fopen

fstrerror

Purpose

Returns an error message explaining the cause of the most recent file I/O error.

Format

s = **`fstrerror`**;

Output

s	string, error message.

Remarks

Any time an I/O error occurs on a file opened with **`fopen`**, an internal error flag is updated. (This flag, unlike those accessed by **`fcheckerr`** and **`fclearerr`**, is not specific to a given file; rather, it is system-wide.) **`fstrerror`** returns an error message based on the value of this flag, clearing it in the process. If no error has occurred, a null string is returned.

Since **`fstrerror`** clears the error flag, if you call it twice in a row, it will always return a null string the second time.

The Windows system command called by **`ftell`** does not set the internal error flag accessed by **`fstrerror`**. Therefore, calling **`fstrerror`** after **`ftell`** on Windows will not produce any error information.

See Also

fopen, ftell

ftell

Purpose

Gets the position of the file pointer in a file.

Format

pos = **`ftell`**(*f*);

Input

f	scalar, file handle of a file opened with **`fopen`**.

Output

pos	scalar, current position of the file pointer in a file.

Remarks

`ftell` returns the position of the file pointer in terms of bytes from the beginning of the file. The call may fail if the file buffer needs to be flushed (see **`fflush`**).

If an error occurs, **`ftell`** returns -1. You can call **`fstrerror`** to find out what the error was.

If you pass **`ftell`** the handle of a file opened with `open` (i.e., a data set or matrix file), your program will terminate with a fatal error.

See Also

fopen, fseek

ftocv

Purpose

Converts a matrix containing floating point numbers into a matrix containing the decimal character representation of each element.

Format

y = **ftocv**(*x*, *field*, *prec*);

Input

x	NxK matrix containing numeric data to be converted.
field	scalar, minimum field width.
prec	scalar, the numbers created will have *prec* places after the decimal point.

Output

y	NxK matrix containing the decimal character equivalent of the corresponding elements in *x* in the format defined by *field* and *prec*.

Remarks

If a number is narrower than *field*, it will be padded on the left with zeros.

If *prec* = 0, the decimal point will be suppressed.

Example

```
    y = seqa(6,1,5);
    x = 0 $+ "beta" $+ ftocv(y,2,0);
    print $x;
```

results in the following output:

```
         beta06
         beta07
         beta08
         beta09
         beta10
```

Notice that the (0 *$+*) above was necessary to force the type of the result to matrix because the string constant "*beta*" would be of type string. The left operand in an expression containing a *$+* operator controls the type of the result.

See Also

ftos

ftos

Purpose

Converts a scalar into a string containing the decimal character representation of that number.

Format

y = **ftos**(*x*, *fmat*, *field*, *prec*);

Input

x	scalar, the number to be converted.
fmat	string, the format string to control the conversion.
field	scalar or 2x1 vector, the minimum field width. If *field* is 2x1, it specifies separate field widths for the real and imaginary parts of *x*.
prec	scalar or 2x1 vector, the number of places following the decimal point. If *prec* is 2x1, it specifies separate precisions for the real and imaginary parts of *x*.

Output

y	string containing the decimal character equivalent of *x* in the format specified.

Remarks

The format string corresponds to the `format/jnt` (justification, notation, trailing character)slash parameter as follows:

```
/rdn     "%*.*lf"

/ren     "%*.*lE"

/ron     "%#*.*lG"

/rzn     "%*.*lG"
```

`/ldn`	`"%- *.*lf"`
`/len`	`"%- *.*lE"`
`/lon`	`"%-# *.*lG"`
`/lzn`	`"%- *.*lG"`

If *x* is complex, you can specify separate formats for the real and imaginary parts by putting two format specifications in the format string. You can also specify separate fields and precisions. You can position the sign of the imaginary part by placing a "+" between the two format specifications. If you use two formats, no "*i*" is appended to the imaginary part. This is so you can use an alternate format if you prefer, for example, prefacing the imaginary part with a "j".

The format string can be a maximum of 80 characters.

If you want special characters to be printed after *x*, include them as the last characters of the format string. For example:

`"%*.*lf,"`	right-justified decimal followed by a comma.
`"%-*.*s "`	left-justified string followed by a space.
`"%*.*lf"`	right-justified decimal followed by nothing.

You can embed the format specification in the middle of other text:

```
    "Time: %*.*lf seconds."
```

If you want the beginning of the field padded with zeros, then put a "*0*" before the first "*" in the format string:

`"%0*.*lf"`	right-justified decimal.

If *prec* = 0, the decimal point will be suppressed.

Example

You can create custom formats for complex numbers with **ftos**. For example,

```
    let c = 24.56124+6.3224e-2i;

    field = 1;
    prec = 3|5;
    fmat = "%lf + j%le is a complex number.";
    cc = ftos(c,fmat,field,prec);
```

results in

```
    cc = "24.561 + j6.32240e-02 is a complex number."
```

Some other things you can do with **ftos**:

```
    let x = 929.857435324123;
    let y = 5.46;
    let z = 5;

    field = 1;
    prec = 0;
    fmat = "%*.*lf";
    zz = ftos(z,fmat,field,prec);

    field = 1;
    prec = 10;
    fmat = "%*.*lE";
    xx = ftos(x,fmat,field,prec);

    field = 7;
    prec = 2;
    fmat = "%*.*lf seconds";
    s1 = ftos(x,fmat,field,prec);
```

```
s2 = ftos(y,fmat,field,prec);

field = 1;
prec = 2;
fmat = "The maximum resistance is %*.*lf ohms.";
om = ftos(x,fmat,field,prec);
```

The results:

```
zz = "5"

xx = "9.2985743532E+002"

s1 = "929.86 seconds"

s2 = "5.46 seconds"

om = "The maximum resistance is 929.86 ohms."
```

See Also

ftocv, stof, format

ftostrC

Purpose

Converts a matrix to a string array using a C language format specification.

Format

sa = **ftostrC**(*x*, *fmt*);

Input

x	NxK matrix, real or complex.
fmt	Kx1, 1xK or 1x1 string array containing format information.

Output

sa	NxK string array.

Remarks

If *fmt* has K elements, each column of *sa* can be formatted separately. If *x* is complex, there must be two format specifications in each element of *fmt*.

Example

```
declare string fmtr = { "%6.3lf",
                         "%11.8lf" };

declare string fmtc = { "(%6.3lf, %6.3lf)",
                         "(%11.8lf, %11.8lf)" };

xr = rndn(4, 2);
xc = sqrt(xr')';

sar = ftostrC(xr, fmtr);
sac = ftostrC(xc, fmtc);

print sar;
print sac;
```

produces:

```
      -0.166 1.05565441
      -1.590 -0.79283296
       0.130 -1.84886957
       0.789 0.86089687

      ( 0.000, -0.407) ( 1.02745044, 0.00000000)
      ( 0.000, -1.261) ( 0.00000000, -0.89041168)
      ( 0.361, 0.000) ( 0.00000000, -1.35973143)
      ( 0.888, 0.000) ( 0.92784529, 0.00000000)
```

See Also

strtof, strtofcplx

g

gamma

Purpose

Returns the value of the gamma function.

Format

y = **gamma**(x);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array.

Remarks

For each element of x this function returns the integral

$$\int_0^\infty t^{(x-1)} e^{-t} dt$$

All elements of *x* must be positive and less than or equal to 169. Values of *x* greater than 169 will cause an overflow.

The natural log of **gamma** is often what is required and it can be computed without the overflow problems of **gamma** using **lnfact**.

Example

```
    y = gamma(2.5);
```

After the code above:

```
    y = 1.329340
```

See Also

cdfchic, cdfbeta, cdffc, cdfnc, cdftc, erf, erfc, lnfact

gammacplx

Purpose

Computes the Gamma function for complex inputs.

Format

f = **gammacplx**(*z*);

Input

z	NxK matrix; *z* may be complex.

Output

`f`	NxK matrix; `f` may be complex.

Technical Notes

Accuracy is 15 significant digits along the real axis and 13 significant digits elsewhere. This routine uses the Lanczos series approximation for the complex Gamma function.

References

1. C. Lanczos, SIAM JNA 1, 1964, pp. 86-96.
2. Y. Luke, "The Special ... approximations," 1969, pp. 29-31.
3. Y. Luke, "Algorithms ... functions," 1977.
4. J. Spouge, SIAM JNA 31, 1994, pp. 931-944.
5. W. Press, "Numerical Recipes."
6. S. Chang, "Computation of special functions," 1996.
7. W. J. Cody "An Overview of Software Development for Special Functions," 1975.
8. P. Godfrey "A note on the computation of the convergent Lanczos complex Gamma approximation."
9. Original code by Paul Godfrey

gammaii

Purpose

Computes the inverse incomplete gamma function.

Format

```
x = gammaii(a, p);
```

Input

a	MxN matrix, exponents.
p	KxL matrix, ExE conformable with *a*, incomplete gamma values.

Output

x	max(M,K) by max(N,L) matrix, abscissae.

Source

`cdfchii.src`

Globals

_ginvinc, *__macheps*

gausset

Purpose

Resets the global control variables declared in `gauss.dec`.

Format

```
gausset;
```

Source

gauss.src

Globals

__altnam, *__con*, *__ff*, *__fmtcv*, *__fmtnv*, *__header*, *__miss*, *__output*, *__row*, *__rowfac*, *__sort*, *__title*, *__tol*, *__vpad*, *__vtype*, *__weight*

gdaAppend

Purpose

Appends data to a variable in a **GAUSS** Data Archive.

Format

```
ret = gdaAppend(filename, x,  varname);
```

Input

filename	string, name of data file.
x	matrix, array, string or string array, data to append.
varname	string, variable name.

Output

ret	scalar, return code, 0 if successful, otherwise one of the following error codes:	
	1	Null file name.
	2	File open error.
	3	File write error.
	4	File read error.
	5	Invalid data file type.
	8	Variable not found.
	10	File contains no variables.
	14	File too large to be read on current platform.
	17	Type mismatch.
	18	Argument wrong size.
	19	Data must be real.
	20	Data must be complex.

Remarks

This command appends the data contained in *x* to the variable *varname* in *filename*. Both *x* and the variable referenced by *varname* must be the same data type, and they must both contain the same number of columns.

Because **gdaAppend** increases the size of the variable, it moves the variable to just after the last variable in the data file to make room for the added data, leaving empty bytes in the variable's old location. It also moves the variable descriptor table, so it is not overwritten by the variable data. This does not change the index of the variable because variable indices are determined NOT by the order of the variable data in a GDA, but by the order of the variable descriptors. Call **gdaPack** to pack the data in a GDA, so it contains no empty bytes.

Example

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");

y = rndn(25,50);
ret = gdaAppend("myfile.gda",y,"x1");
```

This example adds 25*50=1250 elements to *x1*, making it a 125x50 matrix.

See Also

gdaWriteSome, gdaUpdate, gdaWrite

gdaCreate

Purpose

Creates a **GAUSS** Data Archive.

Format

ret = **gdaCreate**(*filename*, *overwrite*);

Input

filename	string, name of data file to create.
overwrite	scalar, one of the following:
	0 error out if file already exists.
	1 overwrite file if it already exists.

Output

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.
3	File write error.
6	File already exists.
7	Cannot create file.

Remarks

This command creates a **GAUSS** Data Archive containing only a header. To add data to the GDA, call **`gdaWrite`**.

It is recommended that you include a `.gda` extension in *filename*. However, **`gdaCreate`** will not force an extension.

Example

```
    ret = gdaCreate("myfile.gda",1);
```

See Also

gdaWrite

gdaDStat

Purpose

Computes descriptive statistics on multiple Nx1 variables in a **GAUSS** Data Archive.

Format

```
dout = gdaDStat(dc0, filename, vars);
```

Input

dc0	an instance of a **dstatmtControl** structure with the following members:		
	dc0.altnames	Kx1 string array of alternate variable names for the output. Default = "".	
	dc0.maxbytes	scalar, the maximum number of bytes to be read per iteration of the read loop. Default = 1e9.	
	dc0.maxvec	scalar, the largest number of elements allowed in any one matrix. Default = 20000.	
	dc0.miss	scalar, one of the following:	
		0	There are no missing values

			(fastest).
		1	Listwise deletion, drop a row if any missings occur in it.
		2	Pairwise deletion.
		Default = 0.	
	dc0.output	scalar, one of the following:	
		0	Do not print output table.
		1	Print output table.
		Default = 1.	
	dc0.row	scalar, the number of rows of *var* to be read per iteration of the read loop.	
		If 0, (default) the number of rows will be calculated using *dc0.maxbytes* and *dc0.maxvec*.	
filename	string, name of data file.		
vars	Kx1 string array, names of variables		
	- or -		
	Kx1 vector, indices of variables.		

Output

dout	an instance of a **dstatmtOut** structure with the following members:

`dout.vnames`	Kx1 string array, the names of the variables used in the statistics.
`dout.mean`	Kx1 vector, means.
`dout.var`	Kx1 vector, variance.
`dout.std`	Kx1 vector, standard deviation.
`dout.min`	Kx1 vector, minima.
`dout.max`	Kx1 vector, maxima.
`dout.valid`	Kx1 vector, the number of valid cases.
`dout.missing`	Kx1 vector, the number of missing cases.
`dout.errcode`	scalar, error code, 0 if successful, or one of the following:

1	No GDA indicated.
4	Not implemented for complex data.
5	Variable must be type matrix.
6	Too many variables specified.
7	Too many missings - no data left after packing.
8	Name variable wrong size.
9	*`altnames`* member of **dstatmtControl** structure wrong size.
11	Data read error.

Remarks

The variables referenced by `vars` must all be Nx1.

The names of the variables in the GDA will be used for the output by default. To use alternate names, set the `altnames` member of the **dstatmtControl** structure.

If pairwise deletion is used, the minima and maxima will be the true values for the valid data. The means and standard deviations will be computed using the correct number of valid observations for each variable.

Example

```
    //Execute structure definition
    #include ds.sdf

    struct dstatmtControl dc0;
    struct dstatmtOut dout;

    //Set structure to default values
    dc0 = dstatmtControlCreate;

    vars = { 1,4,5,8 };
    dout = gdaDStat(dc0,"myfile.gda",vars);
```

This example computes descriptive statistics on the first, fourth, fifth and eighth variables in `myfile.gda`.

Source

`gdadstat.src`

See Also

gdaDStatMat, dstatmtControlCreate

gdaDStatMat

Purpose

Computes descriptive statistics on a selection of columns from a matrix located in a **GAUSS** Data Archive.

Format

```
dout = gdaDStatMat(dc0, filename, gmat, colind, vnamevar);
```

Input

dc0 an instance of a **dstatmtControl** structure with the following members:

dc0.altnames	Kx1 string array of alternate variable names for the output. Default = "". If set, it must have the same number of rows as `colind`.
dc0.maxbytes	scalar, the maximum number of bytes to be read per iteration of the read loop. Default = 1e9.
dc0.maxvec	scalar, the largest number of elements allowed in any one matrix. Default = 20000.
dc0.miss	scalar, one of the following:

0	There are no missing values (fastest).
1	Listwise deletion, drop a row if any

			missings occur in it.
		2	Pairwise deletion.
		Default = 0.	
	`dc0.output`	scalar, one of the following:	
		0	Do not print output table.
		1	Print output table.
		Default = 1.	
	`dc0.row`	scalar, the number of rows of `var` to be read per iteration of the read loop. If 0, (default) the number of rows will be calculated using `dc0.maxbytes` and `dc0.maxvec`.	
`filename`	string, name of data file.		
`gmat`	string, name of matrix - or - scalar, index of matrix.		
`colind`	Kx1 vector, indices of columns in variable to use.		
`vnamevar`	string, name of the string containing the variable names in the matrix - or - scalar, index of the string containing the variable names in the matrix.		

Output

`dout`	an instance of a **dstatmtOut** structure with the following members:		
	`dout.vnames`	Kx1 string array, the names of the variables used in the statistics.	
	`dout.mean`	Kx1 vector, means.	
	`dout.var`	Kx1 vector, variance.	
	`dout.std`	Kx1 vector, standard deviation.	
	`dout.min`	Kx1 vector, minima.	
	`dout.max`	Kx1 vector, maxima.	
	`dout.valid`	Kx1 vector, the number of valid cases.	
	`dout.missing`	Kx1 vector, the number of missing cases.	
	`dout.errcode`	scalar, error code, 0 if successful, otherwise one of the following:	
		1	No GDA indicated.
		3	Variable must be Nx1.
		4	Not implemented for complex data.
		5	Variable must be type matrix.
		7	Too many missings, no data left after packing.
		9	`altnames` member of **dstatmtControl** structure wrong

	size.
11	Data read error.

Remarks

Set *colind* to a scalar 0 to use all of the columns in *var*.

vnamevar must either reference an Mx1 string array variable containing variable names, where M is the number of columns in the data set variable, or be set to a scalar 0. If *vnamevar* references an Mx1 string array variable, then only the elements indicated by *colind* will be used. Otherwise, if *vnamevar* is set to a scalar 0, then the variable names for the output will be generated automatically ("X1,X2,...,XK") unless the alternate variable names are set explicitly in the *altnames* member of the **dstatmtControl** structure.

If pairwise deletion is used, the minima and maxima will be the true values for the valid data. The means and standard deviations will be computed using the correct number of valid observations for each variable.

Example

In order to create a real, working example that you can use, you must first create a sample **GAUSS Data Archive** with the code below.

```
//Create an example GAUSS Data Archive
ret = gdaCreate("myfile.gda",1);

//Add a variable 'A' which is a 10x5 random normal matrix
ret = gdaWrite("myfile.gda",rndn(10,5),"A");

//Add a variable 'COLS' which is a 5x1 string array
string vnames = { "X1", "X2", "X3", "X4", "X5" };
```

```
ret = gdaWrite("myfile.gda", vnames, "COLS");
```

This code above will create a **GAUSS Data Archive** containing two variables, the **GAUSS** matrix *A* containing the data and *COLS* which contains the names for the columns of the matrix *A* which are the model variables (*X1*, *X2*,...).

The code below computes the statistics on each of the columns of the matrix *A*.

```
    #include dstatmt.sdf
    struct dstatmtControl dc0;
    struct dstatmtout dout;

    dc0 = dstatmtControlCreate;
    colind = { 1, 2, 3, 4, 5 };
    dout = gdaDStatMat(dc0, "myfile.gda", "A", colind, "COL
    S" );
```

The final input to **gdaDStatMat** above tells the function the names to use for the columns of *A*. In this example, you can reference the *COLS* variable by name as you see in the example below. Alternatively, you can access this variable by index. Since *COLS* is the second variable in the **GAUSS Data Archive** created at the start of this example, the following is equivalent to the last line above:

```
    dout = gdaDStatMat(dc0, "myfile.gda", "A", colind, 2 );
```

If you wanted to calculate the statistics on just the first, third and fifth columns of *A*:

```
    colind = { 1, 3, 5 };
    dout = gdaDStatMat(dc0, "myfile.gda", "A", colind, "COL
    S" );
```

Notice in these lines above that *COLS* still contains all of the variable names i.e. *X1*, *X2*, *X3*, *X4* and *X5*. *COLS* should always contain the full list of all variables in the matrix *A*.

Source

gdadstat.src

See Also

gdaDStat, dstatmtControlCreate

gdaGetIndex

Purpose

Gets the index of a variable in a **GAUSS** Data Archive.

Format

ind = **gdaGetIndex**(*filename*, *varname*);

Input

filename	string, name of data file.
varname	string, name of variable in the GDA.

Output

ind	scalar, index of variable in the GDA.

Remarks

If **gdaGetIndex** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

Example

```
    ind = gdaGetIndex("myfile.gda","observed");
```

See Also

gdaGetName, gdaReadByIndex

gdaGetName

Purpose

Gets the name of a variable in a **GAUSS** Data Archive.

Format

```
varname = gdaGetName(filename, varind);
```

Input

filename	string, name of data file.
varind	scalar, index of variable in the GDA.

Output

varname	string, name of variable in the GDA.

Remarks

If **gdaGetName** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.

Example

```
varname = gdaGetName("myfile.gda",5);
```

See Also

gdaGetIndex, gdaRead, gdaGetNames

gdaGetNames

Purpose

Gets the names of all the variables in a **GAUSS** Data Archive.

Format

varnames = **gdaGetNames**(*filename*);

Input

filename	string, name of data file.

Output

varnames	Nx1 string array, names of all the variables in the GDA.

Remarks

If **gdaGetNames** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.

4	File read error.
5	Invalid file type.
10	File contains no variables.
13	Result too large for current platform.
14	File too large to be read on current platform.

Example

```
    varnames = gdaGetNames("myfile.gda");
```

See Also

gdaGetTypes, gdaGetName

gdaGetOrders

Purpose

Gets the orders of a variable in a **GAUSS** Data Archive.

Format

ord = **gdaGetOrders**(*filename*, *varname*);

Input

filename	string, name of data file.

varname	string, name of variable in the GDA.

Output

ord	Mx1 vector, orders of the variable in the GDA.

Remarks

If the specified variable is a matrix or string array, then *ord* will be a 2x1 vector containing the rows and columns of the variable respectively. If the variable is a string, then *ord* will be a scalar containing the length of the string. If the variable is an N-dimensional array, then *ord* will be an Nx1 vector containing the sizes of each dimension.

If **gdaGetOrders** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

Example

```
ord = gdaGetOrders("myfile.gda","x5");
```

See Also

gdaGetName, gdaGetIndex

gdaGetType

Purpose

Gets the type of a variable in a **GAUSS** Data Archive.

Format

vartype = **gdaGetType**(*filename*, *varname*);

Input

filename	string, name of data file.
varname	string, name of variable in the GDA.

Output

vartype	scalar, type of the variable in the GDA.

Remarks

vartype may contain any of the following:

6	Matrix

13	String
15	String array
21	Array

If **`gdaGetType`** fails, it will return a scalar error code. Call **`scalerr`** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

Example

```
        vartype = gdaGetType("myfile.gda","x1");
```

See Also

gdaGetTypes

gdaGetTypes

Purpose

Gets the types of all the variables in a **GAUSS** Data Archive.

Format

```
vartypes = gdaGetTypes(filename);
```

Input

filename	string, name of data file.

Output

vartypes	Nx1 vector, types of all the variables in the GDA.

Remarks

vartypes may contain any of the following:

6	Matrix
13	String
15	String array
21	Array

If **gdaGetTypes** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. Valid error codes for this command include:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
10	File contains no variables.
14	File too large to be read on current platform.

Example

```
        vartypes = gdaGetTypes("myfile.gda");
```

See Also

gdaGetNames, gdaRead

gdaGetVarInfo

Purpose

Gets information about all of the variables in a **GAUSS** Data Archive and returns it in an array of **gdavartable** structures.

Include

`gdafns.sdf`

Format

```
vtab = gdaGetVarInfo(filename);
```

Input

filename	string, name of data file.

Output

vtab Nx1 array of **gdavartable** structures, where N is the number of variables in *filename*, containing the following members:

vtab [i].name	string, name of variable.
vtab [i].type	scalar, type of variable.
vtab [i].orders	Mx1 vector or scalar, orders of the variable.

Remarks

The size of *vtab.orders* is dependent on the type of the variable as follows:

Variable Type	***vtab.orders***
array	Mx1 vector, where M is the number of dimensions in the array, containing the sizes of each dimension, from the

	slowest-moving dimension to the fastest-moving dimension.
matrix	2x1 vector containing the rows and columns of the matrix, respectively.
string	scalar containing the length of string, excluding the null terminating byte.
string array	2x1 vector containing the rows and columns of the string array, respectively.

`vtab.type` may contain any of the following:

6	matrix
13	string
15	string array
21	array

Example

```
    //Execute structure definition
    #include gdafns.sdf
    struct gdavartable vtab;

    vtab = gdaGetVarInfo("myfile.gda");
```

Source

gdafns.src

See Also

gdaReportVarInfo, gdaGetNames, gdaGetTypes, gdaGetOrders

gdaIsCplx

Purpose

Checks to see if a variable in a **GAUSS** Data Archive is complex.

Format

```
y = gdaIsCplx(filename, varname);
```

Input

filename	string, name of data file.
varname	string, name of variable in the GDA.

Output

y	scalar, 1 if variable is complex; 0 if real.

Remarks

If **gdaIsCplx** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. Valid error codes for this command include:

1	Null file name.

2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

Example

```
    cplx = gdaIsCplx("myfile.gda","x1");
```

gdaLoad

Purpose

Loads variables in a GDA into the workspace.

Format

ret = **gdaLoad**(*filename*, *create*, *modify*, *rename*, *ftypes*, *errh*, *report*);

Input

filename	string, name of data file.

create	scalar, create flag:	
	0	do not create any new variables in the workspace.
	1	create new variables in the workspace.
modify	scalar, modify flag:	
	0	do not modify any variables in the workspace.
	1	if the name of a variable in the data file matches the name of a variable already in the workspace, modify that variable.
rename	scalar, rename flag:	
	0	do not rename a variable retrieved from the data file when copying it into the workspace.
	1	rename variables retrieved from the data file when copying them into the workspace if there are name conflicts with existing variables, which may not be modified.
ftypes	scalar, type force flag:	
	0	do not force a type change on any variables in the workspace when modifying.
	1	force a type change on a variable in the workspace when modifying it with the data in a variable of the same name in the data file. Note that if *ftypes* is set to 1, **gdaLoad** will follow regular type change rules. The types of sparse matrix and structure variables will NOT be changed.

errh	scalar, controls the error handling of **`gdaLoad`**:	
	0	skip operations that cannot be performed, without setting an error return.
	1	return an error code if operations are skipped.
	2	terminate program if operations are skipped.
report	scalar, controls reporting:	
	0	no reporting.
	1	report only name changes and operations that could not be performed.
	2	report type changes, name changes, and operations that could not be performed.
	3	report everything.

Output

ret	scalar, return code, 0 if successful, otherwise one of the following error codes:	
	4	File read error.
	5	Invalid file type.
	10	File contains no variables.
	14	File too large to be read on current platform.
	24	Variables skipped.
	26	Cannot add structure definition.

`27`	Structure definition does not match.

Remarks

For each variable in *`filename`*, **`gdaLoad`** will first compare the name of the variable against the names of the variables already resident in the **GAUSS** workspace to see if there is a match. If there is not a match, and *`create`* is set to 1, it will create a new variable. Otherwise if *`create`* is set to 0, it will skip that variable.

If the variable name does match that of a variable already resident in the **GAUSS** workspace, and *`modify`* is set to 1, it will attempt to modify that variable. If the types of the two variables are different, and *`ftype`* is set to 1, it will force the type change if possible and modify the existing variable.

If it cannot modify the variable or *`modify`* is set to 0, it will check to see if *`rename`* is set to 1, and if so, attempt to rename the variable, appending an _ *`num`* to the variable name, beginning with *`num`* = 1 and counting upward until it finds a name with which there are no conflicts. If the variable cannot be modified and *`rename`* is set to 0, then the variable will be skipped.

The *`rename`* argument also controls the handling of structure definitions. If a structure variable is encountered in the GDA file, and no variable of the same name exists in the workspace (or the variable is renamed), **`gdaLoad`** will attempt to find a structure definition in the workspace that matches the one in the GDA. Note that in order for structure definitions to match, the structure definition names must be the same as well as the number, order, names, and types of their members.

If no matching structure definition is found, the definition in the file will be loaded into the workspace. If there is already a non-matching structure definition with the same name in the workspace and *`rename`* is set to 1, then **`gdaLoad`** will attempt to rename the structure definition, using the same method as it does for variable names.

If a structure variable is encountered in the GDA file, a structure variable of the same name already exists in the workspace, and *`modify`* is set to 1, then **`gdaLoad`** will modify the existing variable, providing that the structure definitions of the two variables match.

Example

```
    ret = gdaLoad("myfile.gda",1,1,1,1,1,3);
```

This example loads the variables in `myfile.gda` into the workspace, creating a new variable if a variable of the same name does not already exist, modifying an existing variable if a variable of the same name does already exist and the modification does not result in an impossible type change, and renaming the variable if none of the above is possible. The example returns an error code if any variables in `myfile.gda` are skipped and reports all activity.

See Also

gdaSave

gdaPack

Purpose

Packs the data in a **GAUSS** Data Archive, removing all empty bytes and truncating the file.

Format

```
    ret = gdaPack(filename);
```

Input

filename	string, name of data file.

Output

ret	scalar, return code, 0 if successful, otherwise one of the following error codes:	
	1	Null file name.
	2	File open error.
	3	File write error.
	4	File read error.
	5	Invalid data file type.
	10	File contains no variables.
	12	File truncate error.
	14	File too large to be read on current platform.

Remarks

You may want to call **gdaPack** after several calls to **gdaUpdate** to remove all of the empty bytes from a GDA.

Example

```
    ret = gdaPack("myfile.gda");
```

See Also

gdaUpdate, gdaWrite

gdaRead

Purpose

Gets a variable from a **GAUSS** Data Archive.

Format

```
y = gdaRead(filename, varname);
```

Input

filename	string, name of data file.
varname	string, name of variable in the GDA.

Output

y	matrix, array, string or string array, variable data.

Remarks

If **gdaRead** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.

2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

Example

```
    y = gdaRead("myfile.gda","x1");
```

See Also

gdaReadByIndex, gdaGetName

gdaReadByIndex

Purpose

Gets a variable from a **GAUSS** Data Archive given a variable index.

Format

y = **gdaReadByIndex**(*filename*, *varind*);

Input

filename	string, name of data file.
varind	scalar, index of variable in the GDA.

Output

y	matrix, array, string or string array, variable data.

Remarks

If **gdaReadByIndex** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.
10	File contains no variables.

Example

```
    y = gdaReadByIndex("myfile.gda",3);
```

See Also

gdaRead, gdaGetIndex

gdaReadSome

Purpose

Reads part of a variable from a **GAUSS** Data Archive.

Format

y = **`gdaReadSome`**(*`filename`*, *`varname`*, *`index`*, *`orders`*);

Input

`filename`	string, name of data file.
`varname`	string, name of variable in the GDA.
`index`	scalar or Nx1 vector, index into variable where read is to begin.
`orders`	scalar or Kx1 vector, orders of object to output.

Output

`y`	matrix, array, string or string array, variable data.

Remarks

This command reads part of the variable *`varname`* in *`filename`*, beginning at the position indicated by *`index`*. The *`orders`* argument determines the size and shape of the object outputted by **`gdaReadSome`**. The number of elements read equals the product of all of the elements in *`orders`*.

If *`index`* is a scalar, it will be interpreted as the *`index`*th element of the variable. Thus if *`varname`* references a 10x5 matrix, an *`index`* of 42 would indicate the 42nd element, which is equivalent to the [8,2] element of the matrix (remember that **GAUSS** matrices are stored in row major order). If *`index`* is an Nx1 vector, then N must equal the number of dimensions in the variable referenced by *`varname`*.

If *`orders`* is a Kx1 vector, then *`y`* will be a K-dimensional object. If *`orders`* is a scalar r, then *`y`* will be an rx1 column vector. To specify a 1xr row vector, set *`output`* = { 1, r }.

If the variable referenced by *`varname`* is numeric (a matrix or array) and *`orders`* is a scalar or 2x1 vector, then *`y`* will of type matrix. If the variable is numeric and *`orders`* is an Nx1 vector where N>2, then *`y`* will be of type array.

If *`varname`* references a string, then both *`index`* and *`orders`* must be scalars, and *`index`* must contain an index into the string in characters.

If **`gdaReadSome`** fails, it will return a scalar error code. Call **`scalerr`** to get the value of the error code. The error code may be any of the following:

`1`	Null file name.
`2`	File open error.
`4`	File read error.
`5`	Invalid file type.
`8`	Variable not found.
`10`	File contains no variables.
`13`	Result too large for current platform.
`14`	File too large to be read on current platform.
`15`	Argument out of range.
`18`	Argument wrong size.

Example

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");

index = { 35,20 };
orders = { 25,5 };
y = gdaReadSome("myfile.gda","x1",index,orders);
```

This example reads 25*5=125 elements from *x1*, beginning with the [35,20] element. The 125 elements are returned as a 25x5 matrix, *y*.

See Also

gdaWriteSome, gdaRead

gdaReadSparse

Purpose

Gets a sparse matrix from a **GAUSS** Data Archive.

Format

sm = **gdaReadSparse**(*filename*, *varname*);

Input

filename	string, name of data file.
varname	string, name of sparse matrix variable in the GDA.

Output

sm	sparse matrix.

Remarks

If **gdaReadSparse** fails, it will return a sparse scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

Example

```
sparse matrix sm1;
sm1 = gdaReadSparse("myfile.gda","sm");
```

See Also

gdaRead, gdaReadStruct, gdaWrite

gdaReadStruct

Purpose

Gets a structure from a **GAUSS** Data Archive.

Format

{ *instance*, *retcode* } = **gdaReadStruct**(*filename*, *varname*, *structure_type*);

Input

filename	string, name of data file.
varname	string, name of structure instance in the GDA.
structure_type	string, structure type.

Output

instance	instance of the structure.
retcode	scalar, 0 if successful, otherwise, any of the following error codes:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.

8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

Remarks

instance can be an array of structures.

Example

```
struct mystruct {
   matrix x;
   array a;
};

struct mystruct msw;
msw.x = rndn(500,25);
msw.a = areshape(rndn(5000,100),10|500|100);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",msw,"ms");
struct mystruct msr;
{ msr, ret } = gdaReadStruct("myfile.gda","ms","mystruc
t");
```

See Also

gdaRead, gdaReadSparse, gdaWrite

gdaReportVarInfo

Purpose

Gets information about all of the variables in a **GAUSS** Data Archive and returns it in a string array formatted for printing.

Format

```
vinfo = gdaReportVarInfo(filename);
```

Input

filename	string, name of data file.

Output

vinfo	Nx1 string array containing variable information.

Remarks

If you just want to print the information to the window, call **gdaReportVarInfo** without assigning the output to a symbol name:

```
gdaReportVarInfo(filename);
```

Example

```
x1 = rndn(100,50);
```

```
    x2 = rndn(75,5);
    a = areshape(rndn(10000,1),10|100|10);
    fname = "myfile.gda";
    ret = gdaCreate(fname,1);
    ret = gdaWrite(fname,x1,"x1");
    ret = gdaWrite(fname,x2,"x2");
    ret = gdaWrite(fname,a,"a1");
    gdaReportVarInfo(fname);
```

produces:

```
    Index Name Type cOrders
      1    x1   matrix 100x50
      2    x2   matrix 75x5
      3    a1   array 10x100x10
```

Source

gdafns.src

See Also

gdaGetVarInfo, gdaGetNames, gdaGetTypes, gdaGetOrders

gdaSave

Purpose

Writes variables in a workspace to a GDA.

Format

```
ret = gdaSave(filename, varnames, exclude, overwrite,
report);
```

Input

filename	string, name of data file.
varnames	string or NxK string array, names of variables in the workspace to include or exclude.
exclude	scalar, include/exclude flag:
	0 include all variables contained in *varnames*.
	1 exclude all variables contained in *varnames*.
overwrite	scalar, controls the overwriting of the file and variables in the file:
	0 if file exists, return with an error code.
	1 if file exists, overwrite completely.
	2 if file exists, append to file, appending to variable names if necessary to avoid name conflicts.
	3 if file exists, update file. When a name confict occurs, update the existing variable in the file with the new variable.
report	scalar, controls reporting:
	0 no reporting.

1	report only name changes (note that name changes occur only when `overwrite` is set to 2).
3	report everything.

Output

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.
3	File write error.
4	File read error.
5	Invalid file type.
6	File exists and `overwrite` set to 0.
7	Cannot create file.
14	File too large to be read on current platform.
16	Cannot write to GDA - version outdated.
17	Type mismatch.

Remarks

Only initialized variables are written to the GDA with **`gdaSave`**.

If *varnames* is a null string and *exclude* is set to 0, it will be interpreted as indicating all of the variables in the workspace.

You may add an asterisk (*) to the end of a variable name in *varnames* to indicate that all variables beginning with the specified text are to be selected. For example, setting *varnames* to the string "_*" and setting *exclude* to 1 indicates that all variables EXCEPT those starting with an underscore should be written to the GDA.

The names of the variables in the workspace are the names that are given to the variables when they are written to the GDA, with the exception of names that are changed to avoid conflicts.

If you set *overwrite* to 2, and variable name conflicts are encountered, **gdaSave** will append an underscore and a number to the name of the variable it is adding. It will first try changing the name to *name_1*. If there is a conflict with that name, it will change it to *name_2*, and so on until it finds a name that does not conflict with any of the variables already in the GDA.

Example

```
    run -r myfile.gau;
    ret = gdaSave("myfile.gda","x*",0,2,3);
```

This example runs a **GAUSS** program called `myfile.gau` and then writes all initialized variables in the workspace beginning with '*x*' to the file `myfile.gda`. If `myfile.gda` already exists, this example appends to it, changing the names of the variables that it writes to the file if necessary to avoid name conficts. All writing and variable name changing is reported.

See Also

gdaLoad

gdaUpdate

Purpose

Updates a variable in a **GAUSS** Data Archive.

Format

```
ret = gdaUpdate(filename, x,  varname);
```

Input

filename	string, name of data file.
x	matrix, array, string or string array, data.
varname	string, variable name.

Output

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.
2	File open error.
3	File write error.
4	File read error.
5	Invalid data file type.

8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

Remarks

This command updates the variable *varname* in *filename* with the data contained in *x*.

If *x* is larger than the specified variable in the file, then **gdaUpdate** writes the new variable data after the last variable in the data file, moving the variable descriptor table to make room for the data and leaving empty bytes in the place of the old variable. This does not change the index of the variable because variable indices are determined NOT by the order of the variable data in a GDA, but by the order of the variable descriptors.

If *x* is the same size or smaller than the specified variable in the file, then **gdaUpdate** writes the data in *x* over the specified variable. If *x* is smaller, then **gdaUpdate** leaves empty bytes between the end of the updated variable and the beginning of the next variable in the data file.

This command updates variables quickly by not moving data in the file unnecessarily. However, calling **gdaUpdate** several times for one file may result in a file with a large number of empty bytes. To pack the data in a GDA, so it contains no empty bytes, call **gdaPack**. Or to update a variable without leaving empty bytes in the file, call **gdaUpdateAndPack**.

Example

```
    x = rndn(100,50);
    ret = gdaCreate("myfile.gda",1);
    ret = gdaWrite("myfile.gda",x,"x1");
```

```
y = rndn(75,5);
ret = gdaUpdate("myfile.gda",y,"x1");
```

See Also

gdaUpdateAndPack, gdaPack, gdaWrite

gdaUpdateAndPack

Purpose

Updates a variable in a **GAUSS** Data Archive, leaving no empty bytes if the updated variable is smaller or larger than the variable it is replacing.

Format

ret = **gdaUpdateAndPack**(*filename*, *x*, *varname*);

Input

filename	string, name of data file.
x	matrix, array, string or string array, data.
varname	string, variable name.

Output

ret	scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.
2	File open error.
3	File write error.
4	File read error.
5	Invalid data file type.
8	Variable not found.
10	File contains no variables.
12	File truncate error.
14	File too large to be read on current platform.

Remarks

This command updates the variable *varname* in *filename* with the data contained in *x*. **gdaUpdateAndPack** always writes the data in *x* over the specified variable in the file. If *x* is larger than the specified variable, then it first moves all subsequent data in the file to make room for the new data. If *x* is smaller, then **gdaUpdateAndPack** writes the data, packs all of the subsequent data, leaving no empty bytes after the updated variable, and truncates the file.

This command uses disk space efficiently; however, it may be slow for large files (especially if the variable to be updated is one of the first variables in the file). If speed is a concern, you may want to use **gdaUpdate** instead.

Example

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
```

```
    ret = gdaWrite("myfile.gda",x,"x1");

    y = rndn(75,5);
    ret = gdaUpdateAndPack("myfile.gda",y,"x1");
```

See Also

gdaUpdate, gdaWrite

gdaVars

Purpose

Gets the number of variables in a **GAUSS** Data Archive.

Format

nvars = **gdaVars**(*filename*);

Input

filename	string, name of data file.

Output

nvars	scalar, the number of variables in *filename*.

Example

```
    nvars = gdaVars("myfile.gda");
```

Source

`gdafns.src`

See Also

gdaReportVarInfo, gdaGetNames

gdaWrite

Purpose

Writes a variable to a **GAUSS** Data Archive.

Format

ret = **gdaWrite**(*filename*, *x*, *varname*);

Input

filename	string, name of data file.
x	matrix, array, string or string array, data to write to the GDA.
varname	string, variable name.

Output

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.

2	File open error.
3	File write error.
4	File read error.
5	Invalid data file type.
9	Variable name too long.
11	Variable name must be unique.
14	File too large to be read on current platform.

Remarks

gdaWrite adds the data in *x* to the end of the variable data in *filename*, and gives the variable the name contained in *varname*.

Example

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");
```

See Also

gdaWrite32, gdaCreate

gdaWrite32

Purpose

Writes a variable to a **GAUSS** Data Archive using 32-bit system file write commands.

Format

```
ret = gdaWrite32(filename, x,  varname);
```

Input

filename	string, name of data file.
x	matrix, array, string or string array, data to write to the GDA.
varname	string, variable name.

Output

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.
2	File open error.
3	File write error.
4	File read error.
5	Invalid data file type.

9	Variable name too long.
11	Variable name must be unique.
14	File too large to be read on current platform.
25	Not supported for use with a file created on a machine with a different byte order.

Remarks

gdaWrite32 adds the data in *x* to the end of the variable data in *filename*, and gives the variable the name contained in *varname*.

This command is a speed optimization command for Windows. On all other platforms, this function is identical to **gdaWrite**. **gdaWrite** uses system file write commands that support 64-bit file sizes. These commands are slower on Windows XP than the 32-bit file write commands that were used for binary writes in **GAUSS** 6.0 and earlier. **gdaWrite32** uses the 32-bit Windows system write commands, which will be faster on Windows XP. Note, however, that **gdaWrite32** does not support 64-bit file sizes.

This command does not support writing to a GDA that was created on a platform with a different byte order than the current machine. **gdaWrite** supports full cross-platform writing to GDA's.

Example

```
    x = rndn(100,50);
    ret = gdaCreate("myfile.gda",1);
    ret = gdaWrite32("myfile.gda",x,"x1");
```

See Also

gdaWrite, gdaCreate

gdaWriteSome

Purpose

Overwrites part of a variable in a **GAUSS** Data Archive.

Format

```
ret = gdaWriteSome(filename, x,  varname,  index);
```

Input

filename	string, name of data file.
x	matrix, array, string or string array, data.
varname	string, variable name.
index	scalar or Nx1 vector, index into variable where new data is to be written.

Output

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.
2	File open error.
3	File write error.
4	File read error.

`5`	Invalid data file type.
`8`	Variable not found.
`10`	File contains no variables.
`14`	File too large to be read on current platform.
`15`	Argument out of range.
`17`	Type mismatch.
`18`	Argument wrong size.
`19`	Data must be real.
`20`	Data must be complex.

Remarks

This command overwrites part of the variable *`varname`* in *`filename`* with the data contained in *x*. The new data is written to *`varname`* beginning at the position indicated by *`index`*.

If *`index`* is a scalar, it will be interpreted as the *`index`*th element of the variable. Thus if *`varname`* references a 10x5 matrix, an *`index`* of 42 would indicate the 42nd element, which is equivalent to the [8,2] element of the matrix (remember that **GAUSS** matrices are stored in row major order). If *`index`* is an Nx1 vector, then N must equal the number of dimensions in the variable referenced by *`varname`*.

If *`varname`* references a string, then *`index`* must be a scalar containing an index into the string in characters.

`gdaWriteSome` may not be used to extend the size of a variable in a GDA. If there are more elements (or characters for strings) in *x* than there are from the indexed position of the specified variable to the end of that variable, then **gdaWriteSome** will fail. Call **gdaAppend** to append data to an existing variable.

The shape of *x* need not match the shape of the variable referenced by *varname*. If *varnum* references an NxK matrix, then *x* may be any LxM matrix (or P-dimensional array) that satisfies the size limitations described above. If *x* contains R elements, then the elements in *x* will simply replace the indexed element of the specified variable and the subsequent R-1 elements (as they are laid out in memory).

If *varname* references a string array, then the size of the overall variable will change if the sum of the length of the string array elements in *x* is different than the sum of the length of the elements that they are replacing.

In this case, if the variable increases in size, then the variable data will be rewritten after the last variable in the data file, moving the variable descriptor table to make room for the data and leaving empty bytes in its old location. This does not change the index of the variable because variable indices are determined NOTby the order of the variable data in a GDA, but by the order of the variable descriptors. If the variable decreases in size, then **gdaWriteSome** leaves empty bytes between the end of the variable and the beginning of the next variable in the data file. Call **gdaPack** to pack the data in a GDA, so it contains no empty bytes.

Example

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");

y = rndn(75,5);
index = { 52, 4 };
ret = gdaWriteSome("myfile.gda",y,"x1",index);
```

This example replaces 75*5=375 elements in *x1*, beginning with the [52,4] element, with the elements in *y*.

See Also

gdaReadSome, gdaUpdate, gdaWrite

getarray

Purpose

Gets a contiguous subarray from an N-dimensional array.

Format

`y = getarray(a, loc);`

Input

a	N-dimensional array.
loc	Mx1 vector of indices into the array to locate the subarray of interest, where 1 <= M <= N.

Output

y	[N-M]-dimensional subarray or scalar.

Remarks

If N-M>0, **getarray** will return an array of [N-M] dimensions, otherwise, if N-M=0, it will return a scalar.

Example

```
    a = seqa(1,1,720);
    a = areshape(a,2|3|4|5|6);
    loc = { 2,1 };
    y = getarray(a,loc);
```

y will be a 4x5x6 array of sequential values, beginning at [1,1,1] with 361, and ending at [4,5,6] with 480.

See Also

getmatrix

getdims

Purpose

Gets the number of dimensions in an array.

Format

y = **getdims**(*a*);

Input

a	N-dimensional array.

Output

y	scalar, the number of dimensions in the array.

Example

```
    a = arrayinit(3|4|5|6|7|2,0);
    dims = getdims(a);
```

The code above, assigns *dims* to be equal to 6.

See Also

getorders

getf

Purpose

Loads an ASCII or binary file into a string.

Format

y = **getf**(*filename*, *mode*);

Input

filename	string, any valid file name.
mode	scalar 1 or 0 which determines if the file is to be loaded in ASCII mode (0) or binary mode (1).

Output

y	string containing the file.

Remarks

If the file is loaded in ASCII mode, it will be tested to see if it contains any end of file characters. These are ^Z (ASCII 26). The file will be truncated before the first ^Z, and there will be no ^Z's in the string. This is the correct way to load most text files because the ^Z's can cause problems when trying to print the string to a printer.

If the file is loaded in binary mode, it will be loaded just like it is with no changes.

Example

Suppose you have a file which writes the results of its calculations to a file in a report format. For this example, we will use the code snippet below:

```
x1 = rndn(100,5);
y1 = rndu(100,1);

output file = regression_results.txt reset;
call ols("", y1, x1);
output off;

x2 = rndn(100,5);
y2 = rndu(100,1);

output file = ols_results.txt reset;
call ols("", y2, x2);
output off;
```

Running the code above will create a file named "regression_results.txt" and a file named "ols_results.txt" in your current working directory. You can retrieve the output from either of these files with the **getf** command.

```
str = getf("regression_results.txt",1);
print str;
```

You can take this further and create a procedure that will load a list of output files for you. It can then print the output from each file as you are ready to read it.

```
    declare string array fileList = { "regression_results.tx
    t", "ols_results.txt" };

    showOutput(fileList);

    proc (0) = showOutput(fileList);
       local k;
       for i(1, rows(fileList), 1);
          print "Press any key to view the next file:";
          //wait for user input and assign the first key stroke
          //to 'k'
          k = keyw;
          print getf(fileList[i],1);
       endfor;
    endp;
```

See Also

load, save, let, con

getmatrix

Purpose

Gets a contiguous matrix from an N-dimensional array.

Format

y = **getmatrix**(*a*, *loc*);

Input

a	N-dimensional array.
loc	Mx1 vector of indices into the array to locate the matrix of interest, where M equals N, N-1 or N-2.

Output

y	KxL or 1xL matrix or scalar, where L is the size of the fastest moving dimension of the array and K is the size of the second fastest moving dimension.

Remarks

Inputting an Nx1 locator vector will return a scalar, an (N-1)x1 locator vector will return a 1xL matrix, and an (N-2)x1 locator vector will return a KxL matrix.

Example

```
//Create the sequence 1, 2, 3...20
a = seqa(1, 1, 20);

//Reshape the column vector 'a' into a 3x3x2 dimensional
//array
a = areshape(a, 3|3|2);

//Extract the second 3x2 array
mat = getmatrix(a, 2);
```

After code above *a* is equal to:

```
    Plane [1,.,.]

           1.0000000        2.0000000
           3.0000000        4.0000000
           5.0000000        6.0000000

    Plane [2,.,.]

           7.0000000        8.0000000
           9.0000000        10.000000
           11.000000        12.000000

    Plane [3,.,.]

           13.000000        14.000000
           15.000000        16.000000
           17.000000        18.000000
```

and `mat` is equal to:

```
           7.0000000        8.0000000
           9.0000000        10.000000
           11.000000        12.000000
```

See Also

getarray, getmatrix4D

getmatrix4D

Purpose

Gets a contiguous matrix from a 4-dimensional array.

Format

```
y = getmatrix4D(a, i1, i2);
```

Input

a	4-dimensional array.
i1	scalar, index into the slowest moving dimension of the array.
i2	scalar, index into the second slowest moving dimension of the array.

Output

y	KxL matrix, where L is the size of the fastest moving dimension of the array and K is the size of the second fastest moving dimension.

Remarks

getmatrix4D returns the contiguous matrix that begins at the [*i1*,*i2*,1,1] position in array *a* and ends at the [*i1*, *i2*,K,L] position.

A call to **getmatrix4D** is faster than using the more general **getmatrix** function to get a matrix from a 4-dimensional array, especially when *i1* and *i2* are the counters from nested `for` loops.

Example

```
    //Create a column vector 1, 2, 3...120
```

```
a = seqa(1,1,120);

//Reshape the column vector into a 2x3x4x5 dimensional
//array
a = areshape(a,2|3|4|5);

//Extract a submatrix
y = getmatrix4D(a,2,3);
```

After the code above:

```
    101   102   103   104   105
y = 106   107   108   109   110
    111   112   113   114   115
    116   117   118   119   120
```

See Also

getmatrix, getscalar4D, getarray

getname

Purpose

Returns a column vector containing the names of the variables in a **GAUSS** data set.

Format

y = **getname**(*dset*);

Input

dset	string specifying the name of the data set from which the function will obtain the variable names.

Output

y	Nx1 vector containing the names of all of the variables in the specified data set.

Remarks

The output, *y*, will have as many rows as there are variables in the data set.

Example

```
    y = getname("olsdat");
    format 8,8;
    print $y;
```

produces:

```
     TIME
     DIST
     TEMP
     FRICT
```

The above example assumes that the data set `olsdat` contains the variables: *TIME*, *DIST*, *TEMP*, *FRICT*.

Note that the extension is not included in the filename passed to the **getname** function.

See Also

getnamef, indcv

getnamef

Purpose

Returns a string array containing the names of the variables in a **GAUSS** data set.

Format

y = **`getnamef`**(*f*);

Input

f	scalar, file handle of an open data set

Output

y	Nx1 string array containing the names of all of the variables in the specified data set.

Remarks

The output, *y*, will have as many rows as there are variables in the data set.

Example

```
//Open the dataset
open f = olsdat for read;

//Create a string array with the variable names from the
//dataset
y = getnamef(f);

//Check which variables are character and which are numeric
t = vartypef(f);

print y;
```

produces:

```
 time
 dist
 temp
 frict
```

The above example assumes that the data set `olsdat` contains the variables: *`TIME`*, *`DIST`*, *`TEMP`*, *`FRICT`*.

Note the use of **`vartypef`** to determine the types of these variables.

See Also

getname, indcv, vartypef

getNextTradingDay

Purpose

Returns the next trading day.

Format

```
n = getNextTradingDay(a);
```

Input

a	scalar, date in DT scalar format.

Output

n	scalar, next trading day in DT scalar format.

Remarks

A trading day is a weekday that is not a holiday as defined by the New York Stock Exchange from 1888 through 2006. Holidays are defined in `holidays.asc`. You may edit that file to modify or add holidays.

Source

`finutils.src`

Globals

`_fin_holidays`

See Also

getPreviousTradingDay, annualTradingDays

getNextWeekDay

Purpose

Returns the next day that is not on a weekend.

Format

n = **`getNextWeekDay`**(*a*);

Input

a	scalar, date in DT scalar format.

Output

n	scalar, next week day in DT scalar format.

Source

`finutils.src`

See Also

getPreviousWeekDay

getnr

Purpose

Computes number of rows to read per iteration for a program that reads data from a disk file in a loop.

Format

```
nr = getnr(nsets, ncols);
```

Input

nsets	scalar, estimate of the maximum number of duplicate copies of the data matrix read by **readr** to be kept in memory during each iteration of the loop.
ncols	scalar, columns in the data file.

Output

nr	scalar, number of rows **readr** should read per iteration of the read loop.

Remarks

If __*row* is greater than 0, *nr* will be set to __*row*.

If an insufficient memory error is encountered, change __*rowfac* to a number less than 1.0 (e.g., 0.75). The number of rows read will be reduced in size by this factor.

Source

`gauss.src`

Globals

__row, *__rowfac*, *__maxvec*

getnrmt

Purpose

Computes number of rows to read per iteration for a program that reads data from a disk file in a loop.

Format

```
nr = getnr(nsets, ncols, row, rowfac, maxv);
```

Input

nsets	scalar, estimate of the maximum number of duplicate copies of the data matrix read by **readr** to be kept in memory during each iteration of the loop.
ncols	scalar, columns in the data file.
row	scalar, if row is greater than 0, *nr* will be set to *row*.
rowfac	scalar, *nr* will be reduced in size by this factor. If insufficient memory error is encounted, change this to a number less than one (e.g., 0.9).

maxv	scalar, the largest number of elements allowed in any one matrix.

Output

nr	scalar, number of rows **readr** should read per iteration of the read loop.

Source

`gaussmt.src`

getorders

Purpose

Gets the vector of orders corresponding to an array.

Format

y = **getorders**(*a*);

Input

a	N-dimensional array.

Output

y	Nx1 vector of orders, the sizes of the dimensions of the

array.

Example

```
//Allocate a 7x6x5x4x3 dimensional array
a = arrayalloc(7|6|5|4|3,0);
orders = getorders(a);
```

After the code above:

```
            7
            6
orders =    5
            4
            3
```

See Also

getdims

getpath

Purpose

Returns an expanded filename including the drive and path.

Format

fname = **getpath**(*pfname*);

Input

pfname	string, partial filename with only partial or missing path information.

Output

fname	string, filename with full drive and path.

Remarks

This function handles relative path references.

Example

```
    y = getpath("temp.e");
    print y;
```

produces:

```
    C:\gauss\temp.e
```

assuming that `C:\gauss` is the current directory.

Source

`getpath.src`

getPreviousTradingDay

Purpose

Returns the previous trading day.

Format

```
n = getPreviousTradingDay(a);
```

Input

a	scalar, date in DT scalar format.

Output

n	scalar, previous trading day in DT scalar format.

Remarks

A trading day is a weekday that is not a holiday as defined by the New York Stock Exchange from 1888 through 2006. Holidays are defined in `holidays.asc`. You may edit that file to modify or add holidays.

Source

`finutils.src`

Globals

_fin_holidays

See Also

getNextTradingDay

getPreviousWeekDay

Purpose

Returns the previous day that is not on a weekend.

Format

n = **getPreviousWeekDay**(*a*);

Input

a	scalar, date in DT scalar format.

Output

n	scalar, previous week day in DT scalar format.

Source

`finutils.src`

See Also

getNextWeekDay

getRow

Purpose

Returns a specified row from a matrix.

Format

y = **getRow**(*a*, *row*);

Input

a	NxK matrix
row	The row of the matrix to extract.

Output

y	A 1xK row vector.

Remarks

getRow is designed to give an alternative access to rows in a matrix than indexing the matrix by brackets.

Example

First create a matrix, *a*:

```
    a = rndn(10,10);
```

Now you can assign a variable *y* to be equal the third row of *a* with either of the following statements.

```
    y = getRow(a,3);
```

or

```
    y = a[3,.];
```

While both statements will produce the same result, the first may make for code that is easier to read and interpret.

See Also

CR-getTrRow

getscalar3D

Purpose

Gets a scalar from a 3-dimensional array.

Format

```
y = getscalar3D(a, i1, i2, i3);
```

Input

a	3-dimensional array.
i1	scalar, index into the slowest moving dimension of the array.
i2	scalar, index into the second slowest moving dimension of

	the array.
i3	scalar, index into the fastest moving dimension of the array.

Output

y	scalar, the element of the array indicated by the indices.

Remarks

getscalar3D returns the scalar that is located in the [*i1*, *i2*, *i3*] position of array *a*.

A call to **getscalar3D** is faster than using the more general **getmatrix** function to get a scalar from a 3-dimensional array.

Example

```
//Create a column vector 1, 2, 3,...24
a = seqa(1,1,24);

//Reshape the column vector into a 2x3x4 dimensional array
a = areshape(a,2|3|4);

y = getscalar3D(a,1,3,2);
```

A 2x3x4 dimensional array can be thought of as two 3x4 dimensional matrices. The call to **getScalar3D** above, returns the [3,2] element of the first of these matrices. The value of which is:

```
y = 10
```

See Also

getmatrix, getscalar4D, getarray

getscalar4D

Purpose

Gets a scalar from a 4-dimensional array.

Format

```
y = getscalar4D(a, i1, i2, i3, i4);
```

Input

a	4-dimensional array.
i1	scalar, index into the slowest moving dimension of the array.
i2	scalar, index into the second slowest moving dimension of the array.
i3	scalar, index into the second fastest moving dimension of the array.
i4	scalar, index into the fastest moving dimension of the array.

Output

y	scalar, the element of the array indicated by the indices.

Remarks

getscalar4D returns the scalar that is located in the [*i1*, *i2*, *i3*, *i4*] position of array *a*.

A call to **getscalar4D** is faster than using the more general **getmatrix** function to get a scalar from a 4-dimensional array.

Example

```
a = seqa(1,1,120);
a = areshape(a,2|3|4|5);
y = getscalar4D(a,1,3,2,5);
```

The code above assigns *y* equal to 50.

See Also

getmatrix, getscalar3D, getarray

getTrRow

Purpose

Transposes a matrix and then returns a single row from it.

Format

y = **getTrRow**(*a*, *row*);

Input

a	NxK matrix
row	The row of the matrix to extract.

Output

y	A 1xK row vector.

Remarks

getRow is designed to give an alternative access to rows in a matrix than indexing the matrix by brackets.

Example

```
    a = rndn(10,10);
    y = getTrRow(a,3);
```

See Also

getRow

getwind

Purpose

Retrieve the current graphic panel number. Note: This function is for use with the deprecated PQG graphics.

Library

`pgraph`

Format

n = **`getwind`**;

Output

n	scalar, graphic panel number of current graphic panel.

Remarks

The current graphic panel is the graphic panel in which the next graph will be drawn.

Source

`pwindow.src`

See Also

endwind, begwind, window, setwind, nextwind

gosub

Purpose

Causes a branch to a subroutine. Note: This is an advanced function that gives extra flexibility for sophisticated users in some circumstances. In most cases, it is prefereable to create a procedure (`proc`).

Format

```
gosub label;
.
.
.
label:
.
.
.
return;
```

Remarks

For multi-line recursive user-defined functions, see **Procedures and Keywords**, Chapter 1.

When a `gosub` statement is encountered, the program will branch to the label and begin executing from there. When a `return` statement is encountered, the program will resume executing at the statement following the `gosub` statement. Labels are 1-32 characters long and are followed by a colon. The characters can be A-Z or 0-9, but they must begin with an alphabetic character. Uppercase or lowercase is allowed.

It is possible to pass parameters to subroutines and receive parameters from them when they return. See the second example, following.

The only legal way to enter a subroutine is with a `gosub` statement.

If your subroutines are at the end of your program, you should have an `end` statement before the first one to prevent the program from running into a subroutine without using a `gosub`. This will result in a Return without gosub error message.

The variables used in subroutines are not local to the subroutine and can be accessed from other places in your program. (See **Procedures and Keywords**, Chapter 1.)

Example

In the program below the name *mysub* is a label. When the `gosub` statement is executed, the program will jump to the label *mysub* and continue executing from there. When the `return` statement is executed, the program will resume executing at the statement following the `gosub`.

```
x = rndn(3,3);
z = 0;
gosub mysub;
print z;
end;

/* ------ Subroutines Follow ------ */

mysub:
 z = inv(x);
return;
```

Parameters can be passed to subroutines in the following way (line numbers are added for clarity):

```
1. gosub mysub(x,y);
2. pop j; /* b will be in j */
3. pop k; /* a will be in k */
4. t = j*k;
5. print t;
6. end;
7.
8. /* ---- Subroutines Follow ----- */
9.
10. mysub:
11. pop b; /* y will be in b */
12. pop a; /* x will be in a */
13.
```

```
    14. a = inv(b)*b+a;
    15. b = a'b;
    16. return(a,b);
```

In the above example, when the `gosub` statement is executed, the following sequence of events results (line numbers are included for clarity):

1. *x* and *y* are pushed on the stack and the program branches to the label *mysub* in line 10.

11. the second argument that was pushed, *y*, is `pop`'ped into *b*.

12. the first argument that was pushed, *x*, is `pop`'ped into *a*.

14. **inv**(*b*)**b*+*a* is assigned to *a*.

15. *a*'*b* is assigned to *b*.

16. *a* and *b* are pushed on the stack and the program branches to the statement following the `gosub`, which is line 2.

2. the second argument that was pushed, *b*, is `pop`'ped into *j*.

3. the first argument that was pushed, *a*, is `pop`'ped into *k*.

4. *j***k* is assigned to *t*.

5. *t* is printed.

6. the program is terminated with the `end` statement.

Matrices are pushed on a last-in/first-out stack in the `gosub`() and `return`() statements. They must be `pop`'ped off in the reverse order. No intervening statements are allowed between the label and the `pop` or the `gosub` and the `pop`. Only one matrix may be `pop`'ped per `pop` statement.

See Also

goto, proc, pop, return

goto

Purpose

Causes a branch to a label.

Format

```
gotolabel;
.
.
.
label:
```

Remarks

Label names can be any legal **GAUSS** names up to 32 alphanumeric characters, beginning with an alphabetic character or an underscore, not a reserved word.

Labels are always followed immediately by a colon.

Labels do not have to be declared before they are used. **GAUSS** knows they are labels by the fact that they are followed immediately by a colon.

When **GAUSS** encounters a `goto` statement, it jumps to the specified label and continues execution of the program from there.

Parameters can be passed in a `goto` statement the same way as they can with a `gosub`.

Example

```
    x = seqa(.1,.1,5);
    n = { 1 2 3 };
    goto  fip;
    print x;
    end;

    fip:
    print n;
```

produces:

```
    1.0000000 2.0000000 3.0000000
```

See Also

gosub, if

gradMT

Purpose

Computes numerical gradient.

Include

optim.sdf

Format

g = **gradMT**(&*fct*, *par1*, *data1*);

Input

&fct	scalar, pointer to procedure returning either Nx1 vector or 1x1 scalar.
par1	an instance of structure of type **PV** containing parameter vector at which gradient is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.

Output

g	NxK Jacobian or 1xK gradient.

Remarks

par1 must be created using the **pvPack** procedures.

Example

```
#include optim.sdf
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc fct(struct PV p0, struct DS d0);
   local p,y;
   p = pvUnpack(p0, "P");
```

```
        y = p[1] * exp(-p[2] * d0.dataMatrix);
        retp(y);
    endp;

    g = gradMT(&fct,p1,d0);
```

Source

`gradmt.src`

gradMTm

Purpose

Computes numerical gradient with mask.

Include

`optim.sdf`

Format

g = **gradMTm**(*&fct*, *par1*, *data1*, *mask*);

Input

&fct	scalar, pointer to procedure returning either Nx1 vector or 1x1 scalar.
par1	an instance of structure of type **PV** containing parameter vector at which gradient is to be evaluated.

data1	structure of type **DS** containing any data needed by *fct*.
mask	Kx1 matrix, elements in *g* corresponding to elements of *mask* set to zero are not computed, otherwise they are computed.

Output

g	NxK Jacobian or 1xK gradient.

Remarks

par1 must be created using the **pvPack** procedures.

Example

```
#include optim.sdf
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc fct(struct PV p0, struct DS d0);
   local p,y;
   p = pvUnpack(p0, "P");
   y = p[1] * exp(-p[2] * d0.dataMatrix);
   retp(y);
endp;
```

```
mask = { 0, 1 };
g = gradMTm(&fct,p1,d0,mask);
```

Source

`gradmt.src`

gradMTT

Purpose

Computes numerical gradient using available threads.

Include

`optim.sdf`

Format

g = **gradMTT**(*&fct*,*par1*,*data1*);

Input

fct	scalar, pointer to procedure returning either Nx1 vector or 1x1 scalar.
par1	structure of type **PV** containing parameter vector at which gradient is to be evaluated
data1	structure of type **DS** containing any data needed by *fct*

Output

g	NxK Jacobian or 1xK gradient

Remarks

par1 must be created using the **pvPack** procedures

Example

```
#include optim.sdf
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1, 0.1|0.2, "P");

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc fct(struct PV p0, struct DS d0);
   local p,y;
   p = pvUnpack(p0, "P");
   y = p[1] * exp(-p[2] * d0.dataMatrix);
   retp(y);
endp;

g = gradMT(&fct,p1,d0);
```

Source

gradmtt.src

gradMTTm

Purpose

Computes numerical gradient with mask using threads.

Include

```
sqpsolvemt.sdf
```

Format

g = **gradMTTm**(*&fct*, *par1*, *data1*, *mask*);

Input

&fct	scalar, pointer to procedure returning either Nx1 vector or 1x1 scalar
par1	structure of type **PV** containing parameter vector at which gradient is to be evaluated
data1	structure of type **DS** containing any data needed by *fct*
mask	Kx1 matrix, elements in *g* corresponding to elements of mask set to zero are not computed otherwise are computed.

Output

g	NxK Jacobian or 1xK gradient

Remarks

par1 must be created using the **pvPack** procedures

Example

```
#include sqpsolvemt.sdf
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc fct(struct PV p0, struct DS d0);
   local p,y;
   p = pvUnpack(p0,"P");
   y = p[1] * exp(-p[2] * d0.dataMatrix);
  retp(y);
endp;

mask = { 0, 1 };
g = gradMTTm(&fct,p1,d0,mask);
```

Source

gradmtt.src

gradp, gradcplx

Purpose

Computes the gradient vector or matrix (Jacobian) of a vector-valued function that has been defined in a procedure. Single-sided (forward difference) gradients are computed. **gradcplx** allows for complex arguments.

Format

g = **gradp**(*&f*, *x0*);
g = **gradcplx**(*&f*, *x0*);

Input

&f	a pointer to a vector-valued function (*f*: Kx1 → Nx1) defined as a procedure. It is acceptable for *f*(*x*) to have been defined in terms of global arguments in addition to *x*, and thus *f* can return an Nx1 vector:
x0	Kx1 vector of points at which to compute gradient.

```
proc f(x);
   retp( exp(x.*b));
endp;
```

Output

g	NxK matrix containing the gradients of *f* with respect to the variable *x* at *x0*.

Remarks

gradp will return a row for every row that is returned by *f*. For instance, if *f* returns a scalar result, then **gradp** will return a 1xK row vector. This allows the same function to be used regardless of N, where N is the number of rows in the result returned by *f*. Thus, for instance, **gradp** can be used to compute the Jacobian matrix of a set of equations.

Example

```
proc myfunc(x);
   retp(x .* 2 .* exp( x .* x ./ 3 ));
endp;

x0 = 2.5|3.0|3.5;
y = gradp(&myfunc,x0);
```

After the code above, *y* is equal to:

```
 82.989017        0.00000000        0.00000000
0.00000000         281.19753        0.00000000
0.00000000        0.00000000         1087.9541
```

It is a 3x3 matrix because we are passing it 3 arguments and **myfunc** returns 3 results when we do that; the off-diagonals are zeros because the cross-derivatives of 3 arguments are 0.

Source

gradp.src

See Also

hessp, hesscplx

graphprt

Purpose

Controls automatic printer hardcopy and conversion file output. Note: This function is for use with the deprecated PQG graphics. Use the **`plotSave`** function instead.

Library

`pgraph`

Format

`graphprt`(*`str`*);

Input

`str`	string, control string.

Portability

UNIX

Not supported.

Remarks

`graphprt` is used to create hardcopy output automatically without user intervention. The input string *`str`* can have any of the following items, separated by spaces. If *`str`* is a null string, the interactive mode is entered. This is the default.

-p	print graph.
-po=c	set print orientation: *l* landscape. *p* portrait.
-c=n	convert to another file format: *1* Encapsulated PostScript file. *3* HPGL Plotter file. *5* BMP (Windows Bitmap). *8* WMF (Windows Enhanced Metafile).
-cf=name	set converted output file name.
-i	minimize (iconize) the graphics window.
-q	close window after processing.
-w=n	display graph, wait *n* seconds, then continue.

If you are not using graphic panels, you can call **graphprt** anytime before the call to the graphics routine. If you are using graphic panels, call **graphprt** just before the **endwind** statement.

The print option default values are obtained from the viewer application. Any parameters passed through **graphprt** will override the default values. See **Publication Quality Graphics**, Chapter 1.

Example

Automatic print using a single graphics call:

```
library pgraph;
graphset;

load x,y;

graphprt("-p"); /* tell "xy" to print */
xy(x,y);         /* create graph and print */
```

Automatic print using multiple graphic panels. Note **graphprt** is called once just before the **endwind** call:

```
library pgraph;
graphset;

load x,y;

begwind;
window(1,2,0);    /* create two windows */
setwind(1);
xy(x,y);         /* first graphics call */
nextwind;
xy(x,y);         /* second graphics call */
graphprt("-p");
endwind;         /* print page containing all graphs */
```

The next example shows how to build a string to be used with **graphprt**:

```
library pgraph;
graphset;
load x,y;

cvtnam = "mycvt.eps"; /* name of output file */
/* concatenate options into one string */
cmdstr = "-c=1" $+ " -cf=" $+ cvtnam;
cmdstr = cmdstr $+ " -q";
```

```
    graphprt(cmdstr); /* tell "xy" to convert and */
    /* close */
    xy(x,y); /* create graph and convert */
```

The above string *cmdstr* will read as follows:

```
    "-c=1 -cf=mycvt.eps -q"
```

Source

pgraph.src

graphset

Purpose

Reset graphics global variables to default values. Note: This function is for use with the deprecated PQG graphics.

Library

pgraph

Format

```
graphset;
```

Remarks

This procedure is used to reset the defaults between graphs.

graphset may be called between each graphic panel to be displayed.

To change the default values of the global control variables, make the appropriate changes in the file `pgraph.dec` and to the procedure **graphset**.

Source

`pgraph.src`

h

hasimag

Purpose

Tests whether the imaginary part of a complex matrix is negligible.

Format

y = **`hasimag`**(*x*);

Input

x	NxK matrix.

Output

y	scalar, 1 if the imaginary part of *x* has any nonzero elements, 0 if it consists entirely of 0's.

Remarks

The function **`iscplx`** tests whether *x* is a complex matrix or not, but it does not test the contents of the imaginary part of *x*. **`hasimag`** tests the contents of the imaginary part of *x* to see if it is zero.

hasimag actually tests the imaginary part of *x* against a tolerance to determine if it is negligible. The tolerance used is the imaginary tolerance set with the **sysstate** command, case 21.

Some functions are not defined for complex matrices. **iscplx** can be used to determine whether a matrix has no imaginary part and so can pass through those functions. **hasimag** can be used to determine whether a complex matrix has a negligible imaginary part and could thus be converted to a real matrix to pass through those functions.

iscplx is useful as a preliminary check because for large matrices it is much faster than **hasimag**.

Example

```
x = { 1   2 3i,
      4-i 5 6i,
      7   8i 9 };

if hasimag(x);
    //code path for complex case
else;
    //code path for real case
endif;
```

See Also

iscplx

header

Purpose

Prints a header for a report.

Format

```
header(prcnm, dataset, ver);
```

Input

prcnm	string, name of procedure that calls **header**.
dataset	string, name of data set.
ver	2x1 numeric vector, the first element is the major version number of the program, the second element is the revision number. Normally this argument will be the version/revision global *(__??_ver)* associated with the module within which header is called. This argument will be ignored if set to 0.

Global Input

__header string, containing one or more of the following letters:

t	title is to be printed
l	lines are to bracket the title

	d	a date and time is to be printed
	v	version number of program is to be printed
	f	file name being analyzed is to be printed
__title	string, title for header.	

Source

gauss.src

headermt

Purpose

Prints a header for a report.

Format

headermt(*prcnm*, *dataset*, *ver*, *header*, *title*);

Input

prcnm	string, name of procedure that calls **header**.
dataset	string, name of data set.
ver	2x1 numeric vector, the first element is the major version number of the program, the second element is the revision number. Normally this argument will be the version/revision

	global `(__??_ver)` associated with the module within which header is called. This argument will be ignored if set to 0.
header	string, containing one or more of the following letters:
	`t` title is to be printed
	`l` lines are to bracket the title
	`d` a date and time is to be printed
	`v` version number of program is to be printed
	`f` file name being analyzed is to be printed
title	string, title for header.

Source

gaussmt.src

hess

Purpose

Computes the Hessenberg form of a square matrix.

Format

{ *h*, *z* } = **hess**(*x*);

Input

x	KxK matrix.

Output

h	KxK matrix, Hessenberg form.
z	KxK matrix, transformation matrix.

Remarks

hess computes the Hessenberg form of a square matrix. The Hessenberg form is an intermediate step in computing eigenvalues. It also is useful for solving certain matrix equations that occur in control theory (see Van Loan, Charles F. "Using the Hessenberg Decomposition in Control Theory". *Algorithms and Theory in Filtering and Control*. Sorenson, D.C. and R.J. Wets, eds., Mathematical Programming Study No. 18, North Holland, Amsterdam, 1982, 102-111).

z is an orthogonal matrix that transforms *x* into *h* and vice versa. Thus:

```
     h = z'*x*z
```

and since *z* is orthogonal,

```
     x = z*h*z'
```

x is reduced to upper Hessenberg form using orthogonal similiarity transformations. This preserves the Frobenious norm of the matrix and the condition numbers of the eigenvalues.

hess uses the ORTRAN and ORTHES functions from EISPACK.

Example

```
let x[3,3] = 1 2 3
             4 5 6
             7 8 9;

{ h, z } = hess(x);
```

See Also

schur

hessMT

Purpose

Computes numerical Hessian.

Include

`optim.sdf`

Format

h = **hessMT**(*&fct*, *par1*, *data1*);

Input

&fct	scalar, pointer to procedure returning either Nx1 vector or 1x1 scalar.

par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.

Output

h	KxK matrix, Hessian.

Remarks

par1 must be created using the **pvPack** procedures.

Example

```
#include optim.sdf
struct PV p1;
struct DS d0;

p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2, "P");
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc fct(struct PV p0, struct DS d0);
    local p,y;

    p = pvUnpack(p0, "P");
    y = p[1] * exp( -p[2] * d0.dataMatrix);
    retp(y);
endp;
```

```
    h = hessMT(&fct,p1,d0);
```

Source

hessmt.src

hessMTg

Purpose

Computes numerical Hessian using gradient procedure.

Include

optim.sdf

Format

h = **hessMTg**(*&gfct*, *par1*, *data1*);

Input

&gfct	scalar, pointer to procedure computing either 1xK gradient or NxK Jacobian.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *gfct*.

Output

h	KxK matrix, Hessian.

Remarks

par1 must be created using the **pvPack** procedures.

Example

```
#include optim.sdf
struct PV p1;
struct DS d0;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2, "P");
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc gfct(&fct, struct PV p0, struct DS d0);
   local p,y,g1,g2;

   p = pvUnpack(p0, "P");
   g1 = exp(-p[2] * d0.dataMatrix);
   y = p[1] * exp( -p[2] * d0.dataMatrix);
   g2 = -p[1] * d0.dataMatrix .* g1;
  retp(g1~g2);
endp;

h = hessMTg(&gfct,p1,d0);
```

Source

hessmt.src

hessMTgw

Purpose

Computes numerical Hessian using gradient procedure with weights.

Include

```
optim.sdf
```

Format

h = **hessMTgw**(*&gfct*, *par1*, *data1*, *wgts*);

Input

&gfct	scalar, pointer to procedure computing either NxK Jacobian.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *gfct*.
wgts	Nx1 vector.

Output

h	KxK matrix, Hessian.

Remarks

par1 must be created using the **pvPack** procedures.

Example

```
#includeoptim.sdf
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2, "P");
struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);
wgts = zeros(5,1) | ones(10,1);

proc gfct(&fct, struct PV p0, struct DS d0);
   local p,y,g1,g2;

   p = pvUnpack(p0, "P");
   g1 = exp(-p[2] * d0.dataMatrix);
   y = p[1] * exp( -p[2] * d0.dataMatrix);
   g2 = -p[1] * d0.dataMatrix .* g1;
   retp(g1~g2);
endp;

h = hessMTgw(&gfct,p1,d0,wgts);
```

Source

hessmt.src

hessMTm

Purpose

Computes numerical Hessian with mask.

Include

`optim.sdf`

Format

h = **hessMTm**(*&fct*, *par1*, *data1*, *mask*);

Input

&fct	scalar, pointer to procedure returning either Nx1 vector or scalar.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.
mask	KxK matrix, elements in *h* corresponding to elements of mask set to zero are not computed, otherwise are computed.

Output

h	KxK matrix, Hessian.

Remarks

`par1` must be created using the **pvPack** procedures. Only lower left part of mask looked at.

Example

```
#include optim.sdf
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2, "P");
struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

mask = { 1 1,
         1 0 };

 proc fct(struct PV p0, struct DS d0);
   local p,y;

   p = pvUnpack(p0, "P");
   y = p[1] * exp( -p[2] * d0.dataMatrix);
  retp(y);
endp;

h = hessMTm(&fct,p1,d0,mask);
```

Source

hessmt.src

hessMTmw

Purpose

Computes numerical Hessian with mask and weights.

Include

`optim.sdf`

Format

h = **hessMTmw**(*&fct*, *par1*, *data1*, *mask*, *wgts*);

Input

&fct	scalar, pointer to procedure returning Nx1 vector.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.
mask	KxK matrix, elements in *h* corresponding to elements of mask set to zero are not computed, otherwise are computed.
wgts	Nx1 vector, weights.

Output

h	KxK matrix, Hessian.

Remarks

fct must evaluate to an Nx1 vector conformable to the weight vector. *par1* must be created using the **pvPack** procedures.

Example

```
#include optim.sdf

struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2, "P");
struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);
wgts = zeros(5,1) | ones(10,1);

mask = { 1 1,
         1 0 };

proc fct(&fct, struct PV p0, struct DS d0, wgts);
   local p,y;

   p = pvUnpack(p0, "P");
   y = p[1] * exp( -p[2] * d0.dataMatrix);
  retp(y);
endp;

h = hessMTmw(&fct,p1,d0,mask,wgt);
```

Source

hessmt.src

hessMTT

Purpose

Computes numerical Hessian using available threads.

Format

```
h = hessMTT(&fct,par1,data1);
```

Include

`optim.sdf`

Input

fct	scalar, pointer to procedure returning either Nx1 vector or 1x1 scalar.
par1	structure of type **PV** containing parameter vector at which Hessian is to be evaluated
data1	structure of type **DS** containing any data needed by *fct*

Output

h	KxK matrix, Hessian

Remarks

par1 must be created using the **pvPack** procedures

Example

```
#include optim.sdf
struct PV p1;
p1 = pvCreate;

p1 = pvPack(p1,0.1|0.2, "P");
struct DS d0;
d0 = dsCreate;

d0.dataMatrix = seqa(1,1,15);

proc fct(struct PV p0, struct DS d0);
    local p,y;
    p = pvUnpack(p0, "P");
    y = p[1] * exp( -p[2] * d0.dataMatrix);
    retp(y);
endp;

h = hessMTT(&fct,p1,d0);
```

Source

hessmtt.src

hessMTTg

Purpose

Computes numerical Hessian using gradient procedure with available threads.

Include

optim.sdf

Format

h = **hessMTTg**(*&gfct*, *par1*, *data1*);

Input

&gfct	scalar, pointer to procedure computing either 1xK gradient or NxK Jacobian
par1	structure of type PV containing parameter vector at which Hessian is to be evaluated
data1	structure of type DS containing any data needed by *fct*

Output

h	KxK matrix, Hessian

Remarks

par1 must be created using the **pvPack** procedures.

Example

```
    #include optim.sdf
```

```
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2, "P");

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc gfct(&fct, struct PV p0, struct DS d0, wgt);
   local p,y,g1,g2;
   p = pvUnpack(p0, "P");
   g1 = exp(-p[2] * d0.dataMatrix);
   y = p[1] * exp( -p[2] * d0.dataMatrix);
   g2 = -p[1] * d0.dataMatrix .* g1;
   retp(g1~g2);
endp;

h = hessMTTg(&gfct,p1,d0);
```

Source

hessmtt.src

hessMTTgw

Purpose

Computes numerical Hessian using gradient procedure with weights and using available threads.

Include

optim.sdf

Format

h = **hessMTTgw**(*&gfct*, *par1*, *data1*, *wgts*);

Input

gfct	scalar, pointer to procedure computing either 1xK gradient or NxK Jacobian
par1	structure of type PV containing parameter vector at which Hessian is to be evaluated
data1	structure of type DS containing any data needed by fct
wgts	Nx1 vector, weights

Output

h	KxK matrix, Hessian

Remarks

par1 must be created using the **pvPack** procedures.

Example

```
#include optim.sdf

struct PV p1;
p1 = pvCreate;
```

```
p1 = pvPack(p1,0.1|0.2, "P");

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);
wgts = zeros(5,1) | ones(10,1);

proc gfct(&fct, struct PV p0, struct DS d0);
   local p,y,g1,g2;
   p = pvUnpack(p0, "P");
   g1 = exp(-p[2] * d0.dataMatrix);
   y = p[1] * exp( -p[2] * d0.dataMatrix);
   g2 = -p[1] * d0.dataMatrix .* g1;
   retp(g1~g2);
endp;

h = hessMTTg(&gfct,p1,d0,wgts);
```

Source

hessmtt.src

hessMTTm

Purpose

Computes numerical Hessian with mask using available threads.

Include

optim.sdf

Format

h = **hessMTTm**(*&fct*, *par1*, *data1*, *mask*);

Input

fct	scalar, pointer to procedure returning either Nx1 vector or 1x1 scalar.
par1	structure of type **PV** containing parameter vector at which Hessian is to be evaluated
data1	structure of type **DS** containing any data needed by *fct*
mask	KxK matrix, elements in *h* corresponding to elements of mask set to zero are not computed otherwise are computed

Output

h	KxK matrix, Hessian

Remarks

par1 must be created using the **pvPack** procedures. Only lower left part of mask looked at.

Example

```
#include optim.sdf
struct PV p1;
```

```
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2, "P");
struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

mask = { 1 1
         1 0 };

proc fct(struct PV p0, struct DS d0);
   local p,y;
   p = pvUnpack(p0, "P");
   y = p[1] * exp( -p[2] * d0.dataMatrix);
  retp(y);
endp;

h = hessMTTm(&fct,p1,d0,mask);
```

Source

`hessmtt.src`

hessMTw

Purpose

Computes numerical Hessian with weights.

Include

`optim.sdf`

Format

h = **hessMTw**(*&fct*, *par1*, *data1*, *wgts*);

Input

&fct	scalar, pointer to procedure returning Nx1 vector.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.
wgts	Nx1 vector, weights.

Output

h	KxK matrix, Hessian.

Remarks

fct must evaluate to an Nx1 vector conformable to the weight vector. *par1* must be created using the **pvPack** procedures.

Example

```
#includeoptim.sdf
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2, "P");
```

```
struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);
wgt = zeros(5,1) | ones(10,1);

proc fct(&fct, struct PV p0, struct DS d0, wgt);
    local p,y;

    p = pvUnpack(p0, "P");
    y = p[1] * exp( -p[2] * d0.dataMatrix);
    retp(y);
endp;

h = hessMTw(&fct,p1,d0,wgt);
```

Source

hessmt.src

hessp, hesscplx

Purpose

Computes the matrix of second partial derivatives (Hessian matrix) of a function defined as a procedure. **hesscplx** allows for complex arguments.

Format

h = **hessp**(*&f*, *x0*);

Input

&f	pointer to a single-valued function *f*(*x*), defined as a procedure, taking a single Kx1 vector argument (*f*: Kx1 → 1x1); *f*(*x*) may be defined in terms of global arguments in addition to *x*.
x0	Kx1 vector specifying the point at which the Hessian of *f*(*x*) is to be computed.

Output

h	KxK matrix of second derivatives of *f* with respect to *x* at *x0*; this matrix will be symmetric.

Remarks

This procedure requires K*(K+1)/2 function evaluations. Thus if K is large, it may take a long time to compute the Hessian matrix.

No more than 3-4 digit accuracy should be expected from this function, though it is possible for greater accuracy to be achieved with some functions.

It is important that the function be properly scaled, in order to obtain greatest possible accuracy. Specifically, scale it so that the first derivatives are approximately the same size. If these derivatives differ by more than a factor of 100 or so, the results can be meaningless.

Example

```
    x = { 1, 2, 3 };
```

```
proc g(b);
retp( exp(x'b));
endp;

b0 = { 3, 2, 1 };
h = hessp(&g,b0);
```

The resulting matrix of second partial derivatives of **g(b)** evaluated at *b=b0* is:

```
        22026.865  44053.686  66080.596
h =     44053.686  88107.753 132161.059
        66080.596 132161.059 198240.695
```

Source

hessp.src

See Also

gradp, gradcplx

hist

Purpose

Computes and graphs a frequency histogram for a vector. The actual frequencies are plotted for each category. Note: this function is for use with the deprecated PQG graphics. **plotHist** instead.

Library

pgraph

Format

```
{ b, m, freq } = hist(x, v);
```

Input

x	Mx1 vector of data.
v	Nx1 vector, the breakpoints to be used to compute the frequencies - or - scalar, the number of categories.

Output

b	Px1 vector, the breakpoints used for each category.
m	Px1 vector, the midpoints of each category.
freq	Px1 vector of computed frequency counts.

Remarks

If a vector of breakpoints is specified, a final breakpoint equal to the maximum value of *x* will be added if the maximum breakpoint value is smaller.

If a number of categories is specified, the data will be divided into *v* evenly spaced categories.

Each time an element falls into one of the categories specified in *b*, the corresponding element of *freq* will be incremented by one. The categories are interpreted as follows:

```
freq[1] =           x < b[1]
freq[2] = b[1]    < x < b[2]
freq[3] = b[2]    < x < b[3]
   .
   .
   .
freq[P] = b[P-1] < x < b[P]
```

Example

```
library pgraph;
x = rndn(5000,1);
{ b,m,f } = hist(x,20);
```

Source

phist.src

See Also

histp, histf, bar

histf

Purpose

Graphs a histogram given a vector of frequency counts. Note: This function is for use with the deprecated PQG graphics. Use **plotSetHistF** instead.

Library

pgraph

Format

```
histf(f, c);
```

Input

f	Nx1 vector, frequencies to be graphed.
c	Nx1 vector, numeric labels for categories. If this is a scalar 0, a sequence from 1 to **rows**(*f*) will be created.

Remarks

The axes are not automatically labeled. Use **xlabel** for the category axis and **ylabel** for the frequency axis.

Source

phist.src

See Also

hist, bar, xlabel, ylabel

histp

Purpose

Computes and graphs a percent frequency histogram of a vector. The percentages in each category are plotted.

Library

`pgraph`

Format

{ *b*, *m*, *freq* } = **histp**(*x*, *v*);

Input

x	Mx1 vector of data.
v	Nx1 vector, the breakpoints to be used to compute the frequencies - or - scalar, the number of categories.

Output

b	Px1 vector, the breakpoints used for each category.
m	Px1 vector, the midpoints of each category.
freq	Px1 vector of computed frequency counts. This is the vector

of counts, not percentages.

Remarks

If a vector of breakpoints is specified, a final breakpoint equal to the maximum value of *x* will be added if the maximum breakpoint value is smaller.

If a number of categories is specified, the data will be divided into *v* evenly spaced categories.

Each time an element falls into one of the categories specified in *b*, the corresponding element of *freq* will be incremented by one. The categories are interpreted as follows:

```
    freq[1] =           x < b[1]
    freq[2] = b[1]    < x < b[2]
    freq[3] = b[2]    < x < b[3]
      .
      .
      .
    freq[P] = b[P-1] < x < b[P]
```

Source

`phist.src`

See Also

hist, histf, bar

hsec

Purpose

Returns the number of hundredths of a second since midnight.

Format

y = **hsec**;

Output

y	scalar, hundredths of a second since midnight.

Remarks

The number of hundredths of a second since midnight can also be accessed as the [4,1] element of the vector returned by the **date** function.

Example

```
x = rndu(1000,1000);
tStart = hsec;

y = x*x;
tTotal = hsec-tEnd;
```

In this example, **hsec** is used to time a 1000x1000 multiplication in **GAUSS**. A 1000x1000 matrix, *x*, is created, and the current time, in hundredths of a second since midnight, is stored in the variable *tStart*. Then the multiplication is carried out.

Finally, `tStart` is subtracted from **`hsec`** to give the time difference which is assigned to `tTotal`.

See Also

date, time, timestr, ethsec, etstr

if, else, elseif

Purpose

Controls program flow with conditional branching.

Format

```
if scalar_expression;
  list of statements;
elseif scalar_expression;
  list of statements;
elseif scalar_expression;
  list of statements;
else;
  list of statements;
endif;
```

Remarks

scalar_expression is any expression that returns a scalar. It is TRUE if it is not zero, and FALSE if it is zero.

A *list of statements* is any set of **GAUSS** statements.

GAUSS will test the expression after the if statement. If it is TRUE (nonzero), then the first list of statements is executed. If it is FALSE (zero), then **GAUSS** will move to the expression after the first elseif statement, if there is one, and test it. It will keep testing expressions and will execute the first list of statements that corresponds to a TRUE expression. If no expression is TRUE, then the list of statements following the else statement is executed. After the appropriate list of statements is executed, the program will go to the statement following the endif and continue on.

if statements can be nested.

One endif is required per if statement. If an else statement is used, there may be only one per if statement. There may be as many elseif's as are required. There need not be any elseif's or any else statement within an if statement.

Note the semicolon after the else statement.

Example

```
    if x < 0;
       y = -1;
    elseif x > 0;
       y = 1;
    else;
       y = 0;
    endif;
```

See Also

do

imag

Purpose

Returns the imaginary part of *x*.

Format

zi = **imag**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

zi	NxK matrix or N-dimensional array, the imaginary part of *x*.

Remarks

If *x* is real, *zi* will be an NxK matrix or N-dimensional array of zeros.

Example

```
x = { 4i  9     3,
        2   5-6i  7i };
 y = imag(x);
```

```
y = 4    0    0
    0   -6    7
```

See Also

complex, real

#include

Purpose

Inserts code from another file into a **GAUSS** program.

Format

```
#include filename
#include "filename"
```

Remarks

filename can be any legitimate file name.

This command makes it possible to write a section of general-purpose code, and insert it into other programs.

The code from the `#include`'d file is inserted literally as if it were merged into that place in the program with a text editor.

If a path is specified for the file, then no additional searching will be attempted if the file is not found.

If a path is not specified, the current directory will be searched first, then each directory listed in *src_path*. *src_path* is defined in `gauss.cfg`.

`#include /gauss/myprog.prc`	No additional search will be made if the file is not found.

`#include myprog.prc`	The directories listed in *`src_path`* will be searched for `myprog.prc` if the file is not found in the current directory.

Compile time errors will return the line number and the name of the file in which they occur. For execution time errors, if a program is compiled with `#lineson`, the line number and name of the file where the error occurred will be printed. For files that have been `#include`'d this reflects the actual line number within the `#include`'d file. See `#lineson` for a more complete discussion of the use of and the validity of line numbers when debugging.

Example

```
    #include "/gauss/inc/cond.inc"
```

The command will cause the code in the program `cond.inc` to be merged into the current program at the point at which this statement appears.

See Also

run, lineson

indcv

Purpose

Checks one character vector against another and returns the indices of the elements of the first vector in the second vector.

Format

```
z = indcv(what, where);
```

Input

what	Nx1 character vector which contains the elements to be found in vector *where*.
where	Mx1 character vector to be searched for matches to the elements of *what*.

Output

z	Nx1 vector of integers containing the indices of the corresponding element of *what* in *where*.

Remarks

If no matches are found for any of the elements in *what*, then the corresponding elements in the returned vector are set to the **GAUSS** missing value code.

Both arguments will be forced to uppercase before the comparison.

If there are duplicate elements in *where*, the index of the first match will be returned.

Example

```
    let newVars = YEARS BONUS GENDER;
    let what = AGE PAY SEX;
    let where = AGE SEX JOB DATE PAY;
```

```
//Return the indices in 'where' of the items in 'what'
z = indcv(what,where);

//Replace AGE, PAY, SEX with YEARS, BONUS, GENDER
where[z] = newVars;
```

After the code above:

```
             YEARS
            GENDER        1
where =        JOB    z = 5
              DATE        2
             BONUS
```

See Also

indnv, indsav

indexcat

Purpose

Returns the indices of the elements of a vector which fall into a specified category

Format

y = **indexcat**(*x*, *range*);

Input

x	Nx1 vector.
range	scalar or 2x1 vector. If scalar, the function returns the indices of all elements of *x* equal to *range*. If 2x1, then the function returns the indices of all elements of *x* that fall into the range: `range[1] < x ≤ range[2]` If *range* is scalar, it can contain a single missing to specify the missing value as the category.

Output

y	Lx1 vector, containing the indices of the elements of *x* which fall into the category defined by *range*. It will contain error code 13 if there are no elements in this category.

Remarks

Use a loop to pull out indices of multiple categories.

Example

Example 1

Locate and extract the values that fall in a specified range:

```
x = { 1.0, 4.0, 3.3, 4.2, 6.0, 5.7, 8.1, 5.5 };
range = { 4, 6 };
indx = indexcat(x,range);

in_bounds = x[indx]
```

```
       4                4.20
indx = 5   in_bounds = 6.00
       6                5.70
       8                5.50
```

Example 2

Return the indices of the elements of a vector that match a specific value:

```
x = { 3, 7, 9, 1, 9 };
val = 9;
indx = indexcat(x,val);
```

```
indx = 3
       5
```

Example 3

If no matching values are found, **indexcat** will return a scalar error code. Error codes are a special type of missing value that can contain an integer error code. The return from **indexcat** can be checked with **isinfnanmiss** like this:

```
x = { 1, 2, 3, 9, 10 };
range = { 5, 7 };
indx = indexcat(x, range);

//Check to see if 'indexcat' found any matches
if isinfnanmiss(indx);
   print "no matching values found";
endif;
```

The integer value inside of an error code can be retrieved with the **`scalerr`** command. As stated earlier, if **`indexcat`** cannot find a match, it will return an error code that contains the value 13.

indices

Purpose

Processes a set of variable names or indices and returns a vector of variable names and a vector of indices.

Format

{ *`name`*, *`indx`* } = **`indices`**(*`dataset`*, *`vars`*);

Input

`dataset`	string, the name of the data set.
`vars`	Nx1 vector, a character vector of names or a numeric vector of column indices. If scalar 0, all variables in the data set will be selected.

Output

`name`	Nx1 character vector, the names associated with *`vars`*.
`indx`	Nx1 numeric vector, the column indices

associated with *vars*.

Remarks

If an error occurs, **indices** will either return a scalar error code or terminate the program with an error message, depending on the `trap` state. If the low order bit of the trap flag is 0, **indices** will terminate with an error message. If the low order bit of the trap flag is 1, **indices** will return an error code. The value of the trap flag can be tested with **trapchk**; the return from **indices** can be tested with **scalerr**. You only need to check one argument; they will both be the same. The following error codes are possible:

1	Can't open dataset.
2	Index of variable out of range, or undefined data set variables.

Source

`indices.src`

indices2

Purpose

Processes two sets of variable names or indices from a single file. The first is a single variable and the second is a set of variables. The first must not occur in the second set and all must be in the file.

Format

{ *name1*, *indx1*, *name2*, *indx2* } = **indices2**(*dataset*, *var1*, *var2*);

Input

dataset	string, the name of the data set.
var1	string or scalar, variable name or index. This can be either the name of the variable, or the column index of the variable. If null or 0, the last variable in the data set will be used.
var2	Nx1 vector, a character vector of names or a numeric vector of column indices. If scalar 0, all variables in the data set except the one associated with *var1* will be selected.

Output

name1	scalar character matrix containing the name of the variable associated with *var1*.
indx1	scalar, the column index of *var1*.
name2	Nx1 character vector, the names associated with *var2*.
indx2	Nx1 numeric vector, the column indices of *var2*.

Remarks

If an error occurs, **indices2** will either return a scalar error code or terminate the program with an error message, depending on the `trap` state. If the low order bit of the trap flag is 0, **indices2** will terminate with an error message. If the low order bit of the trap flag is 1, **indices2** will return an error code. The value of the trap flag can be tested with **trapchk**; the return from **indices2** can be tested with **scalerr**. You only need to check one argument; they will all be the same. The following error codes are possible:

1	Can't open dataset.
2	Index of variable out of range, or undefined data set variables.
3	First variable must be a single name or index.
4	First variable contained in second set.

Source

`indices2.src`

indicesf

Purpose

Processes a set of variable names or indices and returns a vector of variable names and a vector of indices.

Format

{ *name*, *indx* } = **indicesf**(*fp*, *namein*, *indxin*);

Input

fp	scalar, file handle of an open data set.
namein	Nx1 string array, names of selected columns in the data set. If set to a null string, columns are selected using *indxin*
indxin	Nx1 vector, indices of selected columns in the data set. If set to 0, columns are selected using *namein*.

Output

name	Nx1 string array, the names of the selected columns.
indx	Nx1 vector, the indices of the selected columns.

Remarks

If *namein* is a null string and *indxin* is 0, all columns of the data set will be selected.

If an error occurs, *indx* will be set to a scalar error code. The following error codes are possible:

1	Can't open data file
2	Variable not found
3	Indices outside of range of columns

Source

`indices.src`

See Also

indicesfn, indices

indicesfn

Purpose

Processes a set of variable names or indices and returns a vector of variable names and a vector of indices.

Format

```
{ name, indx } = indicesfn(dataset, namein, indxin);
```

Input

dataset	string, name of the data set.
namein	Nx1 string array, names of selected columns in the data set. If set to a null string, columns are selected using *indxin*
indxin	Nx1 vector, indices of selected columns in the data set. If set to 0, columns are selected using *namein*.

Output

name	Nx1 string array, the names of the selected columns.
indx	Nx1 vector, the indices of the selected columns.

Remarks

If *namein* is a null string and *indxin* is 0, all columns of the data set will be selected.

If an error occurs, *indx* will be set to a scalar error code. The following error codes are possible:

1	Can't open data file
2	Variable not found
3	Indices outside of range of columns

Source

`indices.src`

See Also

indicesf, indices

indnv

Purpose

Checks one numeric vector against another and returns the indices of the elements of the first vector in the second vector.

Format

z = **`indnv`**(*`what`*, *`where`*);

Input

`what`	Nx1 numeric vector which contains the values to be found in vector *`where`*.
`where`	Mx1 numeric vector to be searched for matches to the values in *`what`*.

Output

z	Nx1 vector of integers, the indices of the corresponding elements of *`what`* in *`where`*.

Remarks

If no matches are found for any of the elements in *`what`*, then those elements in the returned vector are set to the **GAUSS** missing value code.

If there are duplicate elements in *`where`*, the index of the first match will be returned.

Example

```
what = { 8, 7, 3 };
where = { 2, 7, 8, 4, 3 };
z = indnv(what,where);
```

```
        3
z = 2
        5
```

indsav

Purpose

Checks one string array against another and returns the indices of the first string array in the second string array.

Format

indx = **indsav**(*what*, *where*);

Input

what	Nx1 string array which contains the values to be found in vector *where*.
where	Mx1 string array to be searched for the corresponding elements of *what*.

Output

indx	Nx1 vector of indices, the values of *what* in *where*.

Remarks

If no matches are found, those elements in the returned vector are set to the **GAUSS** missing value code.

If there are duplicate elements in *where*, the index of the first match will be returned.

intgrat2

Purpose

Integrates the following double integral, using user-defined functions **f**, $\mathbf{g}_1$ and $\mathbf{g}_2$ and scalars *a* and *b*:

$$\int_a^b \int_{g_2(x)}^{g_1(x)} f(x, y)\,dydx$$

Format

y = **intgrat2**(*&f*, *xl*, *gl*);

Input

&f	scalar, pointer to the procedure containing the function to be integrated.

xl	2x1 or 2xN matrix, the limits of *x*. These must be scalar limits.
gl	2x1 or 2xN matrix of function pointers, the limits of *y*. For *xl* and *gl*, the first row is the upper limit and the second row is the lower limit. N integrations are computed.

Global Input

_intord	scalar, the order of the integration. The larger *_intord*, the more precise the final result will be. *_intord* may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.
_intrec	scalar. This variable is used to keep track of the level of recursion of **intgrat2** and may start out with a different value if your program terminated inside of the integration function on a previous run. Always set *_intrec* explicitly to 0 before any call to **intgrat2**.

Output

y	Nx1 vector of the estimated integral(s) of $f(x, y)$, evaluated between the limits given by *xl* and *gl*.

Remarks

The user-defined functions specified by **f** and **gl** must either

1. Return a scalar constant

   - or -

2. Return a vector of function values. **intgrat2** will pass to user-defined functions a vector or matrix for *x* and *y* and expect a vector or matrix to be returned. Use `.*` and `./` instead of `*` and `/`.

Example

```
proc f(x,y);
retp(cos(x) + 1).*(sin(y) + 1));
endp;

proc g1(x);
retp(sqrt(1-x^2));
endp;

proc g2(x);
retp(0);
endp;

xl = 1|-1;
g0 = &g1|&g2;
_intord = 40;
_intrec = 0;
y = intgrat2(&f,xl,g0);
```

This will integrate the function

```
f(x,y) = (cos(x)+1)(sin(y)+1)
```

over the upper half of the unit circle. Note the use of the `.*` operator instead of just `*` in the

definition of *f*(*x*,*y*). This allows *f* to return a vector or matrix of function values.

Source

`intgrat.src`

Globals

_intord, _intq12, _intq16, _intq2, _intq20, _intq24, _intq3, _intq32, _intq4, _intq40, _intq6, _intq8, _intrec

See Also

intgrat3, intquad1, intquad2, intquad3, intsimp

intgrat3

Purpose

Integrates the following triple integral, using user-defined functions and scalars for bounds:

$$\int_{a}^{b}\int_{g_2(x)}^{g_1(x)}\int_{h_2(x,y)}^{h_1(x,y)} f(x,y,z)\,dz\,dy\,dx$$

Format

y = **intgrat3**(*&f*, *xl*, *gl*, *hl*);

Input

&f	scalar, pointer to the procedure containing the function to be integrated. *f* is a function of (*x*, *y*, *z*).

x1	2x1 or 2xN matrix, the limits of *x*. These must be scalar limits.
g1	2x1 or 2xN matrix of function pointers. These procedures are functions of *x*.
h1	2x1 or 2xN matrix of function pointers. These procedures are functions of *x* and *y*.

For *x1*, *g1*, and *h1*, the first row is the upper limit and the second row is the lower limit. N integrations are computed.

Global Input

_intord	scalar, the order of the integration. The larger *_intord*, the more precise the final result will be. *_intord* may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.
_intrec	scalar. This variable is used to keep track of the level of recursion of **intgrat3** and may start out with a different value if your program terminated inside of the integration function on a previous run. Always set *_intrec* explicitly to 0 before any call to **intgrat3**.

Output

y	Nx1 vector of the estimated integral(s) of $f(x,y,z)$ evaluated between the limits given by *x1*, *g1* and *h1*.

Remarks

User-defined functions `f`, and those used in `g1` and `h1` must either:

1. Return a scalar constant

   - or -

2. Return a vector of function values. **`intgrat3`** will pass to user-defined functions a vector or matrix for *x* and *y* and expect a vector or matrix to be returned. Use `.*` and `./` operators instead of just `*` and `/`.

Example

```
proc f(x,y,z);
retp(2);
endp;

proc g1(x);
retp(sqrt(25-x^2));
endp;

proc g2(x);
retp(-g1(x));
endp;

proc h1(x,y);
retp(sqrt(25 - x^2 - y^2));
endp;

proc h2(x,y);
retp(-h1(x,y));
endp;

xl = 5|-5;
```

```
    g0 = &g1|&g2;
    h0 = &h1|&h2;

    _intrec = 0;
    _intord = 40;

    y = intgrat3(&f,xl,g0,h0);
```

This will integrate the function *f*(*x*,*y*,*z*) over the sphere of radius 5. The result will be approximately twice the volume of a sphere of radius 5.

Source

`intgrat.src`

Globals

_intord, *_intq12*, *_intq16*, *_intq2*, *_intq20*, *_intq24*, *_intq3*, *_intq32*, *_intq4*, *_intq40*, *_intq6*, *_intq8*, *_intrec*

See Also

intgrat2, intquad1, intquad2, intquad3, intsimp

inthp1

Purpose

Integrates a user-defined function over an infinite interval.

Include

`inthp.sdf`

Format

```
y = inthp1(&f, pds, ctl);
```

Input

&f	scalar, pointer to the procedure containing the function to be integrated.
pds	scalar, pointer to instance of a **DS** structure. The members of the **DS** are:
	pds->dataMatrix — NxK matrix.
	pds->dataArray — NxKxL... array.
	pds->vnames — string array.
	pds->dsname — string.
	pds->type — scalar.
	The contents, if any, are set by the user and are passed by **inthp1** to the user-provided function without modification.
ctl	instance of an **inthpControl** structure with members
	ctl.maxEvaluations — scalar, maximum number of function evaluations, default = 1e5;
	ctl.p — scalar, termination parameter
	0 — heuristic termination, default.
	1 — deterministic

		termination with infinity norm.
	2,...	deterministic termination with p-th norm.
ctl.d	scalar termination parameter	
	1	if heuristic termination
	0 < *ctl.d* < $\pi/2$	if deterministic termination
ctl.eps	scalar, relative error bound. Default = 1e-6.	

A default *ctl* can be generated by calling **inthpControlCreate**.

Output

y	scalar, the estimated integral of **f**(*x*) evaluated over the interval (-∞,+∞).

Remarks

The user-provided function must have the following format

```
    f(struct DS *pds, x)
```

where

pds	scalar, pointer to an instance of a **DS** structure.
x	scalar, value at which integral will be evaluated.

If *ctl.d* can be specified (see Sikorski and Stenger, 1984), deterministic termination can be specified and accuracy guaranteed. if not, the heuristic method can be used and the value of *clt.d* is disregarded.

The pointer to the instance of the data structure, *pds*, is passed untouched to the user-provided procedure computing the function to be integrated. Any information needed by that function can be put into that data structure.

Example

```
#include inthp.sdf

proc fct(struct DS *pds, x);
local var;
   var = pds->dataMatrix;
retp( exp( -(x*x) / (2*var) ));
endp;

struct DS d0;
struct DS *pds;
variance = 3;
pds = &d0;
d0.dataMatrix = variance;

struct inthpControl c0;
c0 = inthpControlCreate;

r = inthp1(&fct,pds,c0);

format /ld 16,10;
print r;
```

```
    print sqrt(2*pi*variance);
```

results in the following output:

```
     4.3416075273
     4.3416075273
```

References

1. "Optimal Quadratures in H_p Spaces" by K. Sikorski and F. Stenger, *ACM Transactions on Mathematical Software*, 10(2):140-151, June 1984.

Source

```
inthp.src
```

See Also

inthpControlCreate, inthp2, inthp3, inthp4

i

inthp2

Purpose

Integrates a user-defined function over the [a,+∞) interval.

Include

```
inthp.sdf
```

Format

y = **inthp2**(*&f*, *pds*, *ctl*, *a*);

Input

&f	scalar, pointer to the procedure containing the function to be integrated.
pds	scalar, pointer to instance of a **DS** structure. The members of the **DS** are:
	pds->dataMatrix NxK matrix.
	pds->dataArray NxKxL... array.
	pds->vnames string array.
	pds->dsname string.
	pds->type scalar.
	The contents, if any, are set by the user and are passed by **`inthp1`** to the user-provided function without modification.
ctl	instance of an **inthpControl** structure with members
	ctl.maxEvaluations scalar, maximum number of function evaluations, default = 1e5;
	ctl.p scalar, termination parameter
	0 heuristic termination, default.
	1 deterministic termination with infinity norm.
	2,... deterministic termination with p-

			th norm.
	ctl.d	scalar termination parameter	
		1	if heuristic termination
		$0 <$ *ctl.d* $< \pi/2$	if deterministic termination
	ctl.eps	scalar, relative error bound. Default = 1e-6.	
	A default *ctl* can be generated by calling **inthpControlCreate**.		
a	1xN vector, lower limits of integration.		

Output

y	Nx1 vector, the estimated integrals of **f**(x) evaluated over the interval [a,+∞).

Remarks

The user-provided function must have the following format

```
    f(struct DS *pds, x)
```

where

pds	scalar, pointer to an instance of a **DS** structure.
x	scalar, value at which integral will be evaluated.

If `ctl.d` can be specified (see Sikorski and Stenger, 1984), deterministic termination can be specified and accuracy guaranteed. If not, the heuristic method can be used and the value of `ctl.d` is disregarded.

The pointer to the instance of the data structure, `pds`, is passed untouched to the user-provided procedure computing the function to be integrated. Any information needed by that function can be put into that data structure.

Example

```
#include inthp.sdf

proc normal(struct DS *pd0, x);
   local var;
   var = pd0->dataMatrix;
   retp( (1/sqrt(2*pi*var))*exp( -(x*x) / (2*var) ));
endp;


struct DS d0;
struct DS *pd0;

pd0 = &d0;

struct inthpControl c0;
c0 = inthpControlCreate;

lim = 2;

c0.d = pi/4;
c0.p = 2;

var = 1;
d0.dataMatrix = var;
```

```
    r = inthp2(&normal,pd0,c0,lim);

    format /ld 16,10;
    print r;
    print cdfnc(2);
```

produces the following output:

```
     0.0227501281
     0.0227501319
```

References

1. "Optimal Quadratures in H_p Spaces" by K. Sikorski and F. Stenger, *ACM Transactions on Mathematical Software*, 10(2):140-151, June 1984.

Source

`inthp.src`

See Also

inthpControlCreate, inthp1, inthp3, inthp4

inthp3

Purpose

Integrates a user-defined function over the [a,+∞) interval that is oscillatory.

Include

`inthp.sdf`

Format

```
y = inthp3(&f, pds, ctl, a);
```

Input

&f	scalar, pointer to the procedure containing the function to be integrated.
pds	scalar, pointer to instance of a **DS** structure. The members of the **DS** are:

pds->dataMatrix	NxK matrix.
pds->dataArray	NxKxL... array.
pds->vnames	string array.
pds->dsname	string.
pds->type	scalar.

The contents, if any, are set by the user and are passed by **inthp1** to the user-provided function without modification.

ctl instance of an **inthpControl** structure with members

ctl.maxEvaluations	scalar, maximum number of function evaluations, default = 1e5;	
ctl.p	scalar, termination parameter	
	0	heuristic termination, default.
	1	deterministic termination with

			infinity norm.
		2,...	deterministic termination with p-th norm.
	`ctl.d`	scalar termination parameter	
		1	if heuristic termination
		0 < `ctl.d` < `π/2`	if deterministic termination
	`ctl.eps`	scalar, relative error bound. Default = 1e-6.	
	A default `ctl` can be generated by calling **`inthpControlCreate`**.		
a	1xN vector, lower limits of integration.		

Output

y	Nx1 vector, the estimated integrals of **`f`**(x) evaluated over the interval $[a,+\infty)$.

Remarks

This procedure is designed especially for oscillatory functions.

The user-provided function must have the following format

```
f(struct DS *pds, x)
```

where

pds	scalar, pointer to an instance of a **DS** structure.
x	scalar, value at which integral will be evaluated.

If *ctl.d* can be specified (see Sikorski and Stenger, 1984), deterministic termination can be specified and accuracy guaranteed. if not, the heuristic method can be used and the value of *ctl.d* is disregarded.

The pointer to the instance of the data structure, *pds*, is passed untouched to the user-provided procedure computing the function to be integrated. Any information needed by that function can be put into that data structure.

Example

```
#include inthp.sdf

proc fct(struct DS *pd0, x);
   local m,a;
   m = pd0->dataMatrix[1];
   a = pd0->dataMatrix[2];
   retp( exp(-a*x)*cos(m*x));
endp;

struct DS d0;
struct DS *pd0;

struct inthpControl c0;
c0 = inthpControlCreate;
```

```
    c0.p = 2;
    c0.d = pi/3;

    m = 2;
    a = 1;
    pd0 = &d0;
    d0.dataMatrix = m | a;

    lim = 0;

    r = inthp3(&fct,pd0,c0,lim);

    format /ld 16,10;
    print r;
    print a/(a*a + m*m);
```

produces the following output:

```
     0.2000000000
     0.2000000000
```

References

1. "Optimal Quadratures in H_p Spaces" by K. Sikorski and F. Stenger, *ACM Transactions on Mathematical Software*, 10(2):140-151, June 1984.

Source

inthp.src

See Also

inthpControlCreate, inthp1, inthp2, inthp4

inthp4

Purpose

Integrates a user-defined function over the [*a*, *b*] interval.

Include

`inthp.sdf`

Format

y = **inthp4**(*&f*, *pds*, *ctl*, *c*);

Input

&f	scalar, pointer to the procedure containing the function to be integrated.
pds	scalar, pointer to instance of a **DS** structure. The members of the **DS** are:

pds->dataMatrix	NxK matrix.
pds->dataArray	NxKxL... array.
pds->vnames	string array.
pds->dsname	string.
pds->type	scalar.

The contents, if any, are set by the user and are passed by **`inthp1`** to the user-provided function without modification.

ctl	instance of an **inthpControl** structure with members		
	ctl.maxEvaluations	scalar, maximum number of function evaluations, default = 1e5;	
	ctl.p	scalar, termination parameter	
		0	heuristic termination, default.
		1	deterministic termination with infinity norm.
		2,...	deterministic termination with p-th norm.
	ctl.d	scalar termination parameter	
		1	if heuristic termination
		0 < *ctl.d* < *π/2*	if deterministic termination
	ctl.eps	scalar, relative error bound. Default = 1e-6.	
	A default *ctl* can be generated by calling **`inthpControlCreate`**.		
c	2×N vector, upper and lower limits of integration, the first row contains upper limits and the second row the lower.		

Output

y	Nx1 vector, the estimated integrals of **`f`**(*x*) evaluated over the interval [*a*, *b*].

Remarks

The user-provided function must have the following format

```
    f(struct DS *pds, x)
```

where

pds	scalar, pointer to an instance of a **DS** structure.
x	scalar, value at which integral will be evaluated.

If *ctl.d* can be specified (see Sikorski and Stenger, 1984), deterministic termination can be specified and accuracy guaranteed. if not, the heuristic method can be used and the value of *ctl.d* is disregarded.

The pointer to the instance of the data structure, *pds*, is passed untouched to the user-provided procedure computing the function to be integrated. Any information needed by that function can be put into that data structure.

Example

```
    #include inthp.sdf

    proc fct(struct DS *pd0, x);
       local a,b,c;
       a = pd0->dataMatrix[1];
       b = pd0->dataMatrix[2];
       c = pd0->dataMatrix[3];
       retp( 1/sqrt(a*x*x + b*x + c));
    endp;

    struct DS d0;
    struct DS *pd0;
    struct inthpControl c0;
    c0 = inthpControlCreate;

    c0.p = 2;
    c0.d = pi/2;

    a = -1;
    b = -2;
    c = 3;
    pd0 = &d0;
    d0.dataMatrix = a|b|c;

    lim = 1 | -1;

    r = inthp4(&fct,pd0,c0,lim);

    format /ld 16,10;
    print r;
    print pi/2;
```

produces the following output:

```
     1.5707962283
     1.5707963268
```

References

1. "Optimal Quadratures in H_p Spaces" by K. Sikorski and F. Stenger, *ACM Transactions on Mathematical Software*, 10(2):140-151, June 1984.

Source

```
inthp.src
```

See Also

inthpControlCreate, inthp1, inthp2, inthp3

inthpControlCreate

Purpose

Creates default **inthpControl** structure.

Include

```
inthp.sdf
```

Format

c = **`inthpControlCreate`**();

Output

c	instance of **inthpControl** structure with members set to default values.

Source

```
inthp.src
```

See Also

inthp1, inthp2, inthp3, inthp4

intquad1

Purpose

Integrates a specified function using Gauss-Legendre quadrature. A suite of upper and lower bounds may be calculated in one procedure call.

Format

```
y = intquad1(&f, xl);
```

Input

&f	scalar, pointer to the procedure containing the function to be integrated. This must be a function of *x*.
xl	2xN matrix, the limits of *x*. The first row is the upper limit and the second row is the lower limit. N integrations are computed.

Global Input

_intord	scalar, the order of the integration. The larger *_intord*, the more precise the final result will be. *_intord* may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.

Output

y	Nx1 vector of the estimated integral(s) of **f**(*x*) evaluated between the limits given by *xl*.

Remarks

The user-defined function **f** must return a vector of function values. **intquad1** will pass to the user-defined function a vector or matrix for *x* and expect a vector or matrix to be returned. Use the `.*` and `./` instead of `*` and `/`.

Example

```
proc f(x);
   retp(x.*sin(x));
endp;

xl = 1|0;
y = intquad1(&f,xl);
```

This will integrate the function $f(x) = x*sin(x)$ between 0 and 1. Note the use of the `.*` instead of `*`.

Source

integral.src

Globals

_intord, _intq12, _intq16, _intq2, _intq20, _intq24, _intq3, _intq32, _intq4, _intq40, _intq6, _intq8

See Also

intsimp, intquad2, intquad3, intgrat2, intgrat3

intquad2

Purpose

Integrates a specified function using Gauss-Legendre quadrature. A suite of upper and lower bounds may be calculated in one procedure call.

Format

```
y = intquad2(&f, xl, yl);
```

Input

&f	scalar, pointer to the procedure containing the function to be integrated.
xl	2x1 or 2xN matrix, the limits of *x*.
yl	2x1 or 2xN matrix, the limits of *y*.

For *xl* and *yl*, the first row is the upper limit and the second row is the lower limit. N integrations are computed.

Global Input

_intord	scalar, the order of the integration. The larger *_intord*, the more precise the final result will be. *_intord* may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40.

	Default = 12.
_intrec	scalar. This variable is used to keep track of the level of recursion of **intquad2** and may start out with a different value if your program terminated inside of the integration function on a previous run. Always set *_intrec* explicitly to 0 before any calls to **intquad2**.

Output

y	Nx1 vector of the estimated integral(s) of *f*(*x*,*y*) evaluated between the limits given by *x1* and *y1*.

Remarks

The user-defined function *f* must return a vector of function values. **intquad2** will pass to user-defined functions a vector or matrix for *x* and *y* and expect a vector or matrix to be returned. Use .* and ./ instead of * and /.

intquad2 will expand scalars to the appropriate size. This means that functions can be defined to return a scalar constant. If users write their functions incorrectly (using * instead of .*, for example), **intquad2** may not compute the expected integral, but the integral of a constant function.

To integrate over a region which is bounded by functions, rather than just scalars, use **intgrat2** or **intgrat3**.

Example

```
    proc f(x,y);
    retp(x.*sin(x+y));
    endp;
```

```
xl = 1|0;
yl = 1|0;

_intrec = 0;

y = intquad2(&f,xl,yl);
```

This will integrate the function:

```
f(x) = x.*sin(x+y)
```

between $x = 0$ and 1, and between $y = 0$ and 1.

Source

`integral.src`

Globals

_intord, *_intq12*, *_intq16*, *_intq2*, *_intq20*, *_intq24*, *_intq3*, *_intq32*, *_intq4*, *_intq40*, *_intq6*, *_intq8*, *_intrec*

See Also

intquad1, intquad3, intsimp, intgrat2, intgrat3

intquad3

Purpose

Integrates a specified function using Gauss-Legendre quadrature. A suite of upper and lower bounds may be calculated in one procedure call.

Format

```
y = intquad3(&f, xl, yl, zl);
```

Input

&f	scalar, pointer to the procedure containing the function to be integrated. *f* is a function of (*x*, *y*, *z*).
xl	2x1 or 2xN matrix, the limits of *x*.
yl	2x1 or 2xN matrix, the limits of *y*.
zl	2x1 or 2xN matrix, the limits of *z*.

For *xl*, *yl*, and *zl*, the first row is the upper limit and the second row is the lower limit. N integrations are computed.

Global Input

_intord	scalar, the order of the integration. The larger *_intord*, the more precise the final result will be. *_intord* may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.
_intrec	scalar. This variable is used to keep track of the level of recursion of **intquad3** and may start out with a different value if your program terminated inside of the integration function on a previous run. Always set *_intrec* explicitly to 0 before any calls to **intquad3**.

Output

y	Nx1 vector of the estimated integral(s) of *f*(*x*,*y*,*z*) evaluated between the limits given by *xl*, *yl*, and *zl*.

Remarks

The user-defined function *f* must return a vector of function values. **intquad3** will pass to the user-defined function a vector or matrix for *x*, *y* and *z* and expect a vector or matrix to be returned. Use **.*** and **./** instead of * and **/**.

intquad3 will expand scalars to the appropriate size. This means that functions can be defined to return a scalar constant. If users write their functions incorrectly (using * instead of **.***, for example), **intquad3** may not compute the expected integral, but the integral of a constant function.

To integrate over a region which is bounded by functions, rather than just scalars, use **intgrat2** or **intgrat3**.

Example

```
proc f(x,y,z);
retp(x.*y.*z);
endp;

xl = 1|0;
yl = 1|0;
zl = { 1 2 3, 0 0 0 };

_intrec = 0;

y = intquad3(&f,xl,yl,zl);
```

This will integrate the function $f(x) = x*y*z$ over 3 sets of limits, since `z1` is defined to be a 2x3 matrix.

Source

`integral.src`

Globals

_intord, _intq12, _intq16, _intq2, _intq20, _intq24, _intq3, _intq32, _intq4, _intq40, _intq6, _intq8, _intrec

See Also

intquad1, intquad2, intsimp, intgrat2, intgrat3

intrleav

Purpose

Interleaves the rows of two files that have been sorted on a common variable to produce a single file sorted on that variable.

Format

intrleav(*infile1, infile2, outfile, keyvar, keytyp*);

Input

infile1	string, name of input file 1.
infile2	string, name of input file 2.

outfile	string, name of output file.
keyvar	string, name of key variable; this is the column the files are sorted on.
keytyp	scalar, data type of key variable.

1	numeric key, ascending order
2	character key, ascending order
-1	numeric key, descending order
-2	character key, descending order

Remarks

The two files MUST have exactly the same variables, that is, the same number of columns AND the same variable names. They must both already be sorted on the key column. This procedure will combine them into one large file, sorted by the key variable.

If the inputs are null ("" or 0), the procedure will ask for them.

Source

`sortd.src`

See Also

intrleavsa

intrleavsa

Purpose

Interleaves the rows of two string arrays that have been sorted on a common column.

Format

```
y = intrleavsa(sa1, sa2, ikey);
```

Input

sa1	NxK string array 1.
sa2	MxK string array 2.
ikey	scalar integer, index of the key column the string arrays are sorted on.

Output

y	LxK interleaved (combined) string array.

Remarks

The two string arrays MUST have exactly the same number of columns AND have been already sorted on a key column.

This procedure will combine them into one large string array, sorted by the key column.

Source

`sortd.src`

See Also

intrleav

intrsect

Purpose

Returns the intersection of two vectors, with duplicates removed.

Format

y = **`intrsect`**(*`v1`*, *`v2`*, *`flag`*);

Input

`v1`	Nx1 vector.
`v2`	Mx1 vector.
`flag`	scalar, if 1, *`v1`* and *`v2`* are numeric; if 0, character.

Output

y	Lx1 vector containing all unique values that are in both *`v1`* and *`v2`*, sorted in ascending order.

Remarks

Place smaller vector first for fastest operation.

If there are a lot of duplicates within a vector, it is faster to remove them with the function **unique** before calling **intrsect**.

Source

intrsect.src

Example

```
    v1 = { 3, 9, 5, 2, 10, 15 };
    v2 = { 4, 9, 8, 5, 12, 3, 1 };
    y = intrsect(v1,v2,1);
```

Assigns the values that are contained in both input vectors to *y*:

```
          3
    y =   5
          9
```

See Also

intrsectsa

intrsectsa

Purpose

Returns the intersection of two string vectors, with duplicates removed.

Format

```
y = intrsectsa(sv1,  sv2);
```

Input

sv1	Nx1 or 1xN string vector.
sv2	Mx1 or 1xM string vector.

Output

sy	Lx1 vector containing all unique strings that are in both *sv1* and *sv2*, sorted in ascending order.

Remarks

Place smaller vector first for fastest operation.

If there are a lot of duplicates it is faster to remove them with unique before calling **intrsectsa**.

Example

```
string sv1 = { "age", "weight", "bmi" };
string sv2 = { "hdl", "ldl", "age", "bmi", "smoking" };

sy = intrsectsa(sv1,sv2);
print "Both studies reported the following variable
s:";
print sy;
```

The code above, returns:

```
    Both studies reported the following variables:
                age              bmi
```

Source

`intrsect.src`

See Also

intrsect

intsimp

Purpose

Integrates a specified function using Simpson's method with end correction. A single integral is computed in one function call.

Format

y = **intsimp**(&f, *xl*, *tol*);

Input

&f	pointer to the procedure containing the function to be integrated.
xl	2x1 vector, the limits of *x*. The first element is the upper limit and the second element is the lower limit.

tol	The tolerance to be used in testing for convergence.

Output

y	The estimated integral of **f**(*x*) between *xl*[1] and *xl*[2].

Example

```
proc f(x);
  retp(sin(x));
endp;

let xl = { 1, 0 };

y = intsimp(&f,xl,1e-8);
print y;
```

The code above, returns the following:

```
        0.45969769
```

This will integrate the function between 0 and 1.

Source

intsimp.src

See Also

intquad1, intquad2, intquad3, intgrat2, intgrat3

inv, invpd

Purpose

`inv` returns the inverse of an invertible matrix. **`invpd`** returns the inverse of a symmetric, positive definite matrix.

Format

y = **`inv`**(*x*);
y = **`invpd`**(*x*);

Input

x	NxN matrix or K-dimensional array where the last two dimensions are NxN.

Output

y	NxN matrix or K-dimensional array where the last two dimensions are NxN, containing the inverse of *x*.

Remarks

x can be any legitimate expression that returns a matrix or array that is legal for the function.

If *x* is an array, the result will be an array containing the inverses of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a

10x4x4 array, the result will be an array of the same size containing the inverses of each of the 10 4x4 arrays contained in *x*

For **inv**, if *x* is a matrix, it must be square and invertible. Otherwise, if *x* is an array, the 2-dimensional arrays described by the last two dimensions of *x* must be square and invertible.

For **invpd**, if *x* is a matrix, it must be symmetric and positive definite. Otherwise, if *x* is an array, the 2-dimensional arrays described by the last two dimensions of *x* must be symmetric and positive definite.

If the input matrix is not invertible by these functions, they will either terminate the program with an error message or return an error code which can be tested for with the **scalerr** function. This depends on the trap state as follows:

If trap is set to 1, they will return a scalar errorcode:

inv	**invpd**
50	20

If trap is set to 0, they will terminate with an error message:

inv	**invpd**
"Matrix singular"	"Matrix not positive definite"

If the input to **invpd** is not symmetric, it is possible that the function will (erroneously) appear to operate successfully.

Positive definite matrices can be inverted by **inv**. However, for symmetric, positive definite matrices (such as moment matrices), **invpd** is about twice as fast as **inv**.

Example

```
    n = 4000;
```

```
x1 = rndn(n,1);
x = ones(n,1)~x1;
btrue = { 1, 0.5 };
y = x*btrue + rndn(n,1);
bols = invpd(x'x)*x'y;
```

After the code above, *bols* will be equal to:

```
1.00237215
0.48249445
```

This example simulates some data and computes the **ols** coefficient estimator using the **invpd** function. First, the number of observations is specified. Second, a vector *x1* of standard Normal random variables is generated and is concatenated with a vector of **ones**(to create a constant term). The true coefficients are specified, and the dependent variable *y* is created. Then the **ols** coefficient estimates are computed.

When computing least-squares problems with poorly conditioned matrices, the slash operator "**/**" and the function **olsqr** will provide greater accuracy.

invswp

Purpose

Computes a generalized sweep inverse.

Format

y = **invswp**(*x*);

Input

x	NxN matrix.

Output

y	NxN matrix, the generalized inverse of *x*.

Remarks

This will invert any general matrix. That is, even matrices which will not invert using **`inv`** because they are singular will invert using **`invswp`**.

x and *y* will satisfy the two conditions:

1. $xyx = x$
2. $yxy = y$

`invswp` returns a row and column with zeros when the pivot fails. This is good for quadratic forms since it essentially removes rows with redundant information, i.e., the statistices generated will be "correct" but with reduced degrees of freedom.

The tolerance used to determine if a pivot element is zero is taken from the **`crout`** singularity tolerance. The corresponding row and column are zeroed out. See **Singularity Tolerance**, Chapter 1.

iscplx

Purpose

Returns whether a matrix or N-dimensional array is complex or real.

Format

y = **iscplx**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	scalar, 1 if *x* is complex, 0 if it is real.

Example

```
x = { 1, 2i, 3 };
if iscplx(x);
   //code path for complex case
else;
   //code path for real case
endif;
```

See Also

hasimag, iscplxf

iscplxf

Purpose

Returns whether a data set is complex or real.

Format

y = **iscplxf**(*fh*);

Input

fh	scalar, file handle of an open file.

Output

y	scalar, 1 if the data set is complex, 0 if it is real.

See Also

hasimag, iscplx

isden

Purpose

Returns whether a scalar, matrix or N-dimensional array contains denormals.

Format

y = **isden**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	scalar, 1 if *x* contains a denormal, 0 if it does not.

Example

Sometimes denormals can unnecessarily slow down calculations and it is best to flush them to zero. This example tests whether the vector *x* contains any denormals and thensets any values between 0 and 1e-25 to be equal to 0.

```
tol = 1e-25;

//Create a vector that contains a denormal
x = { 1, exp(-724.5), 3 };

if isden(x);
   //Get the index of all elements between 0 and tol
   idx = indexcat(x,0|tol);
   //Set all elements between 0 and tol equal to 0
   x[idx] = 0;
endif;
```

Before the `if` block in the code above, the second element of *x* is equal to approximately 3e-57. After the `if` block this element is set equal to 0, the other elements of *x* are unchanged.

See Also

denToZero

isinfnanmiss

Purpose

Returns true if the argument contains an infinity, NaN, or missing value.

Format

y = **`isinfnanmiss`**(*x*);

Input

x	NxK matrix.

Output

y	scalar, 1 if *x* contains any infinities, NaNs, or missing values, else 0.

See Also

scalinfnanmiss, ismiss, scalmiss

ismiss

Purpose

Returns a 1 if its matrix argument contains any missing values, otherwise returns a 0.

Format

y = **ismiss**(*x*);

Input

x	NxK matrix.

Output

y	scalar, 1 if *x* contains any missing values, otherwise 0.

Remarks

An element of *x* is considered to be a missing if and only if it contains a missing value in the real part. Thus, if *x* = 1 + .i, **ismiss**(*x*) will return a 0.

Example

```
x = { 1, 2, 3, 4 };

//Set the second element of 'x' to be a missing value
x[2] = miss(0,0);

print "before 'if' block, x = " x;

//If there are any missing values in 'x'
if ismiss(x);
   //Remove all rows with missing values from 'x'
   x = packr(x);
endif;
```

```
print "after 'if' block, x = " x;
```

```
before 'if' block, x =
       1.0000000
               .
       3.0000000
       4.0000000
after 'if' block, x =
       1.0000000
       3.0000000
       4.0000000
```

To reset all missing values to a specified value, replace the call to **`packr`** above with a call to **`missrv`**.

See Also

scalmiss, miss, missrv

itos

Purpose

Converts a scalar or matrix to the string representation of an integer.

Format

str = **`itos`**(*x*);

Input

x	scalar or NxK matrix.

Output

y	string or string array containing the string representation of the elements of *x*.

Example

Example 1

```
x = 4;
str = itos(x);
print "x = " x;
print "str = " str;

x =
   1.000000

str =
   1
```

Example 2

```
for i(1, 4, 1);
    print "iteration "$+itos(i);
endfor;
```

See Also

ftos, stof

k

keep (dataloop)

Purpose

Specifies columns (variables) to be saved to the output data set in a data loop.

Format

```
keep variable_list;
```

Remarks

Commas are optional in *variable_list*.

Retains only the specified variables in the output data set. Any variables referenced must already exist, either as elements of the source data set, or as the result of a previous `make`, `vector`, or **code** statement.

If neither `keep` nor `drop` is used, the output data set will contain all variables from the source data set, as well as any newly defined variables. The effects of multiple `keep` and `drop` statements are cumulative.

Example

```
keep age, pay, sex;
```

See Also

drop (dataloop)

key

Purpose

Returns the ASCII value of the next key available in the keyboard buffer.

Format

y = **key**;

Output

y	scalar, ASCII value of next available key in keyboard buffer.

Remarks

If you are working in terminal mode, **key** does not "see" any keystrokes until ENTER is pressed. The value returned will be zero if no key is available in the buffer or it will equal the ASCII value of the key if one is available. The key is taken from the buffer at this time and the next call to **key** will return the next key.

Example

```
format /rds 1,0;
kk = 0;
do until kk == 113;
   kk = key;
   if kk == 0;
      continue;
   elseif kk == vals(" ");
      print "space \\" kk;
```

```
        elseif kk >= vals("0") and kk <= vals("9");
            print "digit \\" kk chrs(kk);
        else;
            print "\\" kk;
        endif;
    endo;
```

This is an example of a loop that processes keyboard input. This loop will continue until the `q` key (ASCII 113) is pressed.

See Also

vals, chrs, upper, lower, con, cons

keyav

Purpose

Check if keystroke is available.

Format

x = **keyav**;

Output

x	scalar, value of key or 0 if no key is available.

See Also

keyw, key

keyw

Purpose

Waits for and gets a key.

Format

k = **`keyw`**;

Output

k	scalar, ASCII value of the key pressed.

Remarks

If you are working in terminal mode, **GAUSS** will not see any input until you press the ENTER key. **`keyw`** gets the next key from the keyboard buffer. If the keyboard buffer is empty, **`keyw`** waits for a keystroke. For normal keys, **`keyw`** returns the ASCII value of the key.

See Also

key

keyword

Purpose

Begins the definition of a keyword procedure. Keywords are user-defined functions with local or global variables.

Format

```
    keyword name(str);
```

Input

name	literal, name of the keyword. This name will be a global symbol.
str	string, a name to be used inside the keyword to refer to the argument that is passed to the keyword when the keyword is called. This will always be local to the keyword, and cannot be accessed from outside the keyword or from other keywords or procedures.

Remarks

A keyword definition begins with the `keyword` statement and ends with the `endp` statement. See **Procedures and Keywords**, Chapter 1.

Keywords always have 1 string argument and 0 returns. **GAUSS** will take everything past *name*, excluding leading spaces, and pass it as a string argument to the keyword. Inside the keyword, the argument is a local string. The user is responsible to manipulate or parse the string.

An example of a keyword definition is:

```
    keyword add(str);
       local tok,sum;
       sum = 0;
       do until str $== "";
          { tok, str } = token(str);
          sum = sum + stof(tok);
```

```
        endo;
        print "Sum is: " sum;
    endp;
```

To use this keyword, type:

```
    add 1 2 3 4 5;
```

This keyword will respond by printing:

```
        Sum is: 15
```

See Also

proc, local, endp

lag (dataloop)

Purpose

Lags variables a specified number of periods.

Format

```
lag nv1 = var1:p1 [[nv2 = var2:p2...]];
```

Input

var	name of the variable to lag.
p	scalar constant, number of periods to lag.

Output

nv	name of the new lagged variable.

Remarks

You can specify any number of variables to lag. Each variable can be lagged a different number of periods. Both positive and negative lags are allowed.

Lagging is executed before any other transformations. If the new variable name is different from that of the variable to lag, the new variable is first created and appended to a temporary data set. This temporary data set becomes the input data set for the dataloop, and is then automatically deleted.

lag1

Purpose

Lags a matrix by one time period for time series analysis.

Format

y = **`lag1`**(*x*);

Input

x	Nx1 column vector or NxK matrix.

Output

y	NxK matrix, *x* lagged 1 period.

Remarks

`lag1` lags *x* by one time period, so the first observations of *y* are missing. **`lag1`** assumes that each column of the input is a different time series and that each row is an observation. Therefore if a 1xK row vector is passed to **`lag1`**, it will return a 1xK of missing values.

Example

```
    y = { 1.2,
          3.4,
          2.5,
          4.1,
          2.8 };
    y_lag = lag1(y);
    print y_lag;
```

will return:

```
                 .
        1.2000000
        3.4000000
        2.5000000
        4.1000000
```

Source

`lag.src`

See Also

lagn, ismiss, packr

lagn

Purpose

Lags a matrix a specified number of time periods for time series analysis.

Format

y = **lagn**(*x*, *t*);

Input

x	NxK matrix.
t	scalar or Px1 vector, number of time periods.

Output

y	NxK matrix, *x* lagged *t* periods.

Example

```
nlags = 2;
x = { 1.4, 2.7, 3.1, 2.9, 3.2, 2.5, 2.8 };
x_lag2 = lagn(x, nlags);
```

will assign *x_lag2* to equal:

```
       .
       .
     1.4
     2.7
     3.1
     2.9
     3.2
```

Creating multiple lags

If the number of time periods to lag is a Px1 column vector, then the output matrix with be an NxP matrix where each column contains one of the lags. For example, changing the `nlags` variable from the example above to be a 3x1 column vector like this:

```
    nlags = { 1, 2, 3 };
    x = { 1.4, 2.7, 3.1, 2.9, 3.2, 2.5, 2.8 };
    lag_mat = lagn(x, nlags);
```

will assign `lag_mat` to equal:

```
             .        .        .
           1.4        .        .
           2.7      1.4        .
           3.1      2.7      1.4
           2.9      3.1      2.7
           3.2      2.9      3.1
           2.5      3.2      2.9
```

Remarks

If `t` is positive, **`lagn`** lags `x` back `t` time periods, so the first `t` observations of `y` are missing. If `t` is negative, **`lagn`** lags `x` forward `t` time periods, so the last `t` observations of `y` are missing.

Source

`lag.src`

See Also

lag1

lapeighb

Purpose

Computes eigenvalues only of a real symmetric or complex Hermitian matrix selected by bounds.

Format

```
ve = lapeighb(x, vl, vu, abstol);
```

Input

x	NxN matrix, real symmetric or complex Hermitian.
vl	scalar, lower bound of the interval to be searched for eigenvalues.
vu	scalar, upper bound of the interval to be searched for eigenvalues; *vu* must be greater than *vl*.
abstol	scalar, the absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*, *b*] of width less than or equal to: `abstol + EPS*max(|a|, |b|)` where EPS is machine precision. If *abstol* is less than or equal to zero, then *EPS**||*T*|| will be used in its place, where *T* is the tridiagonal matrix obtained by reducing the input matrix to tridiagonal form.

Output

ve	Mx1 vector, eigenvalues, where M is the number of eigenvalues on the half open interval [*vl*, *vu*]. If no eigenvalues are found then *ve* is a scalar missing value.

Remarks

lapeighb computes eigenvalues only which are found on on the half open interval [*vl*, *vu*]. To find eigenvalues within a specified range of indices see **lapeighi**. For eigenvectors see **lapeighvi**, or **lapeighvb**. **lapeighb** is based on the LAPACK drivers DSYEVX and ZHEEVX. Further documentation of these functions may be found in the LAPACK User's Guide.

Example

```
    x = { 5 2 1,
          2 6 2,
          1 2 9 };

    vl = 5;
    vu = 10;
    ve = lapeighb(x,vl,vu,1e-15);
    print ve;
```

The code above returns:

```
    6.0000
```

See Also

lapeighvi, lapeighvb

lapeighi

Purpose

Computes eigenvalues only of a real symmetric or complex Hermitian matrix selected by index.

Format

ve = **lapeighi**(*x*, *il*, *iu*, *abstol*);

Input

x	NxN matrix, real symmetric or complex Hermitian.
il	scalar, index of the smallest desired eigenvalue ranking them from smallest to largest.
iu	scalar, index of the largest desired eigenvalue, *iu* must be greater than *il*.
abstol	scalar, the absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*, *b*] of width less than or equal to *abstol* + EPS*max(|a|, |b|), where EPS is machine precision. If *abstol* is less than or equal to zero, then EPS*||T|| will be used in its place, where *T* is the tridiagonal matrix obtained by reducing the input matrix to tridiagonal form.

Output

ve	(*iu*-*il*+1)x1 vector, eigenvalues.

Remarks

lapeighi computes *iu*-*il*+1 eigenvalues only given a range of indices, i.e., the *i*th to *j*th eigenvalues, ranking them from smallest to largest. To find eigenvalues within a specified range see **lapeighxb**. For eigenvectors see **lapeighvi**, or **lapeighvb**. **lapeighi** is based on the LAPACK drivers DSYEVX and ZHEEVX. Further documentation of these functions may be found in the LAPACK User's Guide.

Example

```
x = { 5 2 1,
      2 6 2,
      1 2 9 };

il = 2;
iu = 3;
ve = lapeighi(x,il,iu,0);
print ve;
```

The code above calculates the second and third eigenvalues and returns:

```
6.0000
10.6056
```

To calculate the first, second and third eigenvalues, reusing the same *x* from above:

```
ve = lapeighi(x,1,3,0);
print ve;
```

The output from this code is:

```
     3.3944
     6.0000
    10.6056
```

See Also

lapeighb, lapeighvi, lapeighvb

lapeighvb

Purpose

Computes eigenvalues and eigenvectors of a real symmetric or complex Hermitian matrix selected by bounds.

Format

{ *ve*, *va* } = **lapeighvb**(*x*, *vl*, *vu*, *abstol*);

Input

x	NxN matrix, real symmetric or complex Hermitian.
vl	scalar, lower bound of the interval to be searched for eigenvalues.
vu	scalar, upper bound of the interval to be searched for eigenvalues; *vu* must be greater than *vl*.
abstol	scalar, the absolute error tolerance for the eigenvalues. An

	approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*, *b*] of width less than or equal to *abstol* + EPS*max(|a|, |b|), where EPS is machine precision. If *abstol* is less than or equal to zero, then EPS*||T|| will be used in its place, where *T* is the tridiagonal matrix obtained by reducing the input matrix to tridiagonal form.

Output

ve	Mx1 vector, eigenvalues, where M is the number of eigenvalues on the half open interval [*vl*, *vu*]. If no eigenvalues are found then s is a scalar missing value.
va	NxM matrix, eigenvectors.

Remarks

lapeighvb computes eigenvalues and eigenvectors which are found on the half open interval [*vl*, *vu*]. **lapeighvb** is based on the LAPACK drivers DSYEVX and ZHEEVX. Further documentation of these functions may be found in the LAPACK User's Guide.

Example

```
x = { 5 2 1,
      2 6 2,
      1 2 9 };

vl = 5;
vu = 10;
```

```
    { ve, va } = lapeighvb(x,vl,vu,0);

    print "Eigenvalues" ve;
    print "Eigenvectors = " va;
```

```
    Eigenvalues =    6.0000
    Eigenvectors =
     -0.5774
     -0.5774
      0.5774
```

If you increase the value of *vu* to 12.

```
    { ve, va } = lapeighvb(x,5,12,0);

    print "Eigenvalues" ve;
    print "Eigenvectors = " va;
```

```
    Eigenvalues
      6.0000
     10.6056
    Eigenvectors =
     -0.5774    0.3197
     -0.5774    0.4908
      0.5774    0.8105
```

lapeighvi

Purpose

Computes selected eigenvalues and eigenvectors of a real symmetric or complex Hermitian matrix.

Format

```
{ ve, va } = lapeighvi(x, il, iu, abstol);
```

Input

x	NxN matrix, real symmetric or complex Hermitian.
il	scalar, index of the smallest desired eigenvalue ranking them from smallest to largest.
iu	scalar, index of the largest desired eigenvalue, *iu* must be greater than *il*.
abstol	scalar, the absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*, *b*] of width less than or equal to *abstol* + EPS*max(|a|, |b|), where EPS is machine precision. If *abstol* is less than or equal to zero, then EPS*||T|| will be used in its place, where *T* is the tridiagonal matrix obtained by reducing the input matrix to tridiagonal form.

Output

ve	(*iu*- *il*+1)x1 vector, eigenvalues.
va	Nx(*iu*- *il*+1) matrix, eigenvectors.

Remarks

lapeighvi computes *iu*-*il*+1 eigenvalues and eigenvectors given a range of

indices, i.e., the *i*th to *j*th eigenvalues, ranking them from smallest to largest. To find eigenvalues and eigenvectors within a specified range see **`lapeighvb`**.
`lapeighvi` is based on the LAPACK drivers DSYEVX and ZHEEVX. Further documentation of these functions may be found in the LAPACK User's Guide.

Example

```
x = { 5 2 1,
      2 6 2,
      1 2 9 };

il = 2;
iu = 3;
{ ve,va } = lapeighvi(x,il,iu,0);
print "ve = " ve;
print "va = " va;
```

```
ve =
6.0000
10.6056

va =
-0.5774   0.3197
-0.5774   0.4908
 0.5774   0.8105
```

See Also

lapeighvb, lapeighb

lapgeig

Purpose

Computes generalized eigenvalues for a pair of real or complex general matrices.

Format

{ *va1*, *va2* } = **lapgeig**(*A*, *B*);

Input

A	NxN matrix, real or complex general matrix.
B	NxN matrix, real or complex general matrix.

Output

va1	Nx1 vector, numerator of eigenvalues.
va2	Nx1 vector, denominator of eigenvalues.

Remarks

va1 and *va2* are the vectors of the numerators and denominators respectively of the eigenvalues of the solution of the generalized symmetric eigenproblem of the form $Aw = eBw$ where A and B are real or complex general matrices and *w* = *va1*./*va2*. The generalized eigenvalues are not computed directly because some elements of *va2* may be zero, i.e., the eigenvalues may be infinite. This procedure calls the LAPACK routines DGGEV and ZGGEV.

See Also

lapgeig, lapgeigh

lapgeigh

Purpose

Computes generalized eigenvalues for a pair of real symmetric or Hermitian matrices.

Format

```
ve = lapgeigh(A,  B);
```

Input

A	NxN matrix, real or complex symmetric or Hermitian matrix.
B	NxN matrix, real or complex positive definite symmetric or Hermitian matrix.

Output

ve	Nx1 vector, eigenvalues.

Remarks

ve is the vector of eigenvalues of the solution of the generalized symmetric eigenproblem of the form $Ax = \lambda Bx$.

Example

```
A = { 3 4 5,
      2 5 2,
      3 2 4 };

B = { 4 2 2,
      2 6 1,
      2 1 8 };

ve = lapgeigh(A,B);
print ve;
```

The code above returns:

```
0.1219
0.6787
0.9494
```

This procedure calls the LAPACK routines DSYGV and ZHEGV.

See Also

lapgeig, lapgeighv

lapgeighv

Purpose

Computes generalized eigenvalues and eigenvectors for a pair of real symmetric or Hermitian matrices.

Format

```
{ ve, va } = lapgeighv(A, B);
```

Input

A	NxN matrix, real or complex symmetric or Hermitian matrix.
B	NxN matrix, real or complex positive definite symmetric or Hermitian matrix.

Output

ve	Nx1 vector, eigenvalues.
va	NxN matrix, eigenvectors.

Remarks

ve and *va* are the eigenvalues and eigenvectors of the solution of the generalized symmetric eigenproblem of the form $Ax = \lambda B$. Equivalently, *va* diagonalizes $U'^{-1} A*U^{-1}$ in the following way

```
va*U'-1 A*Y-1va' = ve
```

where $B = U'U$. This procedure calls the LAPACK routines DSYGV and ZHEGV.

Example

```
A = { 3 4 5,
      2 5 2,
      3 2 4 };
```

```
B = { 4 2 2,
      2 6 1,
      2 1 8 };

{ ve, va } = lapgeighv(A,B);

print ve;
```

```
-0.0425
 0.5082
 0.8694
```

```
print va;
```

```
 0.3575 -0.0996 0.9286
-0.2594  0.9446 0.2012
-0.8972 -0.3128 0.3118
```

See Also

lapgeig, lapgeigh

lapgeigv

Purpose

Computes generalized eigenvalues, left eigenvectors, and right eigenvectors for a pair of real or complex general matrices.

Format

{ *va1*, *va2*, *lve*, *rve* } = **lapgeigv**(*A*, *B*);

Input

A	NxN matrix, real or complex general matrix.
B	NxN matrix, real or complex general matrix.

Output

va1	Nx1 vector, numerator of eigenvalues.
va2	Nx1 vector, denominator of eigenvalues.
lve	NxN left eigenvectors.
rve	NxN right eigenvectors.

Remarks

va1 and *va2* are the vectors of the numerators and denominators respectively of the eigenvalues of the solution of the generalized symmetric eigenproblem of the form $Aw = \lambda Bw$ where *A* and *B* are real or complex general matrices and *w* = *va1./va2*. The generalized eigenvalues are not computed directly because some elements of *va2* may be zero, i.e., the eigenvalues may be infinite.

The left and right eigenvectors diagonalize $U'^{-1}*A*U^{-1}$ where $B = U'*U$, that is,

```
     lve*U'^-1A*U*lve' = w
```

and

```
     rve'U'^-1*A*U^-1*rve = w
```

This procedure calls the LAPACK routines DGGEV and ZGGEV.

See Also

lapgeig, lapgeigh

lapgsvdcst

Purpose

Compute the generalized singular value decomposition of a pair of real or complex general matrices.

Format

```
{ C, S, R, U, V, Q } = lapgsvdcst(A, B);
```

Input

A	MxN matrix.
B	PxN matrix.

Output

C	Lx1 vector, singular values for *A*.
S	Lx1 vector, singular values for *B*.
R	(K+L)x(K+L) upper triangular matrix.
U	MxM matrix, orthogonal transformation matrix.
V	PxP matrix, orthogonal transformation matrix.

Q	NxN matrix, orthogonal transformation matrix.

Remarks

(1) The generalized singular value decomposition of *A* and *B* is

```
     U'*A*Q = D1*Z
```

```
     V'*B*Q = D2*Z
```

where *U*, *V*, and *Q* are orthogonal matrices (see **`lapgsvdcst`** and **`lapgsvdst`**). Letting K + L = the rank of *A*|*B* then *R* is a (K+L)x(K+L) upper triangular matrix, *D1* and *D2* are Mx(K+L) and Px(K+L) matrices with entries on the diagonal, *Z* = [*0 R*], and if M-K-L >= 0

```
                          K  L
     D1 =            K   [ I  0 ]
                     L   [ 0  C ]
          M  -  K  - L   [ 0  0 ]
```

```
                      K  L
     D2 =         P  [ 0  S ]
          P  -  L    [ 0  0 ]
```

```
                     N-K-L    K     L
     [ 0  R ]  = K [    0     R11   R12 ]
                 L [    0      0    R22 ]
```

or if M-K-L < 0

```
                      K   M-K   K+L-M
     D1 =    K   [ I    0      0    ]
           M-K   [ 0    0      0    ]
```

```
                        N-K-L  K    M-K  K+L-M
                    K [   0   R11   R12   R13  ]
  [ 0 R ] =       M-K [   0    0    R22   R23  ]
                K+L-M [   0    0     0    R33  ]
```

(2) Form the matrix

$$X = Q \begin{bmatrix} I & 0 \\ 0 & R^{-1} \end{bmatrix}$$

then

$$A = U'^{-1}E_1X$$

$$B = V'^{-1}E_2X^{-1}$$

where

```
  E1 = [ 0  D1 ]

  E2 = [ 0  D2 ]
```

(3) The generalized singular value decomposition of A and B implicitly produces the singular value decomposition of AB^{-1}:

$$AB^{-1} = UD_1D_2^{-1}V'$$

This procedure calls the LAPACK routines DGGSVD and ZGGSVD.

See Also

lapgsvds, lapgsvdst

lapgsvds

Purpose

Compute the generalized singular value decomposition of a pair of real or complex general matrices.

Format

{ *C*, *S*, *R* } = **`lapgsvds`**(*A*, *B*);

Input

A	MxN real or complex matrix.
B	PxN real or complex matrix.

Output

C	Lx1 vector, singular values for *A*.
S	Lx1 vector, singular values for *B*.
R	(K+L)x(K+L) upper triangular matrix.

Remarks

(1) The generalized singular value decomposition of *A* and *B* is

$$U'AQ = D_1 Z$$

$$V'BQ = D_2 Z$$

where *U*, *V*, and *Q* are orthogonal matrices (see **`lapgsvdcst`** and **`lapgsvdst`**). Letting K+L = the rank of *A*|*B* then *R* is a (K+L)x(K+L) upper triangular matrix, D_1 and D_2 are Mx(K+L) and Px(K+L) matrices with entries on the diagonal, *Z* = [*0R*], and if M-K-L >= 0

```
                      K L
D1 =             K  [ I 0 ]
                 L  [ 0 C ]
         M - K - L  [ 0 0 ]
```

```
                  K L
D2 =         P  [ 0 S ]
         P - L  [ 0 0 ]
```

```
                  N-K-L   K    L
[ 0 R ] = K [      0     R11  R12 ]
          L [      0      0   R22 ]
```

or if M-K-L < 0

```
                K  M-K  K+L-M
D1 =     K  [ I    0      0   ]
       M-K  [ 0    0      0   ]
```

```
                      N-K-L  K    M-K  K+L-M
                  K [   0   R11   R12   R13   ]
[ 0 R ] =       M-K [   0    0    R22   R23   ]
              K+L-M [   0    0     0    R33   ]
```

(2) Form the matrix

```
X = Q [ I 0     ]
      [ 0 R^-1  ]
```

then

$$A = U'^{-1}E_1X$$

$$B = V'^{-1}E_2X^{-1}$$

where

```
E1 = [ 0  D1 ]

E2 = [ 0  D2 ]
```

(3) The generalized singular value decomposition of *A* and *B* implicitly produces the singular value decomposition of AB^{-1}:

$$AB^{-1} = UD_1D_2^{-1}V'$$

This procedure calls the LAPACK routines DGGSVD and ZGGSVD.

See Also

lapgsvdcst, lapgsvdst

lapgsvdst

Purpose

Compute the generalized singular value decomposition of a pair of real or complex general matrices.

Format

{ *D1*, *D2*, *Z*, *U*, *V*, *Q* } = **lapgsvdst**(*A*, *B*);

Input

A	MxN matrix.
B	PxN matrix.

Output

D1	Mx(K+L) matrix, with singular values for *A* on diagonal.
D2	Px(K+L) matrix, with singular values for *B* on diagonal.
Z	(K+L)xN matrix, partitioned matrix composed of a zero matrix and upper triangular matrix.
U	MxM matrix, orthogonal transformation matrix.
V	PxP matrix, orthogonal transformation matrix.
Q	NxN matrix, orthogonal transformation matrix.

Remarks

(1) The generalized singular value decomposition of *A* and *B* is

$$U'AQ = D_1Z$$

$$V'BQ = D_2Z$$

where *U*, *V*, and *Q* are orthogonal matrices (see **`lapgsvdcst`** and **`lapgsvdst)`**. Letting K+L = the rank of *A*|*B* then *R* is a (K+L)x(K+L) upper triangular matrix, *D1* and *D2* are Mx(K+L) and Px(K+L) matrices with entries on the diagonal, *Z* = [*0R*], and if M-K-L >= 0

$$
D1 = \begin{array}{c|ccc}
 & K & L & \\
K & [\ I & 0\ ] & \\
L & [\ 0 & C\ ] & \\
M-K-L & [\ 0 & 0\ ] &
\end{array}
$$

$$
D2 = \begin{array}{c|cc}
 & K & L \\
P & [\ 0 & S\ ] \\
P-L & [\ 0 & 0\ ]
\end{array}
$$

$$
[\ 0\ R\ ] = \begin{array}{c|ccc}
 & N-K-L & K & L \\
K & [\ 0 & R11 & R12\ ] \\
L & [\ 0 & 0 & R22\ ]
\end{array}
$$

or if M-K-L < 0

$$
D1 = \begin{array}{c|ccc}
 & K & M-K & K+L-M \\
K & [\ I & 0 & 0\ ] \\
M-K & [\ 0 & 0 & 0\ ]
\end{array}
$$

$$
[\ 0\ R\ ] = \begin{array}{c|cccc}
 & N-K-L & K & M-K & K+L-M \\
K & [\ 0 & R11 & R12 & R13\ ] \\
M-K & [\ 0 & 0 & R22 & R23\ ] \\
K+L-M & [\ 0 & 0 & 0 & R33\ ]
\end{array}
$$

(2) Form the matrix

$$
X = Q \begin{bmatrix} I & 0 \\ 0 & R^{-1} \end{bmatrix}
$$

then

$$
A = U'^{-1}E_1X
$$

$$
B = V'^{-1}E_2X^{-1}
$$

where

```
    E1 = [ 0   D1 ]

    E2 = [ 0   D2 ]
```

(3) The generalized singular value decomposition of A and B implicitly produces the singular value decomposition of AB^{-1}:

```
    AB⁻¹ = UD₁D₂⁻¹V'
```

$$AB^{-1} = UD_1D_2^{-1}V'$$

This procedure calls the LAPACK routines DGGSVD and ZGGSVD.

See Also

lapgsvds, lapgsvdcst

lapgschur

Purpose

Compute the generalized Schur form of a pair of real or complex general matrices.

Format

{ *sa*, *sb*, *Q*, *z* } = **lapgschur**(*A*, *B*);

Input

A	NxN matrix, real or complex general matrix.
B	NxN matrix, real or complex general matrix.

Output

sa	NxN matrix, Schur form of *A*.
sb	NxN matrix, Schur form of *B*.
q	NxN matrix, left Schur vectors.
z	NxN matrix, right Schur vectors.

Remarks

The pair of matrices *A* and *B* are in generalized real Schur form when *B* is upper triangular with non-negative diagonal, and *A* is block upper triangular with 1x1 and 2x2 blocks. The 1x1 blocks correspond to real generalized eigenvalues and the 2x2 blocks to pairs of complex conjugate eigenvalues. The real generalized eigenvalues can be computed by dividing the diagonal element of *sa* by the corresponding diagonal element of *sb*. The complex generalized eigenvalues are computed by first constructing two complex conjugate numbers from 2x2 block where the real parts are on the diagonal of the block and the imaginary part on the off-diagonal. The eigenvalues are then computed by dividing the two complex conjugate values by their corresponding diagonal elements of *sb*. The generalized Schur vectors *q* and *z* are orthogonal matrices that reduce *A* and *B* to Schur form:

```
     sa = q'Az
```

```
     sb = q'Bz
```

This procedure calls the LAPACK routines DGGES and ZGGES.

Source

`lapschur.src`

lapsvdcusv

Purpose

Computes the singular value decomposition of a real or complex rectangular matrix, returns compact *U* and *v*.

Format

{ *u*, *s*, *v* } = **lapsvdcusv**(*x*);

Input

x	MxN matrix, real or complex rectangular matrix.

Output

u	Mxmin(M,N) matrix, left singular vectors.
s	min(M,N)xN matrix, singular values.
v	NxN matrix, right singular values.

Remarks

lapsvdcusv computes the singular value decomposition of a real or complex rectangular matrix. The SVD is

```
    x = usv'
```

where *v* is the matrix of right singular vectors. **lapsvdcusv** is based on the LAPACK drivers DGESVD and ZGESVD. Further documentation of these functions may be found in the LAPACK User's Guide.

Example

```
x = { 2.143 4.345 6.124,
      1.244 5.124 3.412,
      0.235 5.657 8.214 };

{ u,s,v } = lapsvdusv(x);
```

```
print u;
```

```
 -0.55531277  0.049048431 0.83019394
 -0.43090168  0.83684123 -0.33766923
 -0.71130266 -0.54524400 -0.44357356
```

```
print s;
```

```
 13.895868 0.0000000 0.0000000
 0.0000000 2.1893939 0.0000000
 0.0000000 0.0000000 1.4344261
```

```
print v;
```

```
 -0.13624432  -0.62209955 -0.77099263
  0.46497296   0.64704876 -0.60425826
  0.87477862  -0.44081748  0.20110275
```

See Also

lapsvds, lapsvdusv

lapsvds

Purpose

Computes the singular values of a real or complex rectangular matrix

Format

s = **`lapsvds`**(*x*);

Input

x	MxN matrix, real or complex rectangular matrix.

Output

s	min(M,N)x1 vector, singular values.

Remarks

`lapsvds` computes the singular values of a real or complex rectangular matrix. The SVD is

```
    x = usv'
```

where *v* is the matrix of right singular vectors. For the computation of the singular vectors, see **lapsvdcusv** and **lapsvdusv**.

lapsvds is based on the LAPACK drivers DGESVD and ZGESVD. Further documentation of these functions may be found in the LAPACK User's Guide.

Example

```
     x = { 2.143 4.345 6.124,
           1.244 5.124 3.412,
           0.235 5.657 8.214 };

     va = lapsvds(x);
     print va';
```

```
      13.895868 2.1893939 1.4344261
```

```
     xi = { 4+1 3+1 2+2,
            1+2 5+3 2+2,
            1+1 2+1 6+2 };

     ve = lapsvds(xi);
     print ve';
```

```
      10.352877 4.0190557 2.3801546
```

Note the transpose operator (`'`) at the end of the `print` statements. This causes the output of these column vectors to be printed as a row vector.

See Also

lapsvdcusv, lapsvdusv

lapsvdusv

Purpose

Computes the singular value decomposition a real or complex rectangular matrix.

Format

{ *u*, *s*, *v* } = **lapsvdusv**(*x*);

Input

x	MxN matrix, real or complex rectangular matrix.

Output

u	MxM matrix, left singular vectors.
s	MxN matrix, singular values.
v	NxN matrix, right singular values.

Remarks

lapsvdusv computes the singular value decomposition of a real or complex rectangular matrix. The SVD is

```
    x = usv'
```

where *v* is the matrix of right singular vectors. **lapsvdusv** is based on the LAPACK drivers DGESVD and ZGESVD. Further documentation of these functions may be found in the LAPACK User's Guide.

Example

```
x = { 2.143 4.345 6.124,
      1.244 5.124 3.412,
      0.235 5.657 8.214 };

{ u,s,v } = lapsvdusv(x);

print u;
```

```
 -0.5553  0.0490  0.8302
 -0.4309  0.8368 -0.3377
 -0.7113 -0.5452 -0.4436
```

```
print s;
```

```
 13.8959 0.0000 0.0000
 0.0000 2.1894 0.0000
 0.0000 0.0000 1.4344
```

```
print v;
```

```
 -0.1362  0.4650  0.8748
  0.6221  0.6470 -0.4408
 -0.7710 -0.6043  0.2011
```

See Also

lapsvds, lapsvdcusv

let

Purpose

Creates a matrix from a list of numeric or character values. The result is always of type matrix, string, or string array.

Format

```
let x= constant_list;
```

Remarks

Expressions and variable names are not allowed in the `let` command, expressions such as this:

```
let x[2,1] = 3*a b
```

are illegal. To define matrices by combining matrices and expressions, use an expression containing the concatenation operators: **~** and **|**.

Numbers can be entered in scientific notation. The syntax is *dE±n*, where *d* is a number and *n* is an integer (denoting the power of 10):

```
let x = 1e+10 1.1e-4 4.019e+2;
```

Complex numbers can be entered by joining the real and imaginary parts with a sign (+ or -); there should be no spaces between the numbers and the sign. Numbers with no real part can be entered by appending an "i" to the number:

```
let x = 1.2+23 8.56i 3-2.1i -4.2e+6i 1.2e-4-4.5e+3i;
```

If curly braces are used, the `let` is optional.

```
    let x = { 1 2 3, 4 5 6, 7 8 9 };

     x = { 1 2 3, 4 5 6, 7 8 9 };
```

If indices are given, a matrix of that size will be created:

```
    let x[2,2] = 1 2 3 4;
```

```
    x = 1 2
        3 4
```

If indices are not given, a column vector will be created:

```
    let x = 1 2 3 4;
```

```
        1
    x = 2
        3
        4
```

You can create matrices with no elements, i.e., "empty matrices" . Just use a set of empty curly braces:

```
         x = {};
```

Empty matrices are chiefly used as the starting point for building up a matrix, for example in a `do` loop. See **Matrices**, Section 1.0.1, for more information on empty matrices.

Character elements are allowed in a `let` statement:

```
    let x = age pay sex;
    AGE
x = PAY
    SEX
```

Lowercase elements can be created if quotation marks are used. Note that each element must be quoted.

```
let x = "age""pay""sex";
```

```
    age
x = pay
    sex
```

Example

```
let x;
```

assigns *x* to be:

```
x = 0
```

```
let x = { 1 2 3, 4 5 6, 7 8 9 };
```

assigns *x* to be:

```
    1 2 3
x = 3 4 5
    6 7 8
```

```
let x[3,3] = 1 2 3 4 5 6 7 8 9;
```

assigns *x* to be:

```
        1  2  3
x  =  3  4  5
        6  7  8
```

```
let x[3,3] = 1;
```

assigns x to be:

```
        1  1  1
x  =  1  1  1
        1  1  1
```

```
let x[3,3];
```

assigns x to be:

```
        0  0  0
x  =  0  0  0
        0  0  0
```

```
let x = dog cat;
```

assigns x to be:

```
x  =  DOG
        CAT
```

```
let x = "dog""cat";
```

assigns x to be:

```
x  =  dog
        cat
```

```
let string x = { "Median Income", "Country" };
```

assigns *x* to be:

```
    x = Median Income
        Country
```

See Also

con, cons, declare, load

lib

Purpose

Builds and updates library files.

Format

```
lib library file;
lib library -flag;
lib library file -flag1 -flag2;
```

Input

library	literal, name of library.
file	optional literal, name of source file to be updated or added.
flags	optional literal preceded by '-', controls operation of library update. To control handling of path information on source filenames:
	-addpath add paths to entries without paths and

	expand relative paths.
-gausspath	reset all paths using a normal file search.
-leavepath	(default) leave all path information untouched.
-nopath	drop all path information.

To specify a library update or a complete library build:

-update	(default) update the symbol information for the specified file only.
-build	update the symbol information for every library entry by compiling the actual source file.
-delete	delete a file from the library.
-list	list files in a library.

To control the symbol type information placed in the library file:

-strong	(default) use strongly typed symbol entries.
-weak	save no type information. This should only be used to build a library compatible with a previous version of **GAUSS**.

To control location of temporary files for a complete library build:

-tmp	(default) use the directory pointed to by the *tmp_path* configuration variable. If *tmp_path* is not defined, `lib` will look for a

	tmp environment variable.
-disk	use the same directory listed in the lib_ path configuration variable.

Example

Let us suppose that you have a file named myprocs.gss located in your GAUSS src directory. Let us further suppose that you would like to create a new library named mylibrary. You could accomplish that task like this:

```
    lib mylibrary myprocs.gss;
```

Now that this library has been created, you could add other files in the same manner. To add a file named mystats.gss would look like this:

```
    lib mylibrary mystats.gss;
```

This second command will add the file mystats.gss to the mylibrary which was created in the first step above. It will not overwrite or replace the library.

You may print the list of files contained in the library by using the *-list* flag. Entering the command:

```
    lib mylibrary -list;
```

at the GAUSS command line will produce the output similar to:

```
    Listing library:  mylibrary.lcg
        myprocs.gss
        mystats.gs
```

If you add procedures to one of the files in your library, you will need to update the library to reflect these new changes. Continuing with the example from above, if you added some new procedures to the file `mystats.gss`, you could update the `mylibrary` library with the following command:

```
    lib mylibrary mystats.gss -update;
```

Note that, as in the command above, the `-update` flag must be used with a file. To update, or rebuild the references for all files in the library, use the `-build` flag.

```
    lib mylibrary -build;
```

Remarks

The library management functionality offered by the `lib` command can also be accomplished interactively with windows and buttons, using the Library Tool in the user interface. See **The Library Tool**, Chapter 1, for more information on using the Library Tool.

The flags can be shortened to one or two letters, as long as they remain unique-for example, *-b* to *-build* a library, *-li* to list files in a library.

If the filenames include a full path, the compilation process is faster because no unnecessary directory searching is needed during the autoloading process. The default path handling adds a path to each file listed in the library and also expands any relative paths so the system will work from any drive or subdirectory.

When a path is added to a filename containing no path information, the file is searched for on the current directory and then on each subdirectory listed in *src_path*. The first path encountered that contains the file is added to the filename in the library entry.

See Also

library

library

Purpose

Sets up the list of active libraries.

Format

```
library lib1 [[,lib2,lib3,lib4...]];
library;
```

Example

Example 1

If no arguments are given, the list of current libraries will be printed out. For example:

```
library;
```

will produce output similar to:

```
Library path: C:\gauss13\gaussplot\lib
              C:\gauss13\lib

Libraries:    C:\gauss13\lib\user.lcg
              C:\gauss13\lib\gauss.lcg
```

Example 2

Load multiple libraries by passing a comma-separated list of library names.

```
library pgraph, cmlmt;
```

After executing the code above, entering the `library` command without any arguments will produce output similar to this:

```
Library path: C:\gauss13\gaussplot\lib
              C:\gauss13\lib

Libraries:    C:\gauss13\lib\user.lcg
              C:\gauss13\lib\pgraph.lcg
              C:\gauss13\lib\cmlmt.lcg
              C:\gauss13\lib\gauss.lcg
```

The output from the `library` command above is printed in the order in which GAUSS will search. For this particular example, GAUSS will first search the `user` library, then the `pgraph` library, followed by the `cmlmt` library and finally the `gauss` library.

Example 3

Loading a library or list of libraries with the `library` command will also close any open libraries other than `user` and `gauss` which are always loaded.

Continuing with the last example in which the `user`, `pgraph`, `cmlmt` and `gauss` libraries were open, executing the command:

```
library pgraph, tsmt;
```

would open `tsmt`, while closing `pgraph` and `cmlmt`.

Remarks

For more information about the library system, see **Libraries**, Chapter 1 .

The required extension for library files is `.lcg`.

Library searching

If a list of library names is given, they will be the new set of active libraries. The two default libraries are `user.lcg` and `gauss.lcg`. Unless otherwise specified, `user.lcg` will be searched first and `gauss.lcg` will be searched last. Any other user-specified libraries will be searched after `user.lcg` in the order they were entered in the `library` statement.

If the statement:

```
    y = myProc(x);
```

is encountered in a program, **`myProc`** will be searched for in the active libraries. If it is found, it will be compiled. If it cannot be found in a library, the deletion state determines how it is handled:

`autodelete` *`on`*	search for myproc.g
`autodelete` *`off`*	return Undefined symbol error message

If **`myProc`** calls **`myRegress`** and **`myRegress`** calls **`myUtil`** and they are all in separate files, they will all be found by the autoloader.

The source browser and the help facility will search for **`myProc`** in exactly the same sequence as the autoloader.

Library file contents

Library files are simple ASCII files that you can create with a text editor. Here is an example:

```
    /*
    ** This is a GAUSS library file.
    */

    eig.src
```

```
        eig     : proc
        eigsym  : proc
        _eigerr : matrix
    svd.src
        cond    : proc
        pinv    : proc
        rank    : proc
        svd     : proc
        _svdtol : matrix
```

The lines not indented are the file names. The lines that are indented are the symbols defined in that file. As you can see, a **GAUSS** library is a dictionary of files and the global symbols they contain.

Any line beginning with `/*`, `**`, or `*/` is considered a comment. Currently, `//` comments are not supported in library files. Blank lines are okay.

To make the autoloading process more efficient, you can put the full pathname for each file in the library:

```
    /gauss/src/eig.src
        eig      : proc
        eigsym   : proc
        _eigerr  : matrix
    /gauss/src/svd.src
        cond     : proc
        pinv     : proc
        rank     : proc
        svd      : proc
        _svdtol  : matrix
```

Here's a debugging hint. If your program is acting strange and you suspect it is autoloading the wrong copy of a procedure, use the Library Tool on the Source Page, or the `CTRL+F1` hotkey to locate the suspected function. It will use the same search path that the autoloader uses.

See Also

declare, external, lib, proc

#lineson, #linesoff

Purpose

The #lineson command causes **GAUSS** to embed line number and file name records in a program for the purpose of reporting the location where an error occurs. The #linesoff command causes **GAUSS** to stop embedding line and file records in a program.

Format

```
#lineson
#linesoff
```

Remarks

In the "lines on" mode, **GAUSS** keeps track of line numbers and file names and reports the location of an error when an execution time error occurs. In the "lines off" mode, **GAUSS** does not keep track of lines and files at execution time. During the compile phase, line numbers and file names will always be given when errors occur in a program stored in a disk file.

It is easier to debug a program when the locations of errors are reported, but this slows down execution. In programs with several scalar operations, the time spent tracking line numbers and file names is most significant.

These commands have no effect on interactive programs (that is, those typed in the window and run from the command line), since there are no line numbers in such programs.

Line number tracking can be turned on and off through the user interface, but the `#lineson` and `#linesoff` commands will override that.

The line numbers and file names given at run-time will reflect the last record encountered in the code. If you have a mixture of procedures that were compiled without line and file records and procedures that were compiled with line and file records, use the `trace` command to locate exactly where the error occurs.

The Currently active call error message will always be correct. If it states that it was executing procedure **`xyz`** at line number nnn in file ABC and *`xyz`* has no line *`nnn`* or is not in file ABC, you know that it just did not encounter any line or file records in *`xyz`* before it crashed.

When using `#include`'d files, the line number and file name will be correct for the file the error was in within the limits stated above.

See Also

trace

linsolve

Purpose

Solves $Ax = b$ using the inverse function.

Format

x = **`linsolve`**(b, A);

Input

b	NxK matrix.
A	NxN matrix.

Output

x	NxK matrix, the linear solution of *b*/*A* for each column in *b*.

Remarks

linsolve solves for *x* by computing **inv**(*A*)**b*. If *A* is square and *b* contains more than 1 column, it is much faster to use **linsolve** than the **/** operator. However, while faster, there is some sacrifice in accuracy.

A test shows **linsolve** to be acccurate to within approximately 1.2e-11, while the slash operator '**/**' is accurate to within approximately 4e-13. However, the accuracy sacrifice can be much greater for poorly conditioned matrices.

Example

```
b = { 2, 3, 4 };
A = { 10 2 3, 6 14 2, 1 1 9 };
x = linsolve(b,A);
print x
```

```

 0.04586330
 0.13399281
 0.42446043
```

See Also

qrsol, qrtsol, solpd, cholsol

listwise (dataloop)

Purpose

Controls listwise deletion of missing values.

Format

```
listwise [[read]] [[write]];
```

Remarks

If **`read`** is specified, the deletion of all rows containing missing values happens immediately after reading the input file and before any transformations. If **`write`** is specified, the deletion of missing values happens after any transformations and just before writing to the output file. If no **`listwise`** statement is present, rows with missing values are not deleted.

The default is **`read`**.

ln

Purpose

Computes the natural log of all elements of *x*.

Format

y = **ln**(x);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing the natural log values of the elements of x.

Remarks

ln is defined for $x \neq 0$.

If x is negative, complex results are returned.

You can turn the generation of complex numbers for negative inputs on or off in the **GAUSS** configuration file, and with the **sysstate** function, case 8. If you turn it off, **ln** will generate an error for negative inputs.

If x is already complex, the complex number state doesn't matter; **ln** will compute a complex result.

x can be any expression that returns a matrix.

Example

```
    y = ln(16);
    y = 2.7725887
```

See Also

log

lncdfbvn

Purpose

Computes natural log of bivariate Normal cumulative distribution function.

Format

y = **lncdfbvn**(*x1*, *x2*, *r*);

Input

x1	NxK matrix, abscissae.
x2	LxM matrix, abscissae.
r	PxQ matrix, correlations.

Output

y	max(N,L,P) x max(K,M,Q) matrix: `ln Pr(X < x1, X < x2|r)`

Remarks

x1, *x2*, and *r* must be ExE conformable.

Source

`lncdfn.src`

See Also

cdfbvn, lncdfmvn

lncdfbvn2

Purpose

Returns natural log of standardized bivariate Normal cumulative distribution function of a bounded rectangle.

Format

y = **lncdfbvn2**(*h*, *dh*, *k*, *dk*, *r*);

Input

h	Nx1 vector, upper limits of integration for variable 1.
dh	Nx1 vector, increments for variable 1.
k	Nx1 vector, upper limits of integration for variable 2.
dk	Nx1 vector, increments for variable 2.

r	Nx1 vector, correlation coefficients between the two variables.

Output

y	Nx1 vector, the log of the integral from *h*, *k* to *h*+*dh*, *k*+*dk* of the standardized bivariate Normal distribution.

Remarks

Scalar input arguments are okay; they will be expanded to Nx1 vectors.

lncdfbvn2 will abort if the computed integral is negative.

lncdfbvn2 computes an error estimate for each set of inputs-the real integral is **exp**(*y*)±*err*. The size of the error depends on the input arguments. If **trap 2** is set, a warning message is displayed when *err* >= **exp**(*y*)/100.

For an estimate of the actual error, see **cdfBvn2e**.

Example

Example 1

```
    lncdfbvn2(1,1,1,1,0.5);
```

produces:

```
         -3.2180110258198771e+000
```

Example 2

```
    trap 0,2;
    lncdfbvn2(1,1e-15,1,1e-15,0.5);
```

produces:

```
         -7.1171016046360151e+001
```

Example 3

```
    trap 2,2;
    lncdfbvn2(1,-1e-45,1,1e-45,0.5);
```

produces:

```
    WARNING: Dubious accuracy from lncdfbvn2:
     0.000e+000 +/- 2.8e-060
     -INF
```

See Also

cdfbvn2, cdfbvn2e

lncdfmvn

Purpose

Computes natural log of multivariate Normal cumulative distribution function.

Format

y = **lncdfmvn**(x, r);

Input

x	KxL matrix, abscissae.
r	KxK matrix, correlation matrix.

Output

y	Lx1 vector,

```
    ln Pr(X < x|r)
```

Remarks

You can pass more than one set of abscissae at a time; each column of *x* is treated separately.

Source

`lncdfn.src`

See Also

cdfmvn, lncdfbvn

lncdfn

Purpose

Computes natural log of Normal cumulative distribution function.

Format

y = **lncdfn**(x);

Input

x	NxK matrix or N-dimensional array, abscissae.

Output

y	NxK matrix or N-dimensional array, **ln** Pr(X < x)

Source

```
lncdfn.src
```

lncdfn2

Purpose

Computes natural log of interval of Normal cumulative distribution function.

Format

y = **lncdfn2**(x, r);

Input

x	MxN matrix, abscissae.
r	KxL matrix, ExE conformable with *x*, intervals.

Output

y	max(M,K)xmax(N,L) matrix, the log of the integral from *x* to *x*+*dx* of the Normal distribution, i.e., `ln Pr(x < X < x+dx)`

Remarks

The relative error is:

$\lvert x\rvert < 1$	and	$dx < 1$	±1e-14
$1 < \lvert x\rvert < 37$	and	$\lvert dx\rvert < 1/\lvert x\rvert$	±1e-13
$\min(x,x+dx) > -37$	and	$y > -690$	±1e-11 or better

A relative error of ±1e-14 implies that the answer is accurate to better than ±1 in the 14th digit after the decimal point.

Example

```
    print lncdfN2(-10,29);
```

```
    -7.6198530241605269e-24
```

```
print lncdfN2(0,1);
```

```
-1.0748623268620716e+00
```

```
print lncdfN2(5,1);
```

```
 -1.5068446096529453e+01
```

Source

`lncdfn.src`

See Also

cdfn2

lncdfnc

Purpose

Computes natural log of complement of Normal cumulative distribution function.

Format

y = **lncdfnc**(x);

Input

x	NxK matrix, abscissae.

Output

y	NxK matrix, `ln (1 - Pr(X < x))`

Source

`lncdfn.src`

lnfact

Purpose

Computes the natural log of the factorial function and can be used to compute log gamma.

Format

y = **`lnfact`**(*x*);

Input

x	NxK matrix or N-dimensional array, all elements must be positive.

Output

y	NxK matrix containing the natural log of the factorial of each of the elements in *x*.

Remarks

For integer x, this is (approximately) **ln**(x!). However, the computation is done using a formula, and the function is defined for noninteger x.

In most formulae in which the factorial operator appears, it is possible to avoid computing the factorial directly, and to use **lnfact** instead. The advantage of this is that **lnfact** does not have the overflow problems that the factorial (*!*) operator has.

For $x > 1$, this function has at least 6 digit accuracy, for $x > 4$ it has at least 9 digit accuracy, and for $x > 10$ it has at least 12 digit accuracy. For $0 < x < 1$, accuracy is not known completely but is probably at least 6 digits.

Sometimes log gamma is required instead of log factorial. These functions are related by:

```
lngamma(x) = lnfact(x-1);
```

Example

```
let x = 100 500 1000;
y = lnfact(x);
```

```
       363.73938
   y = 2611.3305
       5912.1282
```

Source

lnfact.src

See Also

gamma

Technical Notes

For $x > 1$, Stirling's formula is used.

For $0 < x <= 1$, **ln**(**gamma**(x+1)) is used.

lngammacplx

Purpose

Returns the natural log of the Gamma function.

Format

f = **lngammacplx**(*z*);

Input

z	NxK matrix; *z* may be complex.

Output

f	NxK matrix.

Remarks

Note that **lngammacplx**(*z*) may yield a result with a different imaginary part than **ln**(**gammacplx**(*z*)). This is because **lngammacplx**(*z*) returns the value of the logarithm of **gamma**(*z*) on the corresponding branch of the complex plane, while a call to **ln**(z) always returns a function value with an imaginary part within [-π,π]. Hence the imaginary part of the result can differ by a multiple of 2*π. However, **exp** (**lngammacplx**(z)) = **gammacplx**(z). This routine uses a Lanczos series approximation for the complex **ln(gamma)** function.

References

1. C. Lanczos, SIAM JNA 1, 1964. pp. 86-96.
2. Y. Luke, "The Special ... approximations," 1969 pp. 29-31.
3. Y. Luke, "Algorithms ... functions," 1977.
4. J. Spouge, SIAM JNA 31, 1994. pp. 931.
5. W. Press, "Numerical Recipes."
6. S. Chang, "Computation of special functions," 1996.
7. P. Godfrey, "A note on the computation of the convergent Lanczos complex Gamma approximation."
8. Original code by Paul Godfrey

lnpdfmvn

Purpose

Computes multivariate Normal log-probabilities.

Format

z = **lnpdfmvn**(x, s);

Input

x	NxK matrix, data.
s	KxK matrix, covariance matrix.

Output

z	Nx1 vector, log-probabilities.

Remarks

This computes the multivariate Normal log-probability for each row of *x*.

Source

`lnpdfn.src`

lnpdfmvt

Purpose

Computes multivariate Student's t log-probabilities.

Format

z = **`lnpdfmvt`**(*x*, *s*, *nu*);

Input

x	NxK matrix, data.
s	KxK matrix, covariance matrix.
nu	scalar, degrees of freedom.

Output

z	Nx1 vector, log-probabilities.

Source

`lnpdfn.src`

See Also

lnpdft

lnpdfn

Purpose

Computes standard Normal log-probabilities.

Format

z = **`lnpdfn`**(*x*);

Input

x	NxK matrix or N-dimensional array, data.

Output

z	NxK matrix or N-dimensional array, log-probabilities.

Remarks

This computes the log of the scalar Normal density function for each element of *x*. *z* could be computed by the following **GAUSS** code:

```
     z = -ln(sqrt(2*pi))-x .* x / 2;
```

For multivariate log-probabilities, see **lnpdfmvn**.

Example

```
     x = { -2, -1, 0, 1, 2 };
     z = lnpdfn(x);
```

```
         -2.9189385
         -1.4189385
     z = -0.9189385
         -1.4189385
         -2.9189385
```

lnpdft

Purpose

Computes Student's t log-probabilities.

Format

z = **`lnpdft`**(*x*, *nu*);

Input

x	NxK matrix, data.
nu	scalar, degrees of freedom.

Output

z	NxK matrix, log-probabilities.

Remarks

This does not compute the log of the joint Student's t pdf. Instead, the scalar Normal density function is computed element-by-element.

For multivariate probabilities with covariance matrix see **`lnpdfmvt`**.

See Also

lnpdfmvt

load, loadf, loadk, loadm, loadp, loads

Purpose

Loads from a disk file.

Format

```
load [[path=path]]x, y[ ]=filename,  z=filename;
```

Remarks

All the `load`*xx* commands use the same syntax-they only differ in the types of symbols you use them for:

`load, loadm`	matrix
`loads`	string
`loadf`	function (`fn`)
`loadk`	keyword (`keyword`)
`loadp`	procedure (`proc`)

If no filename is given, as with *x* above, then the symbol name the file is to be loaded into is used as the filename, and the proper extension is added.

If more than one item is to be loaded in a single statement, the names should be separated by commas.

The filename can be either a literal or a string. If the filename is in a string variable, then the ^ (caret) operator must precede the name of the string, as in:

```
    filestr = "mydata/char";
    loadm x = ^filestr;
```

If no extension is supplied, the proper extension for each type of file will be used automatically as follows:

`load`	`.fmt` - matrix file or delimited ASCII file
`loadm`	`.fmt` - matrix file or delimited ASCII file
`loads`	`.fst` - string file
`loadf`	`.fcg` - user-defined function (`fn`) file
`loadk`	`.fcg` - user-defined keyword (`keyword`) file
`loadp`	`.fcg` - user-defined procedure (`proc`) file

These commands also signal to the compiler what type of object the symbol is so that later references to it will be compiled correctly.

A dummy definition must exist in the program for each symbol that is loaded in using `loadf`, `loadk`, or `loadp`. This resolves the need to have the symbol initialized at compile time. When the load executes, the dummy definition will be replaced with the saved definition:

```
    proc corrmat;
    endp;

    loadp corrmat;
    y = corrmat;

    keyword regress(x); endp;
    loadk regress;
    regress x on y z t from data01;

    fn sqrd=;
```

```
     loadf sqrd;
     y = sqrd(4.5);
```

To load **GAUSS** files created with the `save` command, no brackets are used with the symbol name.

If you use `save` to save a scalar error code 65535 (i.e., **`error(65535)`**), it will be interpreted as an empty matrix when you `load` it again.

ASCII data files

To load ASCII data files, square brackets follow the name of the symbol.

Numbers in ASCII files must be delimited with spaces, commas, tabs, or newlines. If the size of the matrix to be loaded is not explicitly given, as in:

```
     load x[] = data.asc;
```

GAUSS will load as many elements as possible from the file and create an Nx1 matrix. This is the preferred method of loading ASCII data from a file, especially when you want to verify if the load was successful. Your program can then see how many elements were actually loaded by testing the matrix with the **`rows`** command, and if that is correct, the Nx1 matrix can be **`reshape`**'d to the desired form. You could, for instance, put the number of rows and columns of the matrix right in the file as the first and second elements and **`reshape`** the remainder of the vector to the desired form using those values.

If the size of the matrix is explicitly given in the `load` command, then no checking will be done. If you use:

```
     load x[500,6] = data.asc;
```

GAUSS will still load as many elements as possible from the file into an Nx1 matrix and then automatically reshape it using the dimensions given.

If you `load` data from a file, `data.asc`, which contains nine numbers (1 2 3 4 5 6 7 8 9), then the resulting matrix will be as follows:

```
    load x[1,9] = data.asc;
```

```
    x = 1 2 3 4 5 6 7 8 9
```

```
    load x[3,3] = data.asc;
```

```
        1 2 3
    x = 4 5 6
        7 8 9
```

```
    load x[2,2] = data.asc;
```

```
    x = 1 2
        3 4
```

```
    load x[2,9] = data.asc;
```

```
    x = 1 2 3 4 5 6 7 8 9
        1 2 3 4 5 6 7 8 9
```

```
    load x[3,5] = data.asc;
```

```
        1 2 3 4 5
    x = 6 7 8 9 1
        2 3 4 5 6
```

`load` accepts pathnames. The following is legal:

```
    loadm k = /gauss/x;
```

This will load /gauss/x.fmt into *k*.

If the **path=** subcommand is used with load and save, the path string will be remembered until changed in a subsequent command. This path will be used whenever none is specified. There are four separate paths for:

1. load, loadm
2. loadf, loadp
3. loads
4. save

Setting any of the four paths will not affect the others. The current path settings can be obtained (and changed) with the **sysstate** function, cases 4-7.

```
     loadm path = /data;
```

This will change the loadm path without loading anything.

```
     load path = /gauss x,y,z;
```

This will load x.fmt, y.fmt, and z.fmt using /gauss as a path. This path will be used for the next load if none is specified.

The load path or save path can be overridden in any particular load or save by putting an explicit path on the filename given to load from or save to as follows:

```
     loadm path = /miscdata;
     loadm x = /data/mydata1, y, z = hisdata;
```

In the above program:

/data/mydata1.fmt would be loaded into a matrix called *x*.

/miscdata/y.fmt would be loaded into a matrix called *y*.

/miscdata/hisdata.fmt would be loaded into a matrix called *z*.

```
    oldmpath = sysstate(5,"/data");
    load x, y;
    call sysstate(5,oldmpath);
```

This will get the old `loadm` path, set it to /data, load `x.fmt` and `y.fmt`, and reset the `loadm` path to its original setting.

See Also

loadd, dataload, save, let, con, cons, sysstate

loadarray

Purpose

Loads an N-dimensional array from a disk file.

Format

`loadarray` [[`path=`*`path`*]] *x*, *y*=*`filename`*;

Remarks

If no filename is given, as with *x* above, then the symbol name the file is to be loaded into is used as the filename, and the proper extension is added.

If more than one item is to be loaded in a single statement, the names should be separated by commas.

The filename can be either a literal or a string. If the filename is in a string variable, then the ^ (caret) operator must precede the name of the string, as in:

```
    filestr = "mydata/adat";
    loadarray x = ^filestr;
```

If no extension is supplied, then an `.fmt` extension will be assumed.

`loadarray` accepts pathnames. The following is legal:

```
    loadarray k = /gauss/a;
```

This will load `/gauss/a.fmt` into *k*.

If the **`path=`** subcommand is used, the path string will be remembered until changed in a subsequent command. This path will be used for all `loadarray`, `loadm`, and `load` calls whenever none is specified.

The current path setting can be obtained (and changed) with the **`sysstate`** function, case 5.

```
    loadarray path = /data;
```

This will change the `loadarray` path without loading anything.

```
    loadarray path = /gauss a,b,c;
```

This will load `a.fmt`, `b.fmt`, and `c.fmt` using `/gauss` as a path. This path will be used for the next `loadarray`, `loadm`, or `load` call if none is specified.

The `load` path or `save` path can be overridden in any particular `load` or `save` by putting an explicit path on the filename given to `load` from or `save` to as follows:

```
    loadarray path = /miscdata;
    loadarray a = /data/mydata1, b, c = hisdata;
```

In the above program:

`/data/mydata1.fmt` would be loaded into an array called *a*.

`/miscdata/b.fmt` would be loaded into an array called *b*.

`/miscdata/hisdata.fmt` would be loaded into an array called `c`.

```
    oldarraypath = sysstate(5,"/data");
    loadarray a, b;
    call  sysstate(5,oldarraypath);
```

This will get the old `loadarray` path, set it to `/data`, load `a.fmt` and `b.fmt`, and reset the `loadarray` path to its original setting.

See Also

load, loadm, save, let, sysstate

loadd

Purpose

Loads a data set.

Format

y = **`loadd`**(*`dataset`*);

Input

`dataset`	string, name of data set.

Output

y	NxK matrix of data.

Remarks

The data set must not be larger than a single **GAUSS** matrix.

If *`dataset`* is a null string or 0, the data set `temp.dat` will be loaded. To load a matrix file, use an `.fmt` extension on *`dataset`*.

Source

`saveload.src`

Globals

`__maxvec`

loadstruct

Purpose

Loads a structure into memory from a file on the disk.

Format

{ *`instance`*, *`retcode`* } = **`loadstruct`**(*`file_name`*, *`structure_type`*);

Input

`file_name`	string, name of file containing structure.
`structure_type`	string, structure type.

Output

instance	instance of the structure.
retcode	scalar, 0 if successful, otherwise 1.

Remarks

instance can be an array of structures.

Example

```
    #include ds.sdf
    struct DS p3;

    { p3, retc } = loadstruct("p2", "ds");
```

loadwind

Purpose

Load a previously saved graphic panel configuration. Note: This function is for use with the deprecated PQG graphics.

Library

pgraph

Format

```
    err = loadwind(namestr);
```

Input

namestr	string, name of file to be loaded.

Output

err	scalar, 0 if successful, 1 if graphic panel matrix is invalid. Note that the current graphic panel configuration will be overwritten in either case.

Source

`pwindow.src`

Globals

_pwindmx

See Also

savewind

local

Purpose

Declare variables that are to exist only inside a procedure.

Format

```
local x, y, f:proc;
```

Remarks

The statement above would place the names `x`, `y`, and `f` in the local symbol table for the current procedure being compiled. This statement is legal only between the `proc` statement and the `endp` statement of a procedure definition.

These symbols cannot be accessed outside of the procedure.

The symbol `f` in the statement above will be treated as a procedure whenever it is accessed in the current procedure. What is actually passed in is a pointer to a procedure.

See **Procedures and Keywords**, Chapter 1.

See Also

proc

locate

Purpose

Positions the cursor in the window.

Format

```
locate m, n;
```

Remarks

`locate` locates the cursor in the current output window.

m and *n* denote the row and column, respectively, at which the cursor is to be located.

The origin (1,1) is the upper left corner.

m and *n* may be any expressions that return scalars. Nonintegers will be truncated to an integer.

Example

```
    r = csrlin;
    c = csrcol;
    cls;
    locate r,c;
```

In this example the window is cleared without affecting the cursor position.

See Also

csrlin, csrcol

loess

Purpose

Computes coefficients of locally weighted regression.

Format

{ *yhat*, *ys*, *xs* } = **loess**(*depvar*, *indvars*);

Input

depvar	Nx1 vector, dependent variable.
indvars	NxK matrix, independent variables.

Global Input

_loess_Span	scalar, degree of smoothing. Must be greater than 2/N. Default = .67777.
_loess_NumEval	scalar, number of points in *ys* and *xs*. Default = 50.
_loess_Degree	scalar, if 2, quadratic fit, otherwise linear. Default = 1.
_loess_WgtType	scalar, type of weights. If 1, robust, symmetric weights, otherwise Gaussian. Default = 1.
__output	scalar, if 1, iteration information and results are printed, otherwise nothing is printed.

Output

yhat	Nx1 vector, predicted *depvar* given *indvars*.
ys	*_loess_numEval*x1 vector, ordinate values given abscissae values in *xs*.
xs	*_loess_numEval*x1 vector, equally spaced abscissae values.

Remarks

Based on Cleveland, William S. "Robust Locally Weighted Regression and Smoothing Scatterplots." JASA, Vol. 74, 1979, 829-836.

Source

```
loess.src
```

loessmt

Purpose

Computes coefficients of locally weighted regression.

Include

```
loessmt.sdf
```

Format

{ *yhat*, *ys*, *xs* } = **loessmt**(*lc0*, *depvar*, *indvars*);

Input

lc0	an instance of a **loessmtControl** structure, containing the following members:
	lc0.Span — scalar, degree of smoothing. Must be greater than 2/N. Default = .67777.
	lc0.NumEval — scalar, number of points in *ys* and *xs*. Default = 50.

	`lc0.Degree`	scalar, if 2, quadratic fit, otherwise linear. Default = 1.
	`lc0.WgtType`	scalar, type of weights. If 1, robust, symmetric weights, otherwise Gaussian. Default = 1.
	`lc0.output`	scalar, if 1, iteration information and results are printed, otherwise nothing is printed.
depvar	Nx1 vector, dependent variable.	
indvars	NxK matrix, independent variables.	

Output

yhat	Nx1 vector, predicted *depvar* given *indvars*.
ys	*lc0.numEval* x 1 vector, ordinate values given abscissae values in *xs*.
xs	*lc0.numEval* x 1 vector, equally spaced abscissae values.

Remarks

Based on Cleveland, William S. "Robust Locally Weighted Regression and Smoothing Scatterplots." JASA, Vol. 74, 1979, 829-836.

Source

`loessmt.src`

See Also

loessmtControlCreate

loessmtControlCreate

Purpose

Creates default **loessmtControl** structure.

Include

`loessmt.sdf`

Format

c = **`loessmtControlCreate`**();

Output

c	instance of a **loessmtControl** structure with members set to default values.

Example

```
struct loessmtControl lc;
lc = loessmtControlCreate();
```

Source

`loessmt.src`

See Also

loessmt

log

Purpose

Computes the log of all elements of *x*.

Format

y = **`log`**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing the log 10 values of the elements of *x*.

Remarks

`log` is defined for $x \neq 0$.

You can turn the generation of complex numbers for negative inputs on or off in the **GAUSS** configuration file, and with the **`sysstate`** function, case 8. If you turn it off, **`log`** will generate an error for negative inputs.

If *x* is already complex, the complex number state doesn't matter; **`log`** will compute a complex result.

x can be any expression that returns a matrix.

Example

```
//Create a 3x3 matrix of random uniform integers from 1
//to 11
x = round(rndu(3,3)*10+1);
y = log(x);
```

If *x* is equal to:

```
4.000   9.000   2.000
5.000   3.000   7.000
2.000   6.000  10.000
```

Then *y* will be equal to:

```
0.602   0.954   0.301
0.699   0.477   0.845
0.301   0.778   1.000
```

See Also

ln

loglog

Purpose

Graphs X vs. Y using log coordinates. Note: This function is for use with the deprecated PQG graphics. Use **`plotLogLog`** instead.

Library

pgraph

Format

```
loglog(x, y);
```

Input

x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

Source

`ploglog.src`

See Also

xy, logx, logy

logx

Purpose

Graphs X vs. Y using log coordinates for the X axis. Note: This function is for use with the deprecated PQG graphics. Use **`plotLogX`** instead.

Library

`pgraph`

Format

```
logx(x, y);
```

Input

x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

Source

`plogx.src`

See Also

xy, logy, loglog

logy

Purpose

Graphs X vs. Y using log coordinates for the Y axis. Note: This function is for use with the deprecated PQG graphics. Use **`plotLogY`** instead.

Library

`pgraph`

Format

```
logy(x, y);
```

Input

x	Nx1 or NxM matrix. Each column represents the X values for a particular line.
y	Nx1 or NxM matrix. Each column represents the Y values for a particular line.

Source

plogy.src

See Also

xy, logx, loglog

loopnextindex

Purpose

Increments an index vector to the next logical index and jumps to the specified label if the index did not wrap to the beginning.

Format

```
loopnextindex lab, i, o [, dim];
```

Input

lab	literal, label to jump to if `loopnextindex` succeeds.
i	Mx1 vector of indices into an array, where M<=N.
o	Nx1 vector of orders of an N-dimensional array.
dim	scalar [1-M], index into the vector of indices *i*, corresponding to the dimension to walk through, positive to walk the index forward, or negative to walk backward.

Remarks

If the argument *dim* is given, `loopnextindex` will walk through only the dimension indicated by *dim* in the specified direction. Otherwise, if *dim* is not given, each call to `loopnextindex` will increment *i* to index the next element or subarray of the corresponding array.

`loopnextindex` will jump to the label indicated by *lab* if the index can walk further in the specified dimension and direction, otherwise it will fall out of the loop and continue through the program.

When the index matches the vector of orders, the index will be reset to the beginning and program execution will resume at the statement following the `loopnextindex` statement.

Example

At its essence, `loopNextIndex` provides a simple way to iterate over the orders of a multi-dimensional array.

```
    //The orders of the array
    orders = { 2, 3, 4 };
```

```
//The starting index of the array
ind = { 1, 1, 1 };

lnilab:
print "ind = " ind;
loopNextIndex lnilab, ind, orders;
```

Running the code above, returns:

```
ind =
 1.000
 1.000
 1.000
ind =
 1.000
 1.000
 2.000
ind =
 1.000
 1.000
 3.000
ind =
 1.000
 1.000
 4.000
ind =
 1.000
 2.000
 1.000
ind =
 1.000
 2.000
 2.000
ind =
      1.000
      2.000
      3.000

      ...continuing on to end with...

      ind =
      2.000
      3.000
      4.000
```

This next example uses the variable *ind* to iterate over and make assignments to the array, *a*.

```
     orders = { 2,3,4,5,6,7 };
     a = arrayalloc(orders,0);
     ind = { 1,1,1,1 };

     loopni:

     setarray a, ind, rndn(6,7);
     loopnextindex loopni, ind, orders;
```

This example sets each 6x7 subarray of array *a*, by incrementing the index at each call of `loopnextindex` and then going to the label *loopni*. When *ind* cannot be incremented, the program drops out of the loop and continues.

```
     ind = { 1,1,4,5 };

     loopni2:

     setarray a, ind, rndn(6,7);
     loopnextindex loopni2, ind, orders, 2;
```

Using the array and vector of orders from the example above, this example increments the second value of the index vector `ind` during each call to `loopnextindex`. This loop will set the 6x7 subarrays of *a* that begin at [1,1,4,5,1,1], [1,2,4,5,1,1], and [1,3,4,5,1,1], and then drop out of the loop.

See Also

nextindex, previousindex, walkindex

lower

Purpose

Converts a string or character matrix to lowercase.

Format

y = **`lower`**(*x*);

Input

x	string or NxK matrix of character data to be converted to lowercase.

Output

y	string or NxK matrix which contains the lowercase equivalent of the data in *x*.

Remarks

If *x* is a numeric matrix, *y* will contain garbage. No error message will be generated since **GAUSS** does not distinguish between numeric and character data in matrices.

Example

```
x = "MATH 401";
y = lower(x);
print y;
```

produces:

```
     math 401
```

The **`lower`** function can be useful when performing case insensitive string comparisons. If you have a program that runs different code depending upon the variable name in a **GAUSS** dataset or spreadsheet file, you or your colleagues may want to analyze data with inconsistent use of case.

```
var1 = "Consumption";

if lower(var1) == "gdp";
   //code for gdp branch
else if lower(var1) == "consumption";
   //code for consumption branch
endif;
```

Using the **`lower`** function, the code above will operate correctly whether *`var1`* is Consumption, CONSUMPTION or consumption.

See Also

upper

lowmat, lowmat1

Purpose

Returns the lower portion of a matrix. **`lowmat`** returns the main diagonal and every element below. **`lowmat1`** is the same except it replaces the main diagonal with ones.

Format

L = **`lowmat`**(*x*);
L = **`lowmat1`**(*x*);

Input

x	NxN matrix.

Output

L	NxN matrix containing the lower elements of the matrix. The upper elements are replaced with zeros. **`lowmat`** returns the main diagonal intact. **`lowmat1`** replaces the main diagonal with ones.

Remarks

The **`lowmat`** function along with **`upmat1`** can be used to extract the LU factors from the return

Example

```
x = { 1 2 -1,
      2 3 -2,
      1 -2 4 };

L = lowmat(x);
L1 = lowmat1(x);
```

The resulting matrices are

```
    1  0  0         1   0   0
L = 2  3  0   L1 =  2   1   0
    1 -2  4         1  -2   1
```

Source

`diag.src`

See Also

upmat, upmat1, diag, diagrv, crout, croutp

ltrisol

Purpose

Computes the solution of $Lx = b$ where L is a lower triangular matrix.

Format

x = **ltrisol**(b, L);

Input

b	PxK matrix.
L	PxP lower triangular matrix.

Output

x	PxK matrix, soluion of *Lx = b*.

ltrisol applies a forward solve to *Lx = b* to solve for *x*. If *b* has more than one column, each column will be solved for separately, i.e., **ltrisol** will apply a forward solve to *L***x*[., *i*] = *b*[., *i*].

lu

Purpose

Computes the LU decomposition of a square matrix with partial (row) pivoting, such that: $X = LU$.

Format

```
{ l, u } = lu(x);
```

Input

x	NxN square nonsingular matrix.

Output

l	NxN "scrambled" lower triangular matrix. This is a lower triangular matrix that has been reordered based on the row pivoting.
u	NxN upper triangular matrix.

Example

```
//Set seed for repeatable random numbers
rndseed 13;

//Print format, display 4 digits after decimal point
format /rd 10,4;

A = rndn(3,3);
{ L, U } = lu(A);
A2 = L*U;
```

```
      -0.0195      0.4054     -0.0874
A =   -1.2948      0.1734      1.9712
       0.5408     -0.1294      0.7646

       0.0150      1.0000      0.0000
L =    1.0000      0.0000      0.0000
      -0.4177     -0.1414      1.0000

      -1.2948      0.1734      1.9712
U =    0.0000      0.4028     -0.1170
       0.0000      0.0000      1.5714

      -0.0195      0.4054     -0.0874
```

```
    L*U = -1.2948      0.1734      1.9712
           0.5408     -0.1294      0.7646
```

See Also

crout, croutp, chol

lusol

Purpose

Computes the solution of $LUx = b$ where L is a lower triangular matrix and U is an upper triangular matrix.

Format

x = **lusol**(b, L, U);

Input

b	PxK matrix.
L	PxP lower triangular matrix.
U	PxP upper triangular matrix.

Output

x	PxK matrix, solution of $LUx = b$.

Remarks

If b has more than one column, each column is solved for separately, i.e., **lusol** solves $LUx[., i] = b[., i]$.

m

machEpsilon

Purpose

Returns the smallest number such that 1+*eps* > 1.

Format

eps = **machEpsilon**;

Output

eps	scalar, machine epsilon.

Source

`machconst.src`

make (dataloop)

Purpose

Specifies the creation of a new variable within a data loop.

Format

```
make [#] numvar = numeric_expression;
make $charvar = character_expression;
```

Remarks

A *numeric_expression* is any valid expression returning a numeric vector. A *character_expression* is any valid expression returning a character vector. If neither '$' nor '#' is specified, '#' is assumed.

The expression may contain explicit variable names and/or **GAUSS** commands. Any variables referenced must already exist, either as elements of the source data set, as `extern`'s, or as the result of a previous `make`, `vector`, or **`code`** statement. The variable name must be unique. A variable cannot be made more than once, or an error is generated.

Example

```
    make sqvpt = sqrt(velocity * pressure * temp);
    make $ gender = lower(gender);
```

See Also

vector (dataloop)

makevars

Purpose

Creates separate global vectors from the columns of a matrix.

Format

```
makevars(x, vnames, xnames);
```

Input

x	NxK matrix whose columns will be converted into individual vectors.
vnames	string or Mx1 character vector containing names of global vectors to create. If 0, all names in *xnames* will be used.
xnames	string or Kx1 character vector containing names to be associated with the columns of the matrix *x*.

Remarks

If *xnames* = 0, the prefix X will be used to create names. Therefore, if there are 9 columns in *x*, the names will be X1-X9, if there are 10, they will be X01-X10, and so on.

If *xnames* or *vnames* is a string, the individual names must be separated by spaces or commas:

```
    vnames = "age pay sex";
```

Since these new vectors are created at execution time, the compiler will not know they exist until after **makevars** has executed once. This means that you cannot access them by name unless you previously `clear` them or otherwise add them to the symbol table. (See **setvars** for a quick interactive solution to this.)

This function is the opposite of **mergevar**.

Example

```
    let x[3,3] = 101 35 50000
                 102 29 13000
                 103 37 18000;
    let xnames = id age pay;
    let vnames = age pay;
    makevars(x,vnames,xnames);
```

Two global vectors, called `age` and `pay`, are created from the columns of *x*.

```
    let x[3,3] = 101 35 50000
                 102 29 13000
                 103 37 18000;
    xnames = "id age pay";
    vnames = "age pay";
    makevars(x,vnames,xnames);
```

This is the same as the example above, except that strings are used for the variable names.

Source

`vars.src`

Globals

__vpad

See Also

mergevar, setvars

makewind

Purpose

Creates a graphic panel of specific size and position and adds it to the list of graphic panels. Note: This function is for the deprecated PQG graphics. For similar functionality, see **`plotLayout`** and **`plotCustomLayout`**.

Library

`pgraph`

Format

```
makewind(xsize, ysize, xshft, yshft, typ);
```

Input

xsize	scalar, horizontal size of the graphic panel in inches.
ysize	scalar, vertical size of the graphic panel in inches.
xshft	scalar, horizontal distance from left edge of window in inches.
yshft	scalar, vertical distance from bottom edge of window in inches.
typ	scalar, graphic panel attribute type. If this value is 1, the graphic panels will be transparent. If 0, the graphic panels will be nontransparent.

Remarks

Note that if this procedure is used when rotating the page, the passed parameters are scaled appropriately to the newly oriented page. The size and shift values will not be true inches when printed, but the graphic panel size to page size ratio will remain the same. The result of this implementation automates the rotation and eliminates the required graphic panel recalculations by the user.

See the **`window`** command for creating tiled graphic panels. For more information on using graphic panels, see **Tiled Graphic Panels**, Section 1.0.1.

Source

`pwindow.src`

See Also

window, endwind, setwind, getwind, begwind, nextwind

margin

Purpose

Sets the margins for the current graph's graphic panel. Note: This function is for use with the deprecated PQG graphics. For similar functionality, use **`plotCustomLayout`**.

Library

`pgraph`

Format

`margin`(*l*, *r*, *t*, *b*);

Input

l	scalar, the left margin in inches.
r	scalar, the right margin in inches.
t	scalar, the top margin in inches.
b	scalar, the bottom margin in inches.

Remarks

By default, the dimensions of the graph are the same as the graphic panel dimensions. With this function the graph dimensions may be decreased. The result will be a smaller plot area surrounded by the specified margin. This procedure takes into consideration the axes labels and numbers for correct placement.

All input inch values for this procedure are based on a full size window of 9x6.855 inches. If this procedure is used with a graphic panel, the values will be scaled to "window inches" automatically.

If the axes must be placed an exact distance from the edge of the page, **`axmargin`** should be used.

Source

`pgraph.src`

See Also

axmargin

matalloc

Purpose

Allocates a matrix with unspecified contents.

Format

y = **`matalloc`**(*r*, *c*);

Input

r	scalar, rows.
c	scalar, columns.

Output

y	*r* x *c* matrix.

Remarks

The contents are unspecified. This function is used to allocate a matrix that will be written to in sections using indexing or used with the Foreign Language Interface as an output matrix for a function called with `dllcall`.

See Also

matinit, ones, zeros, eye

matinit

Purpose

Allocates a matrix with a specified fill value.

Format

y = **`matinit`**(*r*, *c*, *v*);

Input

r	scalar, rows.
c	scalar, columns.
v	scalar, value to initialize.

Output

y	*r* x *c* matrix with each element equal to the value of *v*.

Example

```
format /rd 6,2;
print matinit(3, 4, pi);
```

```
  3.14   3.14   3.14   3.14
  3.14   3.14   3.14   3.14
  3.14   3.14   3.14   3.14
```

See Also

matalloc, ones, zeros, eye

mattoarray

Purpose

Converts a matrix to a type array.

Format

y = **mattoarray**(*x*);

Input

x	matrix.

Output

y	1-or-2-dimensional array.

Remarks

If the argument *x* is a scalar, **mattoarray** will simply return the scalar, without changing it to a type array.

Example

```
    x = 5*ones(2,3);
    y = mattoarray(x);
```

y will be a 2x3 array of fives.

See Also

arraytomat

maxbytes

Purpose

Returns maximum memory to be used.

Format

```
y = maxbytes;
```

Global Input

__maxbytes	scalar, maximum memory to be used.

Output

y	scalar, maximum memory to be used.

Remarks

maxbytes returns the value in the global scalar *__maxbytes*, which can be reset in the calling program.

maxbytes is called by **Run-Time Library** functions and applications when determining how many rows can be read from a data set in one call to **readr**.

maxbytes replaced the obsolete command **coreleft**. If **coreleft** returns a meaningful number for your operating system and if you wish to reference it, set __*maxbytes* = 0 and then call **maxbytes**.

Example

```
y = maxbytes;
print y;
```

```
 100000000.000
```

Source

system.src

maxc

Purpose

Returns a column vector containing the largest element in each column of a matrix.

Format

y = **maxc**(*x*);

Input

x	NxK matrix or sparse matrix.

Output

y	Kx1 matrix containing the largest element in each column of *x*.

Remarks

If *x* is complex, **maxc** uses the complex modulus (**abs**(*x*)) to determine the largest elements.

To find the maximum elements in each row of a matrix, transpose the matrix before applying the **maxc** function.

To find the maximum value in the whole matrix if the matrix has more than one column, nest two calls to **maxc**:

```
y = maxc(maxc(x));
```

Example

```
x = rndBeta(4,2,3,1);
y = maxc(x);
```

If *x* equals:

```
0.87174453 0.70281291
0.90393029 0.95919009
```

```
0.82960656 0.58022236
0.80910492 0.61975567
```

then *y* will equal:

```
     0.90393029
     0.95919009
```

See Also

minc, maxindc, minindc

maxindc

Purpose

Returns a column vector containing the index (i.e., row number) of the maximum element in each column of a matrix.

Format

y = **maxindc**(*x*);

Input

x	NxK matrix.

Output

y	Kx1 matrix containing the index of the maximum element in each column of *x*.

Remarks

If *x* is complex, **maxindc** uses the complex modulus (**abs**(*x*)) to determine the largest elements.

To find the index of the maximum element in each row of a matrix, transpose the matrix before applying **maxindc**.

To find the indices of the largest element in a matrix *x*, use:

```
colInd = maxindc(maxc(x));
rowInd = maxindc(x[.,colInd]);
```

If there are two or more "largest" elements in a column (i.e., two or more elements equal to each other and greater than all other elements), then **maxindc** returns the index of the first one found, which will be the smallest index.

Example

```
x = round(rndn(4,4)*5);
mx = maxc(x);
mxInd = maxindc(x);
```

If *x* is equal to:

```
-2   -8   -1   -2
-1    9    0    7
 9    0    4    8
-2    6    6    1
```

then

```
     9             3
mx = 9    mxInd = 2
```

```
        6                  4
        8                  3
```

See Also

maxc, minindc, minc

maxv

Purpose

Performs an element by element comparison of two matrices and returns the maximum value for each element.

Format

z = **maxv**(*x*, *y*);

Global Input

x	NxK matrix
y	NxK matrix

Output

z	A NxK matrix whose values are the maximum of each element from the arguments *x* and *y*.

Remarks

maxv works for sparse matrices as well as arrays.

Example

```
//Create the sequence 1, 2, 3,...10
x = seqa(1, 1, 10);

//Set 'y' equal to the reverse order of 'x'
y = rev(x);

z = maxv(x,y);
```

```
      1         10         10
      2          9          9
      3          8          8
      4          7          7
x  =  5    y  =  6    z  =  6
      6          5          6
      7          4          7
      8          3          8
      9          2          9
     10          1         10
```

See Also

minv

maxvec

Purpose

Returns maximum vector length allowed.

Format

```
y = maxvec;
```

Global Input

__maxvec	scalar, maximum vector length allowed.

Output

y	scalar, maximum vector length.

Remarks

maxvec returns the value in the global scalar *__maxvec*, which can be reset in the calling program.

maxvec is called by **Run-Time Library** functions and applications when determining how many rows can be read from a data set in one call to **readr**.

Using a value that is too large can cause excessive disk thrashing. The trick is to allow the algorithm making the disk reads to execute entirely in RAM.

Example

```
y = maxvec;
print y;
```

```
20000.000
```

Source

`system.src`

mbesseli

Purpose

Computes modified and exponentially scaled modified Bessels of the first kind of the nth order.

Format

y = **`mbesseli`**(*x*, *n*, *alpha*);
y = **`mbesseli0`**(*x*);
y = **`mbesseli1`**(*x*);
y = **`mbesselei`**(*x*, *n*, *alpha*);
y = **`mbesselei0`**(*x*);
y = **`mbesselei1`**(*x*);

Input

x	Kx1 vector, abscissae.

n	scalar, highest order.
alpha	scalar, 0 <= *alpha* < 1.

Output

y	KxN matrix, evaluations of the modified Bessel or the exponentially scaled modified Bessel of the first kind of the nth order.

Remarks

For the functions that permit you to specify the order, the returned matrix contains a sequence of modified or exponentially scaled modified Bessel values of different orders. For the *i*th row of *y*:

```
    y[i,.] = I_α(x[i]) I_α+1(x[i])...I_alpha+n-1(x[i])
```

The remaining functions generate modified Bessels of only the specified order.

The exponentially scaled modified Bessels are related to the unscaled modifed Bessels in the following way:

```
    mbesselei0(x) = exp(-x) * mbesseli0(x)
```

The use of the scaled versions of the modified Bessel can improve the numerical properties of some calculations by keeping the intermediate numbers small in size.

Example

This example produces estimates for the "circular" response regression model (Fisher, N.I. *Statistical Analysis of Circular Data*. NY: Cambridge University Press, 1993.), where the dependent variable varies between -π and π in a circular manner. The model is

```
    y = µ + G(XB)
```

where **B** is a vector of regression coefficients, *x* a matrix of independent variables with a column of 1's included for a constant, and *y* a vector of "circular" dependent variables, and where *G*() is a function mapping *XB* onto the [-π, π] interval.

The log-likelihood for this model is from Fisher, N.I. ... 1993, 159:

$$log\ L = -N \times ln(I_0(\kappa)) + \kappa \sum_i^N cos(y_i - \mu - G(X_i B))$$

To generate estimates it is necessary to maximize this function using an iterative method. **QNewton** is used here.

κ is required to be nonnegative and therefore in the example below, the exponential of this parameter is estimated instead. Also, the exponentially scaled modified Bessel is used to improve numerical properties of the calculations.

The **arctan** function is used in *G*() to map *XB* to the [-π, π] interval as suggested by Fisher, N.I. ... 1993, 158.

```
    proc G(u);
       retp(2*atan(u));
    endp;

    proc lpr(b);
       local dev;
       /*
       ** b[1] - kappa
       ** b[2] - mu
       ** b[3] - constant
       ** b[4:rows(b)] - coefficients
       */
       dev = y - b[2]- G(b[3] + x * b[4:rows(b)]);
       retp(rows(dev)*ln(mbesselei0(exp(b[1])) -
          sumc(exp(b[1])*(cos(dev)-1))));
    endp;

    loadm data;
    y0 = data[.,1];
    x0 = data[.,2:cols(data)];

    b0 = 2*ones(cols(x0),1);

    { b,fct,grd,ret } = QNewton(&lpr,b0);

    cov = invpd(hessp(&lpr,b));

    print "estimates standard errors";
    print;
    print b~sqrt(diag(cov));
```

Source

ribesl.src

meanc

Purpose

Computes the mean of every column of a matrix.

Format

y = **meanc**(*x*);

Input

x	NxK matrix.

Output

y	Kx1 matrix containing the mean of every column of *x*.

Example

```
    x = meanc(rndu(1e5,4));
```

After the code above, *x* is equal to:

```
    0.5007
    0.5004
    0.4995
    0.5016
```

In this example, 4 columns of uniform random numbers are generated in a matrix, and the mean is computed for each column. Due to the use of random input data in this example, your results may differ slightly.

See Also

stdc

median

Purpose

Computes the medians of the columns of a matrix.

Format

m = **median**(*x*);

Input

x	NxK matrix.

Output

m	Kx1 vector containing the medians of the respective columns of *x*.

Example

```
//Set the seed for repeatable random data
rndseed 4320993;

//Create uniform random integers between 1 and 10
x = ceil(10*rndu(100,3));

//Calculate the median of each column of 'x'
md = median(x);
```

After the code above, *md* is equal to:

```
5.0000
5.0000
6.0000
```

Source

median.src

mergeby

Purpose

Merges two sorted files by a common variable.

Format

```
mergeby(infile1, infile2, outfile, keytyp);
```

Input

infile1	string, name of input file 1.
infile2	string, name of input file 2.
outfile	string, name of output file.
keytyp	scalar, data type of key variable. *1* numeric *2* character

Remarks

This will combine the variables in the two files to create a single large file. The following assumptions hold:

1. Both files have a single (key) variable in common and it is the first variable.
2. All of the values of the key variable are unique.
3. Each file is already sorted on the key variable.

The output file will contain the key variable in its first column.

It is not necessary for the two files to have the same number of rows. For each row for which the key variables match, a row will be created in the output file. *outfile* will contain the columns from *infile1* followed by the columns from *infile2* minus the key column from the second file.

If the inputs are null ("" or 0), the procedure will ask for them.

Source

```
sortd.src
```

mergevar

Purpose

Accepts a list of names of global matrices, and concatenates the corresponding matrices horizontally to form a single matrix.

Format

x = **mergevar**(*vnames*);

Input

vnames	string or Kx1 column vector containing the names of K global matrices.

Output

x	NxM matrix that contains the concatenated matrices, where M is the sum of the columns in the K matrices specified in *vnames*.

Remarks

The matrices specified in *vnames* must be globals and they must all have the same number of rows.

This function is the opposite of **makevars**.

Example

```
//Random integers between 1 and 72
age = ceil(72 * rndu(100, 1));

//Random normal numbers with a mean of 70 and a standard
//deviation of 10
income = 10 * rndn(100, 1) + 70;

//Vertically concatenate the strings
vnames = "age"$|"income";

//Merge the variables into 1 matrix
agInc = mergevar(vnames);
```

The column vectors *age* and *income* will be concatenated horizontally to create *agInc*. The above call to **mergevar** is equivalent to:

```
//Combine the matrices using the horizontal concatenation
//operator
agInc = age~income;
```

Source

vars.src

See Also

makevars

minc

Purpose

Returns a column vector containing the smallest element in each column of a matrix.

Format

$y =$ **minc**(x);

Input

x	NxK matrix or sparse matrix.

Output

y	Kx1 matrix containing the smallest element in each column of x.

Remarks

If *x* is complex, **minc** uses the complex modulus (**abs**(*x*)) to determine the smallest elements.

To find the minimum element in each row, transpose the matrix before applying the **minc** function.

To find the minimum value in the whole matrix, nest two calls to **minc**:

```
    y = minc(minc(x));
```

Example

```
    x = rndn(4,2);
    y = minc(x);
```

If *x* is equal to:

```
    -1.9950  -1.3477
    -0.4031  -1.9137
     0.8136  -2.3155
    -0.9947   1.4061
```

then *y* will equal:

```
    -1.9950
    -2.3155
```

See Also

maxc, minindc, maxindc

minindc

Purpose

Returns a column vector containing the index (i.e., row number) of the smallest element in each column of a matrix.

Format

y = **minindc**(*x*);

Input

x	NxK matrix.

Output

y	Kx1 matrix containing the index of the smallest element in each column of *x*.

Remarks

If *x* is complex, **minindc** uses the complex modulus (**abs**(*x*)) to determine the smallest elements.

To find the index of the smallest element in each row, transpose the matrix before applying **minindc**.

To find the index of the smallest element in a matrix *x*, use:

```
colInd = minindc(minc(x));
rowInd = minindc(x[.,colInd]);
```

If there are two or more "smallest" elements in a column (i.e., two or more elements equal to each other and less than all other elements), then **minindc** returns the index of the first one found, which will be the smallest index.

Example

```
x = round(rndn(5,4)*5);
y = minc(x);
z = minindc(x);
```

If *x* is equal to:

```
         -5      4     -4      0
         -2      3      4      3
x =     -11      5      5      5
          1      2      7      4
         -2      4     -1     -5
```

then *y* and *z* are equal to:

```
       -11           3
y =      2       z = 4
        -4           1
        -5           5
```

See Also

maxindc, minc, maxc

minv

Purpose

Performs an element by element comparison of two matrices and returns the minimum value for each element.

Format

z = **minv**(*x*, *y*);

Global Input

x	NxK matrix
y	NxK matrix

Output

z	A NxK matrix whose values are the minimum of each element from the arguments *x* and *y*.

Remarks

minv works for sparse matrices as well as arrays.

Example

```
//Create the multiplicative sequence 1, 2, 4, 8
x = seqm(1,2,4);
```

```
//Reverse the order of the elements in 'x' and assign them
//to 'y'
y = rev(x);

z = minv(x,y);
```

After the code above:

```
        1            8            1
x = 2        y = 4        z = 2
        4            2            2
        8            1            1
```

See Also

maxv

miss, missrv

Purpose

miss converts specified elements in a matrix to **GAUSS**'s missing value code. **missrv** is the reverse of this, and converts missing values into specified values.

Format

y = **miss**(*x*, *v*);
y = **missrv**(*x*, *v*);

Input

x	NxK matrix.
v	LxM matrix, ExE conformable with *x*.

Output

y	max(N,L) by max(K,M) matrix.

Remarks

For **miss**, elements in *x* that are equal to the corresponding elements in *v* will be replaced with the **GAUSS** missing value code.

For **missrv**, elements in *x* that are equal to the **GAUSS** missing value code will be replaced with the corresponding element of *v*.

For complex matrices, the missing value code is defined as a missing value entry in the real part of the matrix. For complex *x*, then, **miss** replaces elements with a ". + 0i" value, and **missrv** examines only the real part of *x* for missing values. If, for example, an element of *x* = 1 + .i, **missrv** will not replace it.

These functions act like element-by-element operators. If *v* is a scalar, for instance -1, then all -1's in *x* are converted to missing. If *v* is a row (column) vector with the same number of columns (rows) as *x*, then each column (row) in *x* is transformed to missings according to the corresponding element in *v*. If *v* is a matrix of the same size as *x*, then the transformation is done corresponding element by corresponding element.

Missing values are given special treatment in the following functions and operators: *b*/*A* (matrix division when *a* is not square and neither *a* nor *b* is scalar), **counts**,

scalmiss, **maxc**, **maxindc**, **minc**, **minindc**, **miss**, **missex**, **missrv**, **moment**, **packr**, **scalmiss**, **sortc**.

As long as you know a matrix contains no missings to begin with, **miss** and **missrv** can be used to convert one set of numbers into another. For example:

```
     y = missrv(miss(x,0),1);
```

will convert 0's to 1's.

To convert a range of values, such as:

```
     0.5 < x < 1.3
```

into missing values, use the **missex** function.

Example

Example 1

```
x = { 1, 2 };
x_miss = miss(x, 2);
```

After the code above:

```
x_miss = 1
         .
```

Example 2

```
//Create a 3x3 matrix with each element equal to 1
x = ones(3, 3);

//Assign the diagonal of 'x' to be equal to pi
x = diagrv(x, pi);

print "x = " x;

//Change all 1's in 'x' into missing values and assign to
//x_miss
x_miss = miss(x, 1);

print "x_miss = " x_miss;

//Change all missings in 'x_miss' into 2*pi and assign to x
2
x2 = missrv(x_miss, 2*pi);

print "x2 = " x2;
```

The code above, will return:

```
    x =
           3.1415927        1.0000000        1.0000000
           1.0000000        3.1415927        1.0000000
           1.0000000        1.0000000        3.1415927
    x_miss =
           3.1415927                .                .
                   .        3.1415927                .
                   .                .        3.1415927
    x2 =
           3.1415927        6.2831853        6.2831853
           6.2831853        3.1415927        6.2831853
           6.2831853        6.2831853        3.1415927
```

Example 3

If the first input, *x*, is a matrix and the second input, *v*, is a column vector with the same number of rows as the matrix *x*, then **miss** will check the first row of *x* for instances of the value in the first element of the column vector. It will check the second row of *x* for instances of the second element of the column vector, etc. For example:

```
    x = { 1 2 3 1,
          1 2 3 1,
          1 2 3 1 };
    v = { 1,
          2,
          3 };
    x_miss = miss(x, v);
```

will create:

```
    x_miss = . 2 3 .
             1 . 3 1
            1 2 . 1
```

If *v* is a row vector, then **miss** will check each column of *x* for entries that are equal to the corresponding entry in *v*. For example:

```
    x = { 1 2 3,
          1 2 3,
          1 2 3,
          4 5 6 };
    v = { 1 2 3 };
    x_miss = miss(x, v);
```

will create:

```
    x_miss = . . .
             . . .
            . . .
            4 5 6
```

See Also

counts, ismiss, maxc, maxindc, minc, minindc, missex, moment, packr, scalmiss, sortc

missex

Purpose

Converts numeric values to the missing value code according to the values given in a logical expression.

Format

y = **missex**(*x*, *mask*);

Input

x	NxK matrix.
mask	NxK logical matrix (matrix of 0's and 1's) that serves as a "mask" for *x*; the 1's in *mask* correspond to the values in *x* that are to be converted into missing values.

Output

y	NxK matrix that equals *x*, but with those elements that correspond to the 1's in *e* converted to missing.

Remarks

The matrix *e* will usually be created by a logical expression. For instance, to convert all numbers between 10 and 15 in *x* to missing, the following code could be used:

```
y = missex(x, (x .> 10) .and (x .< 15));
```

Note that "dot" operators MUST be used in constructing the logical expressions.

For complex matrices, the missing value code is defined as a missing value entry in the real part of the matrix. For complex *x*, then, **missex** replaces elements with a ". + 0i" value.

This function is like **miss**, but is more general in that a range of values can be converted into missings.

Example

```
//Set seed for repeatable random numbers
```

```
rndseed 49728424;

x = rndu(3,2);

//Logical expression
mask =(x .> .30) .and (x .< .60);
y = missex(x,mask);
```

After the code above:

```
       0.525  0.419          1  1              .      .
x =    0.869  0.973   mask = 0  0    y = 0.869  0.973
       0.021  0.357          0  1        0.021      .
```

A 3x2 matrix of uniform random numbers is created. All values in the interval (0.30, 0.60) are converted to missing.

Source

datatran.src

See Also

miss, missrv

moment

Purpose

Computes a cross-product matrix. This is the same as $x'x$.

Format

$y =$ **moment**(x, d);

Input

x	NxK matrix or M-dimensional array where the last two dimensions are NxK.
d	scalar, controls handling of missing values. *0* missing values will not be checked for. This is the fastest option. *1* "listwise deletion" is used. Any row that contains a missing value in any of its elements is excluded from the computation of the moment matrix. If every row in *x* contains missing values, then **`moment`**(*x*,*1*) will return a scalar zero. *2* "pairwise deletion" is used. Any element of *x* that is missing is excluded from the computation of the moment matrix. Note that this is seldom a satisfactory method of handling missing values, and special care must be taken in computing the relevant number of observations and degrees of freedom.

Output

y	KxK matrix or M-dimensional array where the last two dimensions are KxK, the cross-product of *x*.

Remarks

The fact that the moment matrix is symmetric is taken into account to cut execution time almost in half.

If *x* is an array, the result will be an array containing the cross-products of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10x4x4 array *x*, the resulting array *y* will contain the cross-products of each fo the 10 4x4 arrays contained in *x*, so *y*[*n*,.,.]=*x*[*n*,.,.]'*x*[*n*,.,.] for 1 <= n <= 10.

If there is no missing data then *d* = 0 should be used because it will be faster.

The **/** operator (matrix division) will automatically form a moment matrix (performing pairwise deletions if **trap 2** is set) and will compute the **ols** coefficients of a regression. However, it can only be used for data sets that are small enough to fit into a single matrix. In addition, the moment matrix and its inverse cannot be recovered if the **/** operator is used.

Example

```
xx = moment(x,2);
ixx = invpd(xx);
b = ixx*missrv(x,0)'y;
```

In this example, the regression of *y* on *x* is computed. The moment matrix (*xx*) is formed using the **moment** command (with pairwise deletion, since the second parameter is 2). Then *xx* is inverted using the **invpd** function. Finally, the **ols** coefficients are computed. **missrv** is used to emulate pairwise deletion by setting missing values to 0.

momentd

Purpose

Computes a moment (*x*'*x*) matrix from a **GAUSS** data set.

Format

```
m = momentd(dataset, vars);
```

Input

dataset	string, name of data set.
vars	Kx1 character vector, names of variables - or - Kx1 numeric vector, indices of columns. These can be any size subset of the variables in the data set, and can be in any order. If a scalar 0 is passed, all columns of the data set will be used.

Global Input

__con	scalar, default 1. 1 a constant term will be added. 0 no constant term will be added.
__miss	scalar, default 0. 0 there are no missing values (fastest). 1 do listwise deletion; drop an observation if any missings occur in it. 2 do pairwise deletion; this is equivalent to setting missings to 0 when calculating *m*.

__row	scalar, the number of rows to read per iteration of the read loop, default 0. If 0, the number of rows will be calculated internally. If you get an Insufficient memory error, or you want the rounding to be exactly the same between runs, you can set the number of rows to read before calling **momentd**.

Output

m	MxM matrix, where M = K + *__con*, the moment matrix constructed by calculating X'X where X is the data, with or without a constant vector of ones. Error handling is controlled by the low order bit of the trap flag. **trap 0** terminate with error message **trap 1** return scalar error code in *m* 33 too many missings 34 file not found

Example

```
    z = { age, pay, sex };
    m = momentd("freq",z);
```

Source

momentd.src

movingave

Purpose

Computes moving average of a series.

Format

```
y = movingave(x, d);
```

Input

x	NxK matrix.
d	scalar, order of moving average.

Output

y	NxK matrix, filtered series. The first *d*-1 rows of *x* are set to missing values.

Remarks

movingave is essentially a smoothing time series filter. The moving average is performed by column and thus it treats the NxK matrix as K time series of length N.

See Also

movingaveWgt, movingaveExpwgt

movingaveExpwgt

Purpose

Computes exponentially weighted moving average of a series.

Format

y = **`movingaveExpwgt`**(*x*, *d*, *p*);

Input

x	NxK matrix.
d	scalar, order of moving average.
p	scalar, smoothing coefficient where $0 > p > 1$.

Output

y	NxK matrix, filtered series. The first *d*-1 rows of *x* are set to missing values.

Remarks

`movingaveExpwgt` is smoothing time series filter using exponential weights. The moving average as performed by column and thus it treats the NxK matrix as K time series of length N.

See Also

movingaveWgt, movingave

movingaveWgt

Purpose

Computes weighted moving average of a series

Format

```
y = movingaveWgt(x, d, w);
```

Input

x	NxK matrix.
d	scalar, order of moving average.
w	*d*x1 vector, weights.

Output

y	NxK matrix, filtered series. The first *d*-1 rows of *x* are set to missing values.

Remarks

movingaveWgt is essentially a smoothing time series filter with weights. The moving average as performed by column and thus it treats the NxK matrix as K time series of length N.

See Also

movingave, movingaveExpwgt

msym

Purpose

Allows the user to set the symbol that **GAUSS** uses when missing values are converted to ASCII and vice versa.

Format

```
msym str;
```

Input

str	literal or ^string (up to 8 letters) which, if not surrounded by quotes, is forced to uppercase. This is the string to be printed for missing values. The default is `'.'`.

Remarks

The entire string will be printed out when converting to ASCII in `print` and **`printfm`** statements.

When converting ASCII to binary in `loadm` and `let` statements, only the first character is significant. In other words,

```
msym HAT;
```

will cause '*H*' to be converted to missing on input.

This does not affect **`writer`**, which outputs data in binary format.

Note that msym is a keyword and not a variable being assigned to, so there is no equals sign between msym and the string that is being passed to it.

Example

In the example below, you first create simulated data. The data represents the scores that a group of students received on a particular test and also the time that they took. For your calculations, you only want to consider data from students that completed the test in less than 80 minutes.

The code below replaces the scores from students that took more than 80 minutes with missing values. It uses the msym keyword to change the visual representation used for missing values from a '.' to a 'T'. Though, note that the underlying elements are still missing values, not character or string elements.

```
//Set seed for repeatable random numbers
rndseed 543124;

//Random integers with a mean of 70 and range of 20 to
//represent time taken for test
testTime = ceil(30 * rndu(10, 1)) + 60;

//Random integers with a mean of 1000 and a standard
//deviation of 10
score = ceil(10 * rndn(10, 1)) + 1000;

//Maximum allowed time for test
maxTime = 80;

//Create a mask for times greater than maxTime
mask = testTime .> maxTime;

//Set scores to be missing values if testTime is greater
//than maxTime
mScores = missex(score, mask);
```

```
//Set missing values to print as 'T' to represent that the
//score was invalid because the student took too much time
msym "T";

format /rd 4,0;
print mScores;
```

The code above will return:

```
  T
1010
 997
1002
 985
 997
1007
 995
   T
   T
```

See Also

print, printfm

n

new

Purpose

Erases everything in memory including the symbol table; closes all open files as well as the auxiliary output and turns the window on if it was off; also allows the size of the new symbol table and the main program space to be specified.

Format

```
new;
new nos;
```

Input

nos	scalar, optional input which indicates the maximum number of global symbols allowed.

Remarks

Procedures, user-defined functions, and global matrices, strings, and string arrays are all global symbols.

If you would like your user-defined procedures to not be cleared after a `new` statement, you can either add them to a **GAUSS Library** or create a file in your GAUSSHOME directory with the same name as your procedure and a `.g` file extension. This file `.g` file should only contain your procedure.

This command can be used with arguments as the first statement in a program to clear the symbol table and to allocate only as much space for program code as your program actually needs. When used in this manner, the auxiliary output will not be closed. This will allow you to open the auxiliary output from the command level and run a program without having to remove the `new` at the beginning of the program. If this command is not the first statement in your program, it will cause the program to terminate.

Example

```
    new; /* clear global symbols. */
```

```
    new 300; /* clear global symbols,set maximum
             ** number of global symbols to 300,
             ** and leave program space unchanged.
             */
```

See Also

clear, delete, output

nextindex

Purpose

Returns the index of the next element or subarray in an array.

Format

ni = **nextindex**(*i*, *o*);

Input

i	Mx1 vector of indices into an array, where M<=N.
o	Nx1 vector of orders of an N-dimensional array.

Output

ni	Mx1 vector of indices, the index of the next element or subarray in the array corresponding to *o*.

Remarks

nextindex will return a scalar error code if the index cannot be incremented.

Example

```
//Dimensions of an array
orders = { 3, 4, 5, 6, 7);

//Starting index
ind = { 2, 3, 5 };

//Return the index for the next element
ind = nextindex(ind,orders);
```

After the code above, *ind* will be equal to:

```
    2
    4
    1
```

In this example, **`nextindex`** incremented *`ind`* to index the next 6x7 subarray in array *`a`*.

Using the same data from above, a subsequent call to **`nextindex`**:

```
    ind = nextindex(ind,orders);
```

will assign *`ind`* to be equal to:

```
    2
    4
    2
```

See Also

previousindex, loopnextindex, walkindex

nextn, nextnevn

Purpose

Returns allowable matrix dimensions for computing FFT's.

Format

n = **nextn**(*n0*);
n = **nextnevn**(*n0*);

Input

n0	scalar, the length of a vector or the number of rows or columns in a matrix.

Output

n	scalar, the next allowable size for the given dimension for computing an FFT or RFFT. $n > n0$.

Remarks

nextn and **nextnevn** determine allowable matrix dimensions for computing FFT's. The Temperton FFT routines (see table below) can handle any matrix whose dimensions can be expressed as:

$$2^p x 3^q x 5^r x 7^s$$

where *p*, *q* and *r* are nonnegative integers and *s* is equal to 0 or 1.

with one restriction: the vector length or matrix column size must be even (*p* must be positive) when computing RFFT's.

fftn, etc., automatically pad matrices (with zeros) to the next allowable dimensions; **nextn** and **nextnevn** are provided in case you want to check or fix matrix sizes yourself.

Use the following table to determine what to call for a given function and matrix:

FFT	Vector	Matrix	Matrix
Function	Length	Rows	Columns

fftn	**nextn**	**nextn**	**nextn**
rfftn	**nextnevn**	**nextn**	**nextnevn**
rfftnp	**nextnevn**	**nextn**	**nextnevn**

Example

```
    n = nextn(456);
```

The code above will assign *n* to be equal to 480.

Source

`optim.src`

See Also

fftn, optn, optnevn, rfftn, rfftnp

nextwind

Purpose

Set the current graphic panel to the next available graphic panel. Note: This function is for use with the deprecated PQG graphics. For similar functionality use **`plotLayout`** instead.

Library

pgraph

Format

nextwind;

Remarks

This function selects the next available graphic panel to be the current graphic panel. This is the graphic panel in which the next graph will be drawn.

See the discussion on using graphic panels in **Tiled Graphic Panels**, Section 1.0.1.

Source

`pwindow.src`

See Also

endwind, begwind, setwind, getwind, makewind, window

null

Purpose

Computes an orthonormal basis for the (right) null space of a matrix.

Format

b = **null**(*x*);

Input

x	NxM matrix.

Output

b MxK matrix, where K is the nullity of *x*, such that:

```
x * b = 0 //NxK matrix of 0's
```

and

```
b'b = I  //MxM identity matrix
```

The error returns are returned in *b*:

error code	reason
1	there is no null space
2	*b* is too large to return in a single matrix

Use **scalerr** to test for error returns.

Remarks

The orthogonal complement of the column space of *x*' is computed using the QR decomposition. This provides an orthonormal basis for the null space of *x*.

Example

```
let x[2,4] = 2 1 3 -1
             3 5 1  2;

b = null(x);
z = x*b;
i = b'b;
```

After the code above:

```
             -0.804   0.142
      b =     0.331 -0.473   z = 0   0   i = 1   0
              0.473   0.331      0   0       0   1
              0.142   0.804
```

Source

`null.src`

Globals

_qrdc, *_qrsl*

null1

Purpose

Computes an orthonormal basis for the (right) null space of a matrix and writes it to a **GAUSS** dataset.

Format

nu = **null1**(*x*, *dataset*);

Input

x	NxM matrix.
dataset	string, the name of a data set **null1** will write.

Output

nu	scalar, the nullity of *x*.

Remarks

null1 computes an MxK matrix *b*, where K is the nullity of *x*, such that:

```
    x * b = 0 //NxK matrix of 0's
```

and

```
    b'b = I   //MxM identity matrix
```

The transpose of *b* is written to the data set named by *dataset*, unless the nullity of *x* is zero. If *nu* is zero, the data set is not written.

Source

null.src

Globals

_qrdc, *_qrsl*

numCombinations

Purpose

Computes number of combinations of *n* things taken *k* at a time.

Format

y = **numCombinations**(*n*, *k*);

Input

n	scalar.
k	scalar.

Output

y	scalar, number of combinations of *n* things take *k* at a time.

Remarks

To calculate all of the combinations, use the function **combinate**.

Example

```
y = numCombinations(25,5);

print y;
```

The code above, returns:

```
 53130.0000
```

See Also

combinate, combinated

ols

Purpose

Computes a least squares regression.

Format

{ *vnam*, *m*, *b*, *stb*, *vc*, *stderr*, *sigma*, *cx*, *rsq*, *resid*, *dwstat* } = **ols**(*dataset*, *depvar*, *indvars*)

Input

dataset	string, name of data set or null string. If *dataset* is a null string, the procedure assumes that the actual data has been passed in the next two arguments.
depvar	If *dataset* contains a string: string, name of dependent variable - or - scalar, index of dependent variable. If scalar 0, the last column of the data set will be used. If *dataset* is a null string or 0:

	Nx1 vector, the dependent variable.
indvars	If *dataset* contains a string: Kx1 character vector, names of independent variables - or - Kx1 numeric vector, indices of independent variables. These can be any size subset of the variables in the data set and can be in any order. If a scalar 0 is passed, all columns of the data set will be used except for the one used for the dependent variable. If *dataset* is a null string or 0: NxK matrix, the independent variables.

Global Input

Defaults are provided for the following global input variables, so they can be ignored unless you need control over the other options provided by this procedure.

___*altnam*	character vector, default 0. This can be a (K+1)x1 or (K+2)x1 character vector of alternate variable names for the output. If ___*con* is 1, this must be (K+2)x1. The name of the dependent variable is the last element.
___*con*	scalar, default 1. 1 a constant term will be added, D = K+1. 0 no constant term will be added, D = K.

	A constant term will always be used in constructing the moment matrix *m*.	
__*miss*	scalar, default 0.	
	0	there are no missing values (fastest).
	1	listwise deletion, drop any cases in which missings occur.
	2	pairwise deletion, this is equivalent to setting missings to 0 when calculating *m*. The number of cases computed is equal to the total number of cases in the data set.
__*olsalg*	string, default "cholup." Selects the algorithm used for computing the parameter estimates. The default Cholesky update method is more computationally efficient; however, accuracy can suffer for poorly conditioned data. For higher accuracy, set __*olsalg* to either *qr* or *svd*.	
	qr	Solves for the parameter estimates using a *qr* decomposition.
	svd	Solves for the paramer estimates using a singular value decomposition.
__*output*	scalar, default 1.	
	1	print the statistics.
	0	do not print statistics.
__*row*	scalar, the number of rows to read per iteration of the read	

	loop. Default 0. If 0, the number of rows will be calculated internally. If you get an Insufficient memory error while executing **`ols`**, you can supply a value for `__row` that works on your system. The answers may vary slightly due to rounding error differences when a different number of rows is read per iteration. You can use `__row` to control this if you want to get exactly the same rounding effects between several runs.
`_olsres`	scalar, default 0. 1 compute residuals (`resid`) and Durbin-Watson statistic (`dwstat`). 0 `resid` = 0, `dwstat` = 0.

Output

`vnam`	(K+2)x1 or (K+1)x1 character vector, the variable names used in the regression. If a constant term is used, this vector will be (K+2)x1, and the first name will be "CONSTANT". The last name will be the name of the dependent variable.
`m`	MxM matrix, where M = K+2, the moment matrix constructed by calculating `x'x` where `x` is a matrix containing all useable observations and having columns in the order: 1.0 (constant) indvars (independent variables) depvar (dependent variable) A constant term is always used in computing `m`.

b	Dx1 vector, the least squares estimates of parameters Error handling is controlled by the low order bit of the trap flag.

trap 0	terminate with error message	
trap 1	return scalar error code in *b*	
	30	system singular
	31	system underdetermined
	32	same number of columns as rows
	33	too many missings
	34	file not found
	35	no variance in an independent variable

The system can become underdetermined if you use listwise deletion and have missing values. In that case, it is possible to skip so many cases that there are fewer useable rows than columns in the data set.

stb	Kx1 vector, the standardized coefficients.
vc	DxD matrix, the variance-covariance matrix of estimates.
stderr	Dx1 vector, the standard errors of the estimated parameters.
sigma	scalar, standard deviation of residual.
cx	(K+1)x(K+1) matrix, correlation matrix of variables with the dependent variable as the last column.

rsq	scalar, R square, coefficient of determination.
resid	residuals, *resid* = *y* - *x* * *b*.
	If *_olsres* = 1, the residuals will be computed.
	If the data is taken from a data set, a new data set will be created for the residuals, using the name in the global string variable *_olsrnam*. The residuals will be saved in this data set as an Nx1 column. The *resid* return value will be a string containing the name of the new data set containing the residuals.
	If the data is passed in as a matrix, the *resid* return value will be the Nx1 vector of residuals.
dwstat	scalar, Durbin-Watson statistic.

Remarks

For poorly conditioned data the default setting for *__olsalg*, using the Cholesky update, may produce only four or five digits of accuracy for the parameter estimates and standard error. For greater accuracy, use either the *qr* or singular value decomposition algorithm by setting *__olsalg* to *qr* or *svd*. If you are unsure of the condition of your data, set *__olsalg* to *qr*.

No output file is modified, opened, or closed by this procedure. If you want output to be placed in a file, you need to open an output file before calling **ols**.

Example

```
    y = { 2,
          3,
          1,
          7,
          5 };

    x = { 1 3 2,
          2 3 1,
          7 1 7,
          5 3 1,
          3 5 5 };

    output file = ols.out reset;
    call ols(0,y,x);
    output off;
```

In this example, the output from **`ols`** is put into a file called `ols.out` as well as being printed to the window. This example will compute a least squares regression of *`y`* on *`x`*. The return values are discarded by using a `call` statement.

```
    data = "olsdat";
    depvar = { score };
    indvars = { region, age, marstat };
    _olsres = 1;
    output file = lpt1 on;
    { nam,m,b,stb,vc,std,sig,cx,rsq,resid,dbw } = ols(data,depv
    ar,indvars);
    output off;
```

In this example, the data set `olsdat.dat` is used to compute a regression. The dependent variable is *`score`*. The independent variables are: *`region`*, *`age`*, and *`marstat`*. The residuals and Durbin-Watson statistic will be computed. The output will be sent to the printer as well as the window and the returned values are assigned to variables.

Source

`ols.src`

See Also

olsqr

olsmt

Purpose

Computes a least squares regression.

Format

oout = **olsmt**(*oc0*, *dataset*, *depvar*, *indvars*);

Input

oc0	instance of an **olsmtControl** structure containing the following members:	
	oc0.altnam	character vector, default 0. This can be a (K+1)x1 or (K+2)x1 character vector of alternate variable names for the output. If oc0.con is 1, this must be (K+2)x1. The name of the dependent variable is the last element.
	oc0.con	scalar, default 1. 1 a constant term will be added, D =

		K+1.
	0	no constant term will be added, D = K.
	A constant term will always be used in constructing the moment matrix m.	
oc0.miss	scalar, default 0.	
	0	there are no missing values (fastest).
	1	listwise deletion, drop any cases in which missings occur.
	2	pairwise deletion, this is equivalent to setting missings to 0 when calculating *m*. The number of cases computed is equal to the total number of cases in the data set.
oc0.row	scalar, the number of rows to read per iteration of the read loop. Default 0.	
	If 0, the number of rows will be calculated internally. If you get an Insufficient memory error message while executing **olsmt**, you can supply a value for *oc0.row* that works on your system.	
	The answers may vary slightly due to rounding error differences when a different number of rows is read per iteration. You can use *oc0.row* to control this if you want to get exactly the same rounding effects between	

	several runs.
`oc0.vpad`	scalar, default 1. If 0, internally created variable names are not padded to the same length (e.g. "X1, X2,..., X10"). If 1, they are padded with zeros to the same length (e.g., "X01, X02,..., X10").
`oc0.output`	scalar, default 1. 1 print the statistics. 0 do not print statistics.
`oc0.res`	scalar, default 0. 1 compute residuals (`resid`) and Durbin-Watson statistic (`dwstat.`) 0 `oout.resid` = 0, `oout.dwstat` = 0.
`oc0.rnam`	string, default "_olsmtres". If the data is taken from a data set, a new data set will be created for the residuals, using the name in `oc0.rnam`.
`oc0.maxvec`	scalar, default 20000. The largest number of elements allowed in any one matrix.
`oc0.fcmptol`	scalar, default 1e-12. Tolerance used to fuzz the comparison

		operations to allow for round off error.
	`oc0.alg`	string, default "cholup". Selects the algorithm used for computing the parameter estimates. The default Cholesky update method is more computationally efficient. However, accuracy can suffer for poorly conditioned data. For higher accuracy set `oc0.alg` to either `qr` or `svd`. `qr` Solves for the parameter estimates using a `qr` decomposition. `svd` Solves for the paramer estimates using a singular value decomposition.
`dataset`	string, name of data set or null string. If `dataset` is a null string, the procedure assumes that the actual data has been passed in the next two arguments.	
`depvar`	If `dataset` contains a string: string, name of dependent variable - or - scalar, index of dependent variable. If scalar 0, the last column of the data set will be used. If `dataset` is a null string or 0: Nx1 vector, the dependent variable.	
`indvars`	If `dataset` contains a string:	

Kx1 character vector, names of independent variables
- or -
Kx1 numeric vector, indices of independent variables.

These can be any size subset of the variables in the data set and can be in any order. If a scalar 0 is passed, all columns of the data set will be used except for the one used for the dependent variable.

If `dataset` is a null string or 0:

NxK matrix, the independent variables.

Output

oout	instance of an **olsmtOut** structure containing the following members:	
	oout.vnam	(K+2)x1 or (K+1)x1 character vector, the variable names used in the regression. If a constant term is used, this vector will be (K+2)x1, and the first name will be "CONSTANT". The last name will be the name of the dependent variable.
	oout.m	MxM matrix, where M = K+2, the moment matrix constructed by calculating X' X where X is a matrix containing all useable observations and having columns in the order:

1.0	indvars	depvar
(constant)	(independent	(dependent

	variables)	variable)

A constant term is always used in computing *m*.

`oout.b`	Dx1 vector, the least squares estimates of parameters

Error handling is controlled by the low order bit of the trap flag.

trap 0	terminate with error message	
trap 1	return scalar error code in `b`	
	30	system singular
	31	system underdetermined
	32	same number of columns as rows
	33	too many missings
	34	file not found
	35	no variance in an independent variable

The system can become underdetermined if you use listwise deletion and have missing values. In that case, it is possible to skip so many cases that there are fewer useable rows than columns in the data set.

`oout.stb`	Kx1 vector, the standardized coefficients.
`oout.vc`	DxD matrix, the variance-covariance matrix of estimates.
`oout.stderr`	Dx1 vector, the standard errors of the estimated parameters.
`oout.sigma`	scalar, standard deviation of residual.
`oout.cx`	(K+1)x(K+1) matrix, correlation matrix of variables with the dependent variable as the last column.
`oout.rsq`	scalar, R square, coefficient of determination.
`oout.resid`	residuals, `oout.resid = y - x * oout.b`. If `oc0.olsres` = 1, the residuals will be computed. If the data is taken from a data set, a new data set will be created for the residuals, using the name in `oc0.rnam`. The residuals will be saved in this data set as an Nx1 column. The `oout.resid` return value will be a string containing the name of the new data set containing the residuals. If the data is passed in as a matrix, the `oout.resid` return value will be the Nx1 vector of residuals.
`oout.dwstat`	scalar, Durbin-Watson statistic.

Remarks

For poorly conditioned data the default setting for *oc0*.alg, using the Cholesky update, may produce only four or five digits of accuracy for the parameter estimates and standard error. For greater accuracy, use either the *qr* or singular value decomposition algorithm by setting *oc0*.alg to *qr* or *svd*. If you are unsure of the condition of your data, set *oc0*.alg to *qr*.

No output file is modified, opened, or closed by this procedure. If you want output to be placed in a file, you need to open an output file before calling **olsmt**.

Example

```
#include olsmt.sdf
struct olsmtControl oc0;
struct olsmtOut oOut;
oc0 = olsmtControlCreate;

y = { 2,
      3,
      1,
      7,
      5 };

x = { 1 3 2,
      2 3 1,
      7 1 7,
      5 3 1,
      3 5 5 };

output file = olsmt.out reset;
oOut = olsmt(oc0,0,y,x);
output off;
```

In this example, the output from **olsmt** is put into a file called `olsmt.out` as well as being printed to the window. This example will compute a least squares regression of *y* on *x*.

```
    #include olsmt.sdf
    struct olsmtControl oc0;
    struct olsmtOut oOut;
    oc0 = olsmtControlCreate;

    data = "olsdat";
    depvar = { score };
    indvars = { region,age,marstat };
    oc0.res = 1;
    output file = lpt1 on;
    oOut = olsmt(oc0,data,depvar,indvars);
    output off;
```

In this example, the data set `olsdat.dat` is used to compute a regression. The dependent variable is *`score`*. The independent variables are: *`region`*, *`age`*, and *`marstat`*. The residuals and Durbin-Watson statistic will be computed. The output will be sent to the printer as well as the window and the returned values are assigned to variables.

Source

`olsmt.src`

See Also

olsmtControlCreate, olsqrmt

olsmtControlCreate

Purpose

Creates default **olsmtControl** structure.

Include

```
olsmt.sdf
```

Format

c = **`olsmtControlCreate`**();

Output

c	instance of an **olsmtControl** structure with members set to default values.

Example

Since structures are strongly typed in **GAUSS**, each structure must be declared before it can be used.

```
// declare 'ctl' as an olsmtControl structure
struct olsmtControl ctl;

// initialize structure 'ctl'
ctl = olsmtControlCreate;
```

The members of the **olsmtControl** structure and their default values are described in the manual entry for **`olsmt`**.

Source

```
olsmt.src
```

See Also

olsmt

olsqr

Purpose

Computes OLS coefficients using QR decomposition.

Format

b = **`olsqr`**(*y*, *x*);

Input

y	Nx1 vector containing dependent variable.
x	NxP matrix containing independent variables.

Output

b	Px1 vector of least squares estimates of regression of *y* on *x*. If *x* does not have full rank, then the coefficients that cannot

be estimated will be zero.

Remarks

This provides an alternative to y/x for computing least squares coefficients.

This procedure is slower than the **/** operator. However, for near singular matrices it may produce better results.

olsqr handles matrices that do not have full rank by returning zeros for the coefficients that cannot be estimated.

Example

```
    A = rndn(4,4);
    b = rndn(4,1);
    x = olsqr(b,A);
```

See Also

ols, olsqr2, orth, qqr

olsqr2

Purpose

Computes OLS coefficients, residuals, and predicted values using the QR decomposition.

Format

```
{ b, r, p } = olsqr2(y, x);
```

Input

y	Nx1 vector containing dependent variable.
x	NxP matrix containing independent variables.

Output

b	Px1 vector of least squares estimates of regression of *y* on *x*. If *x* does not have full rank, then the coefficients that cannot be estimated will be zero.
r	Px1 vector of residuals. (*r* = *y* - *x***b*)
p	Px1 vector of predicted values. (*p* = *x***b*)

Remarks

This provides an alternative to *y*/*x* for computing least squares coefficients.

This procedure is slower than the **/** operator. However, for near singular matrices, it may produce better results.

olsqr2 handles matrices that do not have full rank by returning zeros for the coefficients that cannot be estimated.

See Also

olsqr, orth, qqr

olsqrmt

Purpose

Computes OLS coefficients using QR decomposition.

Format

```
b = olsqrmt(y, x,  tol);
```

Input

y	Nx1 vector containing dependent variable.
x	NxP matrix containing independent variables.
tol	scalar, the tolerance for testing if diagonal elements are approaching zero. The default value is 10^{-14}.

Output

b	Px1 vector of least squares estimates of regression of *y* on *x*. If *x* does not have full rank, then the coefficients that cannot be estimated will be zero.

Remarks

This provides an alternative to *y*/*x* for computing least squares coefficients.

This procedure is slower than the **/** operator. However, for near singular matrices it may produce better results.

olsqrmt handles matrices that do not have full rank by returning zeros for the coefficients that cannot be estimated.

Source

`olsmt.src`

See Also

olsmt, olsqr2

ones

Purpose

Creates a matrix of ones.

Format

y = **ones**(*r*, *c*);

Input

r	scalar, number of rows.
c	scalar, number of columns.

Output

y	*r* x *c* matrix of ones.

Remarks

Noninteger arguments will be truncated to an integer.

Example

```
    x = ones(3,2);
```

The code above assigns *x* to be equal to:

```
    1.0000000        1.0000000
    1.0000000        1.0000000
    1.0000000        1.0000000
```

See Also

zeros, eye

open

Purpose

Opens an existing **GAUSS** data file.

Format

```
    open fh=filename;
    open fh=filename for mode;
    open fh=filename for mode varindxi offs;
```

Input

filename	literal or ^string. *filename* is the name of the file on the disk. The name can include a path if the directory to be used is not the current directory. This filename will automatically be given the extension `.dat`. If an extension is specified, the `.dat` will be overridden. If the file is an `.fmt` matrix file, the extension must be explicitly given. If the name of the file is to be taken from a string variable, the name of the string must be preceded by the ^ (caret) operator.
mode	literal, the modes supported with the optional `for` subcommand are:
	`read` — This is the default file opening mode and will be the one used if none is specified. Files opened in this mode cannot be written to. The pointer is set to the beginning of the file and the **`writer`** function is disabled for files opened in this way. This is the only mode available for matrix files (`.fmt`), which are always written in one piece with the `save` command.
	`append` — Files opened in this mode cannot be read.

		The pointer will be set to the end of the file so that a subsequent write to the file with the **`writer`** function will add data to the end of the file without overwriting any of the existing data in the file. The **`readr`** function is disabled for files opened in this way. This mode is used to add additional rows to the end of a file.
	`update`	Files opened in this mode can be read from and written to. The pointer will be set to the beginning of the file. This mode is used to make changes in a file.
`offs`	scalar, offset added to "index variables."	

The optional **`varindxi`** subcommand tells **GAUSS** to create a set of global scalars that contain the index (column position) of the variables in a **GAUSS** data file. These "index variables" will have the same names as the corresponding variables in the data file but with "i" added as a prefix. They can be used inside index brackets, and with functions like **`submat`** to access specific columns of a matrix without having to remember the column position.

The optional *`offs`* argument is an offset that will be added to the index variables. This is useful if data from multiple files are concatenated horizontally in one matrix. It can be any scalar expression. The default is 0.

The index variables are useful for creating submatrices of specific variables without requiring that the positions of the variables be known. For instance, if there are two variables,

	`xvar` and `yvar` in the data set, the index variables will have the names `ixvar`, `iyvar`. If `xvar` is the first column in the data file, and `yvar` is the second, and if no offset, `offs`, has been specified, then `ixvar` and `iyvar` will equal 1 and 2 respectively. If an offset of 3 had been specified, then these variables would be assigned the values 4 and 5 respectively. The **`varindxi`** option cannot be used with `.fmt` matrix files because no column names are stored with them. If **`varindxi`** is used, **GAUSS** will ignore the Undefined symbol error for global symbols that start with "i". This makes it much more convenient to use index variables because they don't have to be cleared before they are accessed in the program. Clearing is otherwise necessary because the index variables do not exist until execution time when the data file is actually opened and the names are read in from the header of the file. At compile time a statement like: `y=x[.,ixvar];` will be illegal if the compiler has never heard of `ixvar`. If **`varindxi`** is used, this error will be ignored for symbols beginning with "i". Any symbols that are accessed before they have been initialized with a real value will be trapped at execution time with a Variable not initialized error.

Output

fh	scalar, file handle. *fh* is the file handle which will be used by most commands to refer to the file within **GAUSS**. This file handle is actually a scalar containing an integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the `open` command is executed. If the file was not successfully opened, the file handle will be set to -1.

Remarks

The file must exist before it can be opened with the `open` command. To create a new file, see `create` or `save`.

A file can be opened simultaneously under more than one handle. See the second example following.

If the value that is in the file handle when the `open` command begins to execute matches that of an already open file, the process will be aborted and a File already open message will be given. This gives you some protection against opening a second file with the same handle as a currently open file. If this happens, you would no longer be able to access the first file.

It is important to set unused file handles to zero because both `open` and `create` check the value that is in a file handle to see if it matches that of an open file before they proceed with the process of opening a file. This should be done with **`close`** or `closeall`.

Example

```
fname = "/data/rawdat";
open dt = ^fname for append;

if dt == -1;
   print "File not found";
   end;
endif;
y = writer(dt,x);
if y /= rows(x);
   print "Disk Full";
   end;
endif;
```

```
    dt = close(dt);
```

In the example above, the existing data set `/data/rawdat.dat` is opened for appending new data. The name of the file is in the string variable *`fname`*. In this example the file handle is tested to see if the file was opened successfully. The matrix *`x`* is written to this data set. The number of columns in *`x`* must be the same as the number of columns in the existing data set. The first row in *`x`* will be placed after the last row in the existing data set. The **`writer`** function will return the number of rows actually written. If this does not equal the number of rows that were attempted, then the disk is probably full.

```
    open fin = mydata for read;
    open fout = mydata for update;

    do until eof(fin);
       x = readr(fin,100);
       x[.,1 3] = ln(x[.,1 3]);
       call writer(fout,x);
    endo;

    closeall fin,fout;
```

In the above example, the same file, `mydata.dat`, is opened twice with two different file handles. It is opened for read with the handle *`fin`*, and it is opened for update with the handle *`fout`*. This will allow the file to be transformed in place without taking up the extra space necessary for a separate output file. Notice that *`fin`* is used as the input handle and *`fout`* is used as the output handle. The loop will terminate as soon as the input handle has reached the end of the file. Inside the loop the file is read into a matrix called *`x`* using the input handle, the data are transformed (columns 1 and 3 are replaced with their natural logs), and the transformed data is written back out using the output handle. This type of operation works fine as long as the total number of rows and columns does not change.

The following example assumes a data file named `dat1.dat` that has the variables: *`visc`*, *`temp`*, *`lub`*, and *`rpm`*:

```
     open f1 = dat1 varindxi;
     dtx = readr(f1,100);
     x = dtx[.,irpm ilub ivisc];
     y = dtx[.,itemp];
     call seekr(f1,1);
```

In this example, the data set `dat1.dat` is opened for reading (the `.dat` and the **`for read`** are implicit). **`varindxi`** is specified with no constant. Thus, index variables are created that give the positions of the variables in the data set. The first 100 rows of the data set are read into the matrix *`dtx`*. Then, specified variables in a specified order are assigned to the matrices *`x`* and *`y`* using the index variables. The last line uses the **`seekr`** function to reset the pointer to the beginning of the file.

```
     open q1 = c:dat1 varindx;
     open q2 = c:dat2 varindx colsf(q1);
     nr = 100;
     y = readr(q1,nr)~readr(q2,nr);
     closeall q1,q2;
```

In this example, two data sets are opened for reading and index variables are created for each. A constant is added to the indices for the second data set (*`q2`*), equal to the number of variables (columns) in the first data set (*`q1`*). Thus, if there are three variables *`x1`*, *`x2`*, *`x3`* in *`q1`*, and three variables *`y1`*, *`y2`*, *`y3`* in *`q2`*, the index variables that were created when the files were opened would be *`ix1`*, *`ix2`*, *`ix3`*, *`iy1`*, *`iy2`*, *`iy3`*. The values of these index variables would be 1, 2, 3, 4, 5, 6, respectively. The first 100 rows of the two data sets are read in and concatenated to produce the matrix *`y`*. The index variables will thus give the correct positions of the variables in *`y`*.

```
     open fx = x.fmt;
     rf = rowsf(fx);
```

```
    sampsize = round(rf*0.1);
    rndsmpx = zeros(sampsize,colsf(fx));

    for(1, sampsize, 1);
       r = ceil(rndu(1,1)*rf);
       call seekr(fx,r);
       rndsmpx[i,.] = readr(fx,1);
    endfor;

    fx = close(fx);
```

In this example, a 10% random sample of rows is drawn from the matrix file `x.fmt` and put into the matrix **`rndsmpx`**. Note that the extension `.fmt` must be specified explicitly in the `open` statement. The **`rowsf`** command is used to obtain the number of rows in `x.fmt`. This number is multiplied by 0.10 and the result is rounded to the nearest integer; this yields the desired sample size. Then random integers (r) in the range 1 to *`rf`* are generated. **`seekr`** is used to locate to the appropriate row in the matrix, and the row is read with **`readr`** and placed in the matrix **`rndsmpx`**. This is continued until the complete sample has been obtained.

See Also

dataopen, create, close, closeall, readr, writer, seekr, eof

optn, optnevn

Purpose

Returns optimal matrix dimensions for computing FFT's.

Format

```
n = optn(n0);
n = optnevn(n0);
```

Input

n0	scalar, the length of a vector or the number of rows or columns in a matrix.

Output

n	scalar, the next optimal size for the given dimension for computing an FFT or RFFT. $n > n0$.

Remarks

optn and **optnevn** determine optimal matrix dimensions for computing FFT's. The Temperton FFT routines (see table following) can handle any matrix whose dimensions can be expressed as:

$$2^p x 3^q x 5^r x 7^s$$

where *p*, *q* and *r* are nonnegative integers and *s* is equal to 0 or 1.

with one restriction: the vector length or matrix column size must be even (*p* must be positive) when computing RFFT's.

fftn, etc., pad matrices to the next allowable dimensions; however, they generally run faster for matrices whose dimensions are highly composite numbers, that is, products of several factors (to various powers), rather than powers of a single factor. For example, even though it is bigger, a 33600x1 vector can compute as much as 20% faster than a 32768x1 vector, because 33600 is a highly composite number, $2^6 * 3 * 5^2 * 7$, whereas 32768 is a simple power of 2, 2^{15}. **optn** and **optnevn** are provided so you can take advantage of this fact by hand-sizing matrices to optimal dimensions before computing the FFT.

Use the following table to determine what to call for a given function and matrix:

FFT	Vector	Matrix	Matrix
Function	Length	Rows	Columns
fftn	**optn**	**optn**	**optn**
rfftn	**optnevn**	**optn**	**optnevn**
rfftnp	**optnevn**	**optn**	**optnevn**

Example

```
    n = optn(231);
```

The above code assigns *n* to be equal to 240.

See Also

fftn, nextn, nextnevn, rfftn, rfftnp

orth

Purpose

Computes an orthonormal basis for the column space of a matrix.

Format

y = **orth**(*x*);

Input

x	NxK matrix.

Global Input

_orthtol	scalar, the tolerance for testing if diagonal elements are approaching zero. The default is 1.0e-14.

Output

y	NxL matrix such that *y*'*y* = **eye**(*L*) and whose columns span the same space as the columns of *x*; L is the rank of *x*.

Example

```
x = { 6 5 4,
      2 7 5 };

y = orth(x);
```

After the code above:

```
    y = -0.58123819       -0.81373347    y'y = 1   0
        -0.81373347        0.58123819          0   1
```

Source

qqr.src

See Also

qqr, olsqr

output

Purpose

This command makes it possible to direct the output of `print` statements to two different places simultaneously. One output device is always the window or standard output. The other can be selected by the user to be any disk file or other suitable output device such as a printer.

Format

```
output file=filename
output file=filename [on|off|reset];
```

Input

filename	literal or ^string. The *file=filename* subcommand selects the file or device to which output is to be sent.

If the name of the file is to be taken from a string variable, the name of the string must be preceded by the ^ (caret) operator.

The default file name is `output.out`.

`on, off, reset`	literal, mode flag:	
	`on`	opens the auxiliary output file or device and causes the results of all `print` statements to be sent to that file or device. If the file already exists, it will be opened for appending. If the file does not already exist, it will be created.
	`off`	closes the auxiliary output file and turns off the auxiliary output.
	`reset`	similar to the *`on`* subcommand, except that it always creates a new file. If the file already exists, it will be destroyed and a new file by that name will be created. If it does not exist, it will be created.

Remarks

After you have written to an output file you have to close the file before you can print it or edit it with the **GAUSS** editor. Use

```
    output off;
```

The selection of the auxiliary output file or device remains in effect until a new selection is made, or until you get out of **GAUSS**. Thus, if a file is named as the output device in one program, it will remain the output device in subsequent programs until a new *file=filename* subcommand is encountered.

The command

```
     output file=filename;
```

will select the file or device but will not open it. A subsequent **output on** or **output reset** will open it and turn on the auxiliary output.

The command **output off** will close the file and turn off the auxiliary output. The filename will remain the same. A subsequent **output on** will cause the file to be opened again for appending. A subsequent **output reset** will cause the existing file to be destroyed and then recreated and will turn on the auxiliary output.

The command `output` by itself will cause the name and status (i.e., open or closed) of the current auxiliary output file to be printed to the window.

The output to the console can be turned off and on using the **screen off** and **screen on** commands. Output to the auxiliary file or device can be turned off or on using the **output off** or **output on** command. The defaults are **screen on** and **output off**.

The auxiliary file or device can be closed by an explicit **output off** statement, by an `end` statement, or by an interactive `new` statement. However, a `new` statement at the beginning of a program will not close the file. This allows programs with `new` statements in them to be run without reopening the auxiliary output file.

If a program sends data to a disk file, it will execute much faster if the window is off.

The `outwidth` command will set the line width of the output file. The default is 80.

Example

```
     output file = out1.out on;
```

This statement will open the file `out1.out` and will cause the results of all subsequent `print` statements to be sent to that file. If `out1.out` already exists, the new output will be appended.

```
     output file = out2.out;
     output on;
```

This is equivalent to the previous example.

```
     output reset;
```

This statement will create a new output file using the current filename. If the file already exists, any data in it will be lost.

```
     output file = mydata.asc reset;
     screen off;
     format /m1/rz 1,8;
     open fp = mydata;

     do until eof(fp);
        print readr(fp,200);;
     endo;

     fp = close(fp);
     end;
```

The program above will write the contents of the **GAUSS** file `mydata.dat` into an ASCII file called `mydata.asc`. If there had been an existing file by the name of `mydata.asc`, it would have been overwritten.

The */m1* parameter in the `format` statement in combination with the `;;` at the end of the `print` statement will cause one carriage return/line feed pair to be written at the beginning of each row of the output file. There will not be an extra line feed added at the end of each 200 row block.

The `end` statement above will automatically perform **`output off`** and **`screen on`**.

See Also

outwidth, screen, end, new

outtyp (dataloop)

Purpose

Specifies the precision of the output data set.

Format

```
outtyp num_constant;
```

Input

num_constant	scalar, precision of output data set.

Remarks

num_constant must be 2, 4, or 8, to specify integer, single precision, or double precision, respectively.

If `outtyp` is not specified, the precison of the output data set will be that of the input data set. If character data is present in the data set, the precision will be forced to double.

Example

```
    outtyp 8;
```

outwidth

Purpose

Specifies the width of the auxiliary output.

Format

```
outwidth n;
```

Input

n	scalar, width of auxilary output.

Remarks

n specifies the width of the auxiliary output in columns (characters). After printing *n* characters on a line, **GAUSS** will output a line feed.

If a matrix is being printed, the line feed sequence will always be inserted between separate elements of the matrix rather than being inserted between digits of a single element.

n may be any scalar-valued expressions in the range of 2-256. Nonintegers will be truncated to an integer. If 256 is used, no additional lines will be inserted.

The default is 80 columns.

Example

```
    outwidth 132;
```

This statement will change the auxiliary output width to 132 columns.

See Also

output, print

p

pacf

Purpose

Computes sample partial autocorrelations.

Format

rkk = **pacf**(*y*, *k*, *d*);

Input

y	Nx1 vector, data.
k	scalar, maximum number of partial autocorrelations to compute.
d	scalar, order of differencing.

Output

rkk	Kx1 vector, sample partial autocorrelations.

Example

```
proc pacf(y,k,d);
```

```
        local a,l,j,r,t;
        r = acf(y,k,d);
        a = zeros(k,k);
        a[1,1] = r[1];
        t = 1;
        l = 2;

        do while l le k;
          a[l,l] =(r[l]-a[l-1,1:t]*rev(r[1:l-1]))/
          (1-a[l-1,1:t]*r[1:t]);
          j = 1;

          do while j <= t;
            a[l,j] = a[l-1,j] - a[l,l]*a[l-1,l-j];
            j = j+1;
          endo;

          t = t+1;
          l = l+1;
        endo;

     retp(diag(a));
     endp;
```

Source

tsutil.src

packedToSp

Purpose

Creates a sparse matrix from a packed matrix of non-zero values and row and column indices.

Format

```
y = packedToSp(r, c, p);
```

Input

r	scalar, rows of output matrix.
c	scalar, columns of output matrix.
p	Nx3 or Nx4 matrix, containing non-zero values and row and column indices.

Output

y	*r* x *c* sparse matrix.

Remarks

If *p* is Nx3, *y* will be a real sparse matrix. Otherwise, if *p* is Nx4, *y* will be complex.

The format for *p* is as follows:

If *p* is Nx3:

Column 1	Column 2	Column 3
non-zero values	row indices	column indices

If *p* is Nx4:

Column 1	Column 2	Column 3	Column 4

real non-zero values	imaginary non-zero values	row indices	column indices

Note that **spCreate** may be faster.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **packedToSp**.

Example

```
//Declare 'y' to be a sparse matrx
sparse matrix y;

//Create a 15x10 matrix 'y' in which:
//y[2,4] = 1.1; y[5,1] = 2.3; y[8,9] = 3.4;
//y[13,5] = 4.2
//all other values in 'y' will be zeros
p = { 1.1 2 4, 2.3 5 1, 3.4 8 9, 4.2 13 5 };
y = packedToSp(15,10,p);
```

After the code above, *y* is a sparse matrix, containing the following non-zero values:

```
Non-zero value    Index

    1.1           (2,4)
    2.3           (5,1)
    3.4           (8,9)
    4.2          (13,5)
```

See Also

spCreate, denseToSp

packr

Purpose

Deletes the rows of a matrix that contain any missing values.

Format

y = **`packr`**(*x*);

Input

x	NxK matrix.

Output

y	LxK submatrix of *x* containing only those rows that do not have missing values in any of their elements.

Remarks

This function is useful for handling missing values by "listwise deletion," particularly prior to using the **/** operator to compute least squares coefficients.

If all rows of a matrix contain missing values, **`packr`** returns a scalar missing value. This can be tested for quickly with the **`scalmiss`** function.

Example

Example 1

```
//Set the rng seed for repeatable random numbers
rndseed 7342692;

//Create a 3x3 matrix of random integers between 1 and 10
x = ceil(rndu(3, 3) * 10);

//Turn all elements with a value of 8 into missing values
x2 = miss(ceil(rndu(3,3)*10),8);

//Remove all rows that contain missing values
y = packr(x2);
```

After the code above:

```
    6 10  3          6 10  3
x = 8  7  8     x2 = .  7  .     y = 6 10 3
    8  6  7          .  6  7
```

Example 2

```
//Open a GAUSS data file for reading
open fp = mydata;
obs = 0;
sum = 0;

//Continue looping until the end of the file has been
//reached
do until eof(fp);
   //Read in 100 lines of the data file and remove any rows
   //with missing values
   x = packr(readr(fp,100));
   //Check to see if 'packr' returned a missing value; if
   //not, update 'obs' and 'sum'
   if not scalmiss(x);
     obs = obs + rows(x);
```

```
            sum = sum + sumc(x);
          endif;
       endo;
       mean = sum/obs;
```

In this example the sums of each column in a data file are computed as well as a count of the rows that do not contain any missing values. **packr** is used to delete rows that contain missings and **scalmiss** is used to skip the two sum steps if all the rows are deleted for a particular iteration of the read loop. Then the sums are divided by the number of observations to obtain the means.

See Also

scalmiss, miss, missrv

parse

Purpose

Parses a string, returning a character vector of tokens.

Format

tok = **parse**(*str*, *delim*);

Input

str	string consisting of a series of tokens and/or delimiters.
delim	NxK character matrix of delimiters that might be found in *str*.

Output

tok	Mx1 character vector consisting of the tokens contained in *str*. All tokens are returned; any delimiters found in *str* are ignored.

Remarks

The tokens in *str* must be 8 characters or less in size. This is because they are returned in a character vector in which each element is represented as a double precision value. If they are longer, the contents of *tok* is unpredictable. Use string arrays to create arrays of text with elements longer than 8 characters.

Example

Example 1

```
names = "GDP;GNP;M1;M2";
namesVec = parse(names, ";");

//The '$' is used when printing character vectors
print $namesVec;
```

The code above will return:

```
GDP
GNP
 M1
 M2
```

Example 2

```
obs = 1000;
names = "Age,Weight,Height";
```

```
//Create uniform random integers between 1 and 77
data1 = ceil(77 * rndu(obs,1));

//Create normal random integers centered at 100 with a
//standard deviation of 9
data2 = ceil(100 + 9*rndn(obs,1));

//Create uniform random numbers between 0 and 60
data3 = ceil(60 * rndu(obs,1));

//Horizontally concatenate data into 'obs'x3 matrix
data = data1~data2~data3;

//Print the data using the procedure below
printStats(names, data);

//Create procedure to take our data, calculate some basic
//stats and print them
proc (0) = printStats( names, data);
   local title, vars, sepVars;

   //Set to print with 6 spaces between numbers and 0
   //digits after the decimal
   format /rd 6,0;

   //Create the titles to print for each column
   title = parse("var,mean,max,min", ",");

   //Extract the substrings from 'names' into a character
   //array using the comma as a separator between tokens
   sepVars = parse(names, ",");
   print "----------------------------------------";

   //The '$' tells GAUSS to print as character data
```

```
        print $title';
        print "---------------------------------------"
        //Loop through as many times as there are rows in
        //'sepVars'
        for i( 1, rows(sepVars), 1);
           //Two semi-colons at the end of a print statement
           //prevents a new-line after the print
           print $sepVars[i];;
           print meanc(data[.,i]);;
           print maxc(data[.,i]);;
           print minc(data[.,i]);
        endfor;
        print "---------------------------------------";
     endp;
```

The code above will produce output like this:

```
     -----------------------------------
        var   mean     max     min
     -----------------------------------
        Age     38      77       1
     Weight    101     135      75
     Height     31      60       1
     -----------------------------------
```

See Also

token

pause

Purpose

Pauses for a specified number of seconds.

Format

```
pause(sec);
```

Input

sec	scalar, seconds to pause.

Remarks

This function can be used to delay a program, allowing users time to view graphics and/or data printed to the program output window.

Source

`pause.src`

See Also

wait

pdfCauchy

Purpose

Computes the probability density function for the Cauchy distribution.

Format

```
y = pdfCauchy(x,a,b);
```

Input

x	NxK matrix, an Nx1 vector or scalar.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The probability density function for the Cauchy distribution is defined as

$$f(x) = \left(\pi\sigma\left(1 + \left(\frac{x-\mu}{\sigma}\right)^2\right)\right)^{-1}$$

See Also

cdfCauchy

pdfexp

Purpose

Computes the probability density function for the exponential distribution.

Format

```
y = pdfexp(x,a,m);
```

Input

x	NxK matrix, Nx1 vector or scalar. *x* must be greater than *a*.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.
m	Scalar, mean parameter. *m* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The probability density function for the exponential distribution is defined as

$$f(x) = \lambda exp(-\lambda(x-\gamma))$$

See Also

cdfexp

pdfGenPareto

Purpose

Computes the probability density function for the Generalized Pareto distribution.

Format

```
y = pdfGenPareto(x,a,o,k);
```

Input

x	NxK matrix, an Nx1 vector or scalar.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.
o	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *o* must be greater than 0.
k	Shape parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The probability density function for the Generalized Pareto distribution is defined as

$$f(x) = \begin{cases} \frac{1}{\sigma}\left(1 + k\frac{(x-\mu)}{\sigma}\right)^{-1-1/k} & k \neq 0 \\ \frac{1}{\sigma}exp\left(-\frac{(x-\mu)}{\sigma}\right) & k = 0 \end{cases}$$

See Also

cdfGenPareto

pdfLaplace

Purpose

Computes the probability density function for the Laplace distribution.

Format

```
y = pdfLaplace(x,a,b);
```

Input

x	NxK matrix, Nx1 vector or scalar.
a	Scalar, location parameter.
b	Scalar, scale parameter. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The probability density function for the Laplace distribution is defined as

$$f(x) = \frac{\lambda}{2}exp(-\lambda\,|\,x-\mu|)$$

See Also

cdfCauchy, pdfCauchy

pdflogistic

Purpose

Computes the probability density function for the logistic distribution.

Format

y = **pdflogistic**(*x*,*a*,*b*);

Input

x	NxK matrix, an Nx1 vector or scalar.
a	Location parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The probability density function for the logistic distribution is defined as

$$f(x) = \frac{exp(-z)}{\sigma(1 + exp(-z))^{-2}}$$

See Also

cdflogistic

pdfn

Purpose

Computes the standard Normal (scalar) probability density function.

Format

y = **pdfn**(*x*);

Input

x	NxK matrix.

Output

y	NxK matrix containing the standard Normal probability density function of *x*.

Remarks

This does not compute the joint Normal density function. Instead, the scalar Normal density function is computed element-by-element. *y* could be computed by the following **GAUSS** code:

```
    y =(1/sqrt(2*pi))*exp(-(x.*x)/2);
```

Example

```
    x = { -3, -2, 0, 2, 3 };
    y = pdfn(x);
```

After the code above:

```
        0.0044318484
         0.053990967
    y  =  0.39894228
         0.053990967
        0.0044318484
```

pdfRayleigh

Purpose

Computes the probability density function of the Rayleigh distribution.

Format

y = **`pdfRayleigh`**(*x*, *b*);

Input

x	NxK matrix, an Nx1 vector or scalar. *x* must be greater than 0.
b	Scale parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *b* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The probability density function of the Rayleigh distribution is defined as

$$\frac{x exp\left(\frac{-x^2}{2\sigma^2}\right)}{\sigma^2}$$

See Also

cdfRayleighinv

p

pdfWeibull

Purpose

Computes the probability density function of a Weibull random variable.

Format

```
y = pdfWeibull(x,k,lambda);
```

Input

x	NxK matrix, Nx1 vector or scalar. *x* must be greater than 0.
k	Shape parameter; NxK matrix, Nx1 vector or scalar, ExE conformable with *x*. *k* must be greater than 0.
lambda	Scale parameter; may be matrix, Nx1 vector or scalar, ExE conformable with *x*. *lambda* must be greater than 0.

Output

y	NxK matrix, Nx1 vector or scalar.

Remarks

The probability density function of a Weibull random variable is defined as

$$f(x, \lambda, k) = \begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)k} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

See Also

cdfWeibull, cdfWeibullInv

pi

Purpose

Returns the mathematical constant π.

Format

y = **pi**;

Output

y	scalar, the value of π.

Example

```
//Print 14 digits and allow 16 digits worth of space for
//each printed number
format /rdn 16,14;
print pi;
```

will return:

```
     3.14159265358979
```

pinv

Purpose

Computes the Moore-Penrose pseudo-inverse of a matrix, using the singular value decomposition. This pseudo-inverse is one particular type of generalized inverse.

Format

y = **pinv**(*x*);

Input

x	NxM matrix.

Global Input

_svdtol	scalar, any singular values less than *_svdtol* are treated as zero in determining the rank of the input matrix. The default value for *_svdtol* is 1.0e-13.

Output

y	MxN matrix that satisfies the 4 Moore-Penrose conditions: $xyx = x$ $yxy = y$ xy is symmetric yx is symmetric

Global Output

`_svderr`	scalar, if not all of the singular values can be computed _ `svderr` will be nonzero.

Example

pinv can be used to solve an undertermined least squares problem.

```
//Create an underdetermined system of equations 'A'
A = rndn(4, 5);

//Create a right hand side
b = rndn(4,1);

if rank(A) < cols(A);
   print "A does not have full rank, using pinv to sol
ve";
   Api = pinv(A);
   x = Api*b;
else;
```

```
        print "A has full rank, solve with '/' operator";
        x = b/A;
    endif;
```

Least squares problems with full rank can also be solved with the **GAUSS** functions: **`ols`**, **`olsqr`** and **`olsqr2`**.

Source

`svd.src`

pinvmt

Purpose

Computes the Moore-Penrose pseudo-inverse of a matrix, using the singular value decomposition. This pseudo-inverse is one particular type of generalized inverse.

Format

{ *y*, *err* } = **`pinvmt`**(*x*, *tol*);

Input

x	NxM matrix.
tol	scalar, any singular values less than *tol* are treated as zero in determining the rank of the input matrix.

Output

y	MxN matrix that satisfies the 4 Moore-Penrose conditions: $xyx = x$ $yxy = y$ xy is symmetric yx is symmetric
err	scalar, if not all of the singular values can be computed *err* will be nonzero.

pinvmt can be used to solve an undertermined least squares problem.

```
tol = 1e-13;

//Create an underdetermined system of equations 'A'
A = rndn(4, 5);

//Create a right hand side
b = rndn(4,1);

if rank(A) < cols(A);
   print "A does not have full rank, using pinvmt to s
olve";
   Api = pinvmt(A, tol);
   x = Api*b;
else;
   print "A has full rank, solve with '/' operator";
   x = b/A;
endif;
```

Least squares problems with full rank can also be solved with the **GAUSS** functions: **`ols`**, **`olsqr`** and **`olsqr2`**.

Source

```
svdmt.src
```

plotAddBar

Purpose

Adds a bar or a set of bars to an existing graph.

Format

```
plotAddBar(myPlot, val, ht);
plotAddBar(val, ht);
```

Input

myPlot	A **plotControl** structure.
val	Nx1 numeric vector, bar labels. If scalar 0, a sequence from 1 to **`rows`**(*ht*) will be created.
ht	NxK numeric vector, bar heights. K overlapping or side-by-side sets of N bars will be graphed.

Remarks

`plotAddBar` may only add bars to 2-D graphs.

This function will not change any of the current graph's settings other than to resize the view as necessary to display the new curve.

See Also

plotAddHist, plotAddHistF, plotAddHistP, plotAddPolar, plotAddXY

plotAddBox

Purpose

Adds a box graph to an existing graph.

Format

```
plotAddBox(myPlot,  grp, y);
plotAddBox(grp, y);
```

Input

myPlot	A **plotControl** structure.
grp	1xM vector. This contains the group numbers corresponding to each column of *y* data. If scalar 0, a sequence from 1 to **cols**(*y*) will be generated automatically for the X axis.
y	NxM matrix. Each column represents the set of *y* values for an individual percentiles box symbol.

Remarks

plotAddBox may only add a box graph to 2-D graphs.

This function will not change any of the current graph's settings other than to resize the view as necessary to display the new curve.

See Also

plotAddHist, plotAddHistF, plotAddHistP, plotAddPolar, plotAddXY

plotAddHist

Purpose

Adds a histogram to an existing graph.

Format

`plotAddHist`(*myPlot*, *x*, *v*);
`plotAddHist`(*x*, *v*);

Input

myPlot	A **plotControl** structure.
x	Mx1 vector of data.
v	Nx1 vector, the breakpoints to be used to compute the frequencies - or - scalar, the number of categories.

Remarks

`plotAddHist` may only add a histogram to 2-D graphs.

This function will not change any of the current graph's settings other than to resize the view as necessary to display the new curve.

See Also

plotAddBar, plotAddHistF, plotAddHistP, plotAddPolar, plotAddXY

plotAddHistF

Purpose

Adds a frequency histogram to an existing graph.

Format

`plotAddHistF`(*`myPlot`*, *`f`*, *`c`*);
`plotAddHistF`(*`f`*, *`c`*);

Input

myPlot	A **plotControl** structure.
f	Nx1 vector, frequencies to be graphed.
c	Nx1 vector, numeric labels for categories. If this is a scalar 0, a sequence from 1 to **`rows`**(*f*) will be created.

Remarks

`plotAddHistF` may only add a histogram to 2-D graphs.

This function will not change any of the current graph's settings other than to resize the view as necessary to display the new curve.

See Also

plotAddBar, plotAddHist, plotAddHistP, plotAddPolar, plotAddXY

plotAddHistP

Purpose

Adds a percent histogram to an existing graph.

Format

```
plotAddHistP(myPlot, x, v);
plotAddHistP(x, v);
```

Input

myPlot	A **plotControl** structure.
x	Mx1 vector of data.
v	Nx1 vector, the breakpoints to be used to compute the frequencies - or - scalar, the number of categories.

Remarks

plotAddHistP may only add a histogram to 2-D graphs.

This function will not change any of the current graph's settings other than to resize the view as necessary to display the new curve.

See Also

plotAddBar, plotAddHist, plotAddHistF, plotAddPolar, plotAddXY

plotAddPolar

Purpose

Adds a graph using polar coordinates to an existing polar graph.

Format

```
plotAddPolar(myPlot, radius, theta);
plotAddPolar(radius, theta);
```

Input

myPlot	A **plotControl** structure.
radius	Nx1 or NxM matrix. Each column contains the magnitude for a particular line.
theta	Nx1 or NxM matrix. Each column represents the angle values for a particular line.

Remarks

plotAddPolar may only add curves to 2-D graphs.

This function will not change any of the current graph's settings other than to resize the view as necessary to display the new curve.

See Also

plotAddBar, plotAddHist, plotAddHistF, plotAddHistP, plotAddXY

plotAddScatter

Purpose

Adds a 2-dimensional scatter plot to an existing graph.

Format

```
plotAddScatter(myPlot, x, y);
plotAddScatter(x, y);
```

Input

myPlot	A **plotControl** structure.
x	Nx1 or NxM matrix. Each column contains the X values for a particular data point.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular data point.

Remarks

plotAddScatter may only add a scatter plot to 2-D graphs.

This function will not change any of the current graph's settings other than to resize the view as necessary to display the new curve.

See Also

plotAddBar, plotAddHist, plotAddHistF, plotAddHistP, plotAddScatter, plotAddXY

plotAddTS

Purpose

Adds a curve of time series data to an existing time series plot.

Format

```
plotAddTS(myPlot, dtstart, frequency, y);
plotAddTS(dtstart, frequency, y);
```

Input

myPlot	A **plotControl** structure.	
dtstart	Scalar, starting date in DT scalar format.	
frequency	Scalar, frequency of the data per year. Valid options include:	
	1	Yearly
	4	Quarterly
	12	Monthly
y	Nx1 or NxM matrix. Each column contains the Y values for	

P

a particular line.

Examples

Example 1

```
//Create some data to plot
y = rndn(100, 1);

//The first input starts the series in January of 1982
//The second input specifies the data to be monthly
plotTS(1982, 12, y);

y2 = rndu(28, 1);

//Add the data from 'y2' as quarterly data
//starting in Q2 of 1980
plotAddTS(198004, 4, y2);
```

Remarks

You may only add time series graphs to other time series graphs. For more information on time series graphs, see **Time Series Plots in GAUSS**, Section 1.1.

By default missing values in the *y* variable will be represented as gaps in the line.

See Also

plotSetXTicLabel, plotSetXTicInterval, plotTS

plotAddXY

Purpose

Adds an XY graph to an existing graph.

Format

```
plotAddXY(myPlot, x, y);
plotAddXY(x, y);
```

Input

myPlot	A **plotControl** structure.
x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

Remarks

plotAddXY may only add curves to 2-D graphs.

This function will not change any of the current graph's settings other than to resize the view as necessary to display the new curve.

See Also

plotAddBar, plotAddHist, plotAddHistF, plotAddHistP, plotAddPolar

plotBar

Purpose

Generates a bar graph.

Format

```
plotBar(myPlot, val, ht);
plotBar(val, ht);
```

Input

myPlot	A **plotControl** structure.
val	Nx1 numeric vector, bar labels. If scalar 0, a sequence from 1 to **rows**(*ht*) will be created.
ht	NxK numeric vector, bar heights. K overlapping or side-by-side sets of N bars will be graphed.

Example

In this example, five bars will be created.

```
// Create data
x = seqa(1, 1, 5);
y = { 1.5, 2, 3, 0.5, 1 };

// Draw bar graph
plotBar(x, y);
```

Remarks

To control the color and texture of the bars as well as whether they are stacked or side by side:

If you are passing a **plotControl** structure to your graph, you may use the function **`plotSetBar`**.

If you are not passing a **plotControl** structure, these properties are set in the Preferences. To access the, select **Tools->Preferences** from the **GAUSS** main menu. Select Graphics on the left side of the preferences and then select the radio button next to "Bar." A dropdown menu will be available under Group 1 for both of these options.

See Also

plotXY, plotLogX, plotHist

plotBox

Purpose

Graphs data using the box graph percentile method.

Format

`plotBox`(*`myPlot`*, *`grp`*, *y*);
`plotBox`(*`grp`*, *y*);

Input

`myPlot`	A **plotControl** structure.
`grp`	1xM vector. This contains the group numbers corresponding

	to each column of *y* data. If scalar 0, a sequence from 1 to **cols**(*y*) will be generated automatically for the X axis.
y	NxM matrix. Each column represents the set of *y* values for an individual percentiles box symbol.

Remarks

If missing values are encountered in the *y* data, they will be ignored during calculations and will not be plotted.

See Also

plotHistP, plotScatter

plotClearLayout

Purpose

Clears any previously set plot layouts.

Format

```
plotClearLayout();
```

Example

```
//Create a 1x2 Plot Layout and insert a percentage
//histogram of some random normal numbers in the first
//cell.
plotLayout(1, 2, 1);
```

```
    plotHistP(rndn(1000, 1), 30);

    //Insert gamma distributed random numbers into the second
    //cell.
    plotLayout(1, 2, 2);
    plotHistP(rndGamma(1000, 1, 3, 2), 30);

    //Display the image for 2 seconds
    pause(2);

    //Clear the 1x2 layout
    plotClearLayout();

    //Plot percentage histogram of beta distributed random
    //numbers. This graph will take up the entire plot window
    //since the 1x2 plot layout has been cleared.
    plotHistP(rndBeta(1000, 1, 2, 1), 30);
```

Remarks

After calling this function all subsequent graphs will be drawn to fill the entire graph window.

See Also

plotSetBar, plotBar, plotLayout, plotCustomLayout

plotContour

Purpose

Graphs a matrix of contour data.

Format

```
    plotContour(myPlot, x, y,  z);
    plotContour(x, y,  z);
```

Input

myPlot	Optional input: **plotControl** structure.
x	1xK vector, the X axis data.
y	Nx1 vector, the Y axis data.
z	NxK matrix, the matrix of height data to be plotted.

Remarks

A vector of evenly spaced contour levels will be generated automatically from the *z* matrix data. Each contour level will be labeled. For unlabeled contours, use **ztics**.

To specify a vector of your own unequal contour levels, set the vector *_plev* before calling **contour**.

To specify your own evenly spaced contour levels, see **ztics**.

See Also

plotSurface

plotCustomLayout

Purpose

Plots a graph of user-specified size at a user-specified location.

Format

`plotCustomLayout`(*`xStart`*, *`yStart`*, *`width`*, *`height`*);

Input

`xStart`	scalar, the distance from the left edge of the canvas to the left edge of the custom plot expressed as a number between 0 and 1.
`yStart`	scalar, the distance from the bottom edge of the canvas to the bottom edge of the custom plot expressed as a number between 0 and 1.
`width`	scalar, the width of the custom plot expressed as a number between 0 and 1.
`height`	scalar, the height of the custom plot expressed as a number between 0 and 1.

Example

```
//Create an additive sequence starting from -pi and moving
//forward in 0.1 increments
x = seqa(-pi, 0.1, 63);
```

```
//Plot the cosine of x
plotXY(x, cos(x));

//Create a custom section for the next graph starting 10%
//from the main graph's left edge, 10% from the bottom of
//the main graph, with a width and height both equalling
//30% of the width of the main graph.
plotCustomLayout(0.1, 0.1, 0.3, 0.3);

//Plot the next graph in the custom layout
plotXY(x[1:20], cos(x[1:20] ) ));

//Prevent the next graph from being drawn in this custom
//region
plotClearLayout();
```

Remarks

After calling this function all subsequent graphs will be plotted inside of the specified custom layout until the layout is reset with **`plotLayout`**, or the layout is cleared with **`plotClearLayout`**.

See Also

plotSetBar, plotBar, plotHistP, plotGetDefaults

plotGetDefaults

Purpose

Gets default settings for plotting graphs.

Format

```
myPlot = plotGetDefaults(graph);
```

Input

graph	String, name of graph type: bar, box, hist, polar, scatter, surface or xy.

Output

myPlot	A **plotControl** structure.

Example

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure with defaults for an
//'xy' graph
myPlot = plotGetDefaults("xy");

//Create some data to plot
x = seqa(-5, 0.1, 50);
y = pdfn(x);

//Make a desired change to the plotControl structure
plotSetTitle(&myPlot, "Default XY Settings");

//Plot the data using the plotControl structure
plotXY(myPlot, x, y);
```

Remarks

The **`plotGetDefaults`** function will use the default settings for the specified graph type. These may be accessed from the main menu bar: **Tools->Preferences->Graphics**.

See Also

plotSetBkdColor, plotSetLineColor, plotSetLineSymbol

plotHist

Purpose

Computes and graphs a frequency histogram for a vector. The actual frequencies are plotted for each category.

Format

`plotHist`(*`myPlot`*, *x*, *v*);
`plotHist`(*x*, *v*);

Input

`myPlot`	A **plotControl** structure.
x	Mx1 vector of data.
v	Nx1 vector, the breakpoints to be used to compute the frequencies - or - scalar, the number of categories

Example

```
    //Create some data to plot
    x = rndn(5000, 1);

    //Plot the data
    plotHist(x, 20);
```

See Also

plotHistP, plotHistF, plotBar

plotHistF

Purpose

Graphs a histogram given a vector of frequency counts.

Format

```
plotHistF(myPlot, f, c);
plotHistF(f, c);
```

Input

myPlot	A **plotControl** structure.
f	Nx1 vector, frequencies to be graphed.
c	Nx1 vector, numeric labels for categories. If this is a scalar 0, a sequence from 1 to **rows**(*f*) will be created.

Remarks

The axes are not automatically labeled. Use the functions **`plotSetXLabel`** and and **`plotSetYLabel`**.

See Also

plotHist, plotBar, plotSetXLabel

plotHistP

Purpose

Computes and graphs a percent frequency histogram of a vector. The percentages in each category are plotted.

Format

`plotHistP`(*`myPlot`*, *x*, *v*);
`plotHistP`(*x*, *v*);

Input

myPlot	A **plotControl** structure.
x	Mx1 vector of data.
v	Nx1 vector, the breakpoints to be used to compute the frequencies - or - scalar, the number of categories.

See Also

plotHist, plotHistF, plotBar, plotBox, plotScatter

plotLayout

Purpose

Divides a plot into a grid of subplots and assigns the cell location in which to draw the next created graph.

Format

plotLayout(*gRows*, *gCols*, *ind*);

Input

gRows	scalar, number of rows of the graph layout.
gCols	scalar, number of columns of the graph layout.
ind	scalar, cell location in which to place the next created graph.

Example

```
//Create 10x4 matrix where each column is an additive
//sequence from 0.1 to 1.0
x = seqa(0.1, 0.1, 10);
y = ones(10, 4).*x;

//Apply a function to each column of 'y'
```

```
    y[.,1] = cos(x);
    y[.,2] = sin(x);
    y[.,3] = cdfn(x);
    y[.,4] = exp(x);

    for i(1, 4, 1);
       //Divide plot canvas into a 2x2 grid of subplot
       //locations and place each newly created graph in the
       //next available cell location.
       plotLayout(2, 2, i);

       //Plot each column of y in a separate subplot window.
       plotXY(x, y[.,i]);
    endfor;

    //Clear the layout so the next plot will not be inside this
    //layout
    plotClearLayout();
```

Remarks

After calling this function all subsequent graphs will be plotted inside of the specified layout until the layout is reset with **plotLayout**, or the layout is cleared with **plotClearLayout**.

See Also

plotBar, plotClearLayout, plotCustomLayout, plotHist

plotLogLog

Purpose

Graphs X vs. Y using log coordinates.

Format

plotLogLog(*myPlot*, *x*, *y*);
plotLogLog(*x*, *y*);

Input

myPlot	A **plotControl** structure.
x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

See Also

plotXY, plotLogX, plotLogY

plotLogX

Purpose

Graphs X vs. Y using log coordinates for the X axis.

Format

```
plotLogX(myPlot, x, y);
plotLogX(x, y);
```

Input

myPlot	A **plotControl** structure.
x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

See Also

plotXY, plotLogY, plotLogLog

plotLogY

Purpose

Graphs X vs. Y using log coordinates for the Y axis.

Format

```
plotLogY(myPlot, x, y);
plotLogY(x, y);
```

Input

myPlot	A **plotControl** structure.
x	Nx1 or NxM matrix. Each column represents the X values for a particular line.
y	Nx1 or NxM matrix. Each column represents the Y values for a particular line.

See Also

plotXY, plotLogX, plotLogLog

plotOpenWindow

Purpose

Opens a new, empty graphic window to be used by the next drawn graph.

Format

```
plotOpenWindow();
```

Example

```
//Create data
x = rndn(10000, 1);
x2 = rndn(10000, 1);
x3 = rndn(10000, 1);

//Plot first vector as a percentage histogram with 30 bins
```

```
    plotHistP(x, 30);

    //Plot second vector, drawing over the previously created
    //graph.
    plotHistP(x2, 30);

    //Create a new graphic window and plot the second vector as
    //a percentage histogram with 30 bins inside this new
    //window.
    plotOpenWindow();

    //Draw the graph
    plotHistP(x3, 30);
```

Remarks

To automatically open each new graph in a new graph window, use **`plotSetNewWindow`** or set the preference in the main applicaton menu. This may be found by selecting **Tools->Preferences** and then clicking on **Graphics** on the left side of the preferences window.

If you select the radio button next to "New Window" at the top of the graphics preferences window, each new graph will be automatically drawn in a new graphics window.

See Also

plotSave, plotCustomLayout, plotSetLegend, plotSetNewWindow

plotPolar

Purpose

Graph data using polar coordinates.

Format

```
    plotPolar(myPlot, radius, theta);
    plotPolar(radius, theta);
```

Input

myPlot	A **plotControl** structure.
radius	Nx1 or NxM matrix. Each column contains the magnitude for a particular line.
theta	Nx1 or NxM matrix. Each column represents the angle values for a particular line.

See Also

plotXY, plotLogX, plotLayout, plotSetXLabel

plotSave

Purpose

Saves the last created graph to a user specified file type.

Format

```
    plotSave(filename, size);
```

Input

filename	String, name of the file to create with a file type extension. Available file extensions include: `.jpg`, `.plot`, `.png`, `.pdf`, `.svg`, `.tiff`.
size	2x1 vector, dimensions of the saved graph in centimeters.

Example

```
//Create data
x = seqa(1, 1, 10);
y = cos(x);

//Plot the data
plotXY(x, y);

//Save the graph as a pdf with a width of 30 cm and a
//height of 18 cm
dim = { 30, 18 };
plotSave("mygraph.pdf", dim);
```

Technical Notes

The `.plot` file extension is an xml file that is the native format used by **GAUSS** to save graphs.

See Also

plotCustomLayout, plotSetLegend

plotScatter

Purpose

Creates a 2-dimensional scatter plot.

Format

```
plotScatter(myPlot, x, y);
plotScatter(x, y);
```

Input

myPlot	A **plotControl** structure.
x	Nx1 or NxM matrix. Each column contains the X values for a particular data point.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular data point.

Example

```
//Create random normal data
x = rndn(50, 1);

//Reverse the order of 'x' and set it to be the 'y' value
y = rev(x);

//Plot the data
plotScatter(x, y);
```

See Also

plotXY, plotLogLog, plotBox, plotHistP

plotSetBar

Purpose

Sets the fill style and format of bars in a histogram or bar graph.

Format

```
plotSetBar(&myPlot, fillType, barStacked);
```

Input

&myPlot	A **plotControl** structure pointer.	
fillType	Nx1 vector, where N is the number of bar styles to set.	
	0	Solid, beveled edge
	1	Solid
	2	Dense 1
	3	Dense 2
	4	Dense 3
	5	Dense 4
	6	Dense 5

	7	Dense 6
	8	Horizontal lines
	9	Vertical lines
	10	Cross pattern
	11	B diagonal pattern
	12	F diagonal pattern
	13	Diagonal Cross
barStacked	Scalar, 1 for stacked or 0 for side-by-side bars.	

Example

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
myPlot = plotGetDefaults("bar");

//Set the first set of bars to have a solid-fill, the
//second set to have a fill of horizontal lines, the third
//to have a diagonal cross fill and set the bars to be
//side-by-side.
textures = { 0, 8, 13 };
plotSetBar(&myPlot, textures, 0);

//Create data
x = seqa(1, 1, 5);
y = { 1.5, 2, 3, 0.5, 1 };

//Draw bar graph
```

```
    plotBar(myPlot, x, y);
```

Remarks

When graphing without the use of a **plotControl** structure, these settings may be chosen through the **Tools->Preferences->Graphics** menu, after selecting the Bar radio button. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotBar, plotGetDefaults, plotHist

plotSetBkdColor

Purpose

Sets the background color of a graph.

Format

```
    plotSetBkdColor(&myPlot, color);
```

Input

&myPlot	A **plotControl** structure pointer.
color	String, name or rgb value of the new color.

Example

```
    //Declare plotControl structure
    struct plotControl myPlot;

    //Initialize plotControl structure
    myPlot = plotGetDefaults("polar");

    //Set new background color to light grey
    plotSetBkdColor(&myPlot, "light grey");

    //Create data
    x = seqa(0.1, 0.1, 200);
    y = x;

    //Create a polar plot of the data with the new background
    //color
    plotPolar(myPlot, x, y);
```

Remarks

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotSetLineColor, plotSetLineSymbol

plotSetGrid

Purpose

Controls the settings for the background grid of a plot.

Format

plotSetGrid(*&myPlot*, *ticStyle*, *color*);
plotSetGrid(*&myPlot*, *ticStyle*);
plotSetGrid(*&myPlot*, *onOff*);

Input

ticStyle	String, specifies whether grid marks should be drawn on minor tic marks or only on major tic marks. Options: "major" or "minor."
color	String, name or rgb value of the new color.
onOff	String, turns the grid on or off. Options: "on" or "off." If used, this must be the only argument passed to the function besides the **plotControl** structure pointer.

Example

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
myPlot = plotGetDefaults("scatter");
```

```
//Set grid to be black and on the major tics only
plotSetGrid(&myPlot, "black", "major");

//Create a scatter plot of random data
plotScatter(myPlot, seqa(1, 1, 10 ), rndn(10, 1));

//Turn off the grid
plotSetGrid(&myPlot, "off");
```

See Also

plotCustomLayout, plotSetTitle

plotSetLegend

Purpose

Adds a legend to a graph.

Format

```
plotSetLegend(&myPlot, label, location, orientation);
plotSetLegend(&myPlot, label, location);
plotSetLegend(&myPlot, label);
plotSetLegend(&myPlot, onOff);
```

Input

&myPlot	A **plotControl** structure pointer.

label	String array, names of the line labels.
location	String or 2x1 vector, the location to place the legend. **String case:** The location string may contain up to three tokens, or words. 1. Vertical location: top (default), middle or bottom. 2. Horizontal location: left, center or right (default). 3. Inside/Outside location: inside (default), below or outside. **2x1 vector case:** The first element sets the horizontal location and the second sets the vertical location of the bottom left corner of the graph; expressed as a percentage of the total height and width of the graph.
orientation	scalar, 0 for a horizontal legend or 1 for a vertical legend.
onOff	string, "on" or "off". "on" will add the default legend to each graph. "off" will stop **GAUSS** from adding the default legend to subsequent graphs.

Technical Notes

The location parameter (in the string case) is a string with up to three tokens or words that are separated by a space. For example,

```
location = "top right";
location = "right top";
location = "inside top right";
```

will all set the legend to the top right position, inside the graph. To locate the bottom left corner of the legend at the origin:

```
location = { 0, 0 };
```

To place the bottom left corner of the legend in the center of the graph:

```
location = { 0.5, 0.5 };
```

Example

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
myPlot = plotGetDefaults("xy");

//Set labels, location, and orientation of legend
label = "sample A"$|"sample B";
location = "top right";
orientation = 0;
plotSetLegend(&myPlot, label, location, orientation);

//Create data
x = seqa(1, 1, 10);
```

```
    y = cos(x);

    //Plot the data with the legend settings
    plotXY(myplot, x, y);
```

See Also

plotLayout, plotCustomLayout, plotOpenWindow

plotSetLineColor

Purpose

Sets the line colors for a graph.

Format

```
plotSetLineColor(&myPlot, colors);
```

Input

&myPlot	A **plotControl** structure pointer.
colors	String array, name or rgb value of the new colors.

Example

```
    //Declare plotControl structure
    struct plotControl myPlot;

    //Initialize plotControl structure
    myPlot = plotGetDefaults("xy");

    //Set new line colors to aqua and midnight blue
    clrs = "aqua"$|"midnight blue";
    plotSetLineColor(&myPlot, clrs);

    //Create data
    x = seqa(0.1, 1, 50);
    y = sin(x)~cos(x);

    //Plot the data with the new line colors
    plotXY(myPlot, x, y);
```

Remarks

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotSetLineSymbol

plotSetLineStyle

Purpose

Sets the line styles for a graph.

Format

plotSetLineStyle(*&myPlot*, *newStyle*);

Input

&myPlot	A **plotControl** structure pointer.
newStyle	Matrix, new line styles. Options include:
	1 Solid line.
	2 Dot line.
	3 Dash line.
	4 Dash-Dot line.
	5 Dash-Dot-Dot line.

Example

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
myPlot = plotGetDefaults("xy");

//Set line 1 as a solid line, set line 2 as a dot line,
//etc.
newStyle = { 1, 2, 3, 4, 5 };
plotSetLineStyle(&myPlot, newStyle);

//Create data
x = seqa(0.1, 1, 50);
y = sin(x)~cos(x);

//Plot the data with the new line styles
plotXY(myPlot, x, y);
```

Remarks

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotSetTitle, plotSetLineSymbol

plotSetLineSymbol

Purpose

Sets the symbols displayed on the plotted points of a graph.

Format

```
plotSetLineSymbol(&myPlot, newSymbol, symbolWidth);
plotSetLineSymbol(&myPlot, newSymbol);
```

Input

&myPlot	A **plotControl** structure pointer.
newSymbol	Matrix, new line symbol settings. Options include:
	-1 None.

	0	Ellipse.
	1	Rectangle.
	2	Diamond.
	3	Upward pointing triangle.
	4	Downward pointing triangle.
	5	Triangle.
	6	Leftward pointing triangle.
	7	Rightward pointing triangle.
	8	Cross.
	9	Diagonal cross.
	10	Horizontal line.
	11	Vertical line.
	12	Star 1.
	13	Star 2.
	14	Hexagon.
symbolWidth	Scalar, width to draw line symbols.	

Example

```
    //Declare plotControl structure
    struct plotControl myPlot;
```

```
//Initialize plotControl structure
myPlot = plotGetDefaults("xy");

//Set line 1 to have no symbol
//Set line 2 to display an ellipse at each plotted point.
newSymbol = { -1, 0 };
symbolWidth = 5;
plotSetLineSymbol(&myPlot, newSymbol, symbolWidth);

//Create data
x = seqa(0.1, 0.1, 50);
y = sin(x)~cos(x);

//Plot the data with the new line symbols
plotXY(myPlot, x, y);
```

Remarks

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotSetXLabel, plotSetLineColor

plotSetLineThickness

Purpose

Sets the thickness of the lines on a graph.

Format

`plotSetLineThickness`(*`&myPlot`*, *`newTh`*);

Input

`&myPlot`	A **plotControl** structure pointer.
`newTh`	1 x N matrix, new line thickness settings.

Example

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
myPlot = plotGetDefaults("xy");

//Set all lines to have a thickness of 2
newTh = 2;
plotSetLineThickness(&myPlot, newTh);

//Create data
x = seqa(0.1, 1, 50);
y = sin(x)~cos(x);

//Plot the data with the new line thickness settings
plotXY(myPlot, x, y);
```

Remarks

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotLayout, plotSetTitle

plotSetNewWindow

Purpose

Determines whether each new graph is drawn in a new graph tab or re-uses a pre-existing graph tab.

Format

plotSetNewWindow(*&myPlot*, *newW*);

Input

&myPlot	A **plotControl** structure pointer.
newW	Scalar, 1 to create a new graph tab or 0 to re-use.

Example

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
```

```
    myPlot = plotGetDefaults("xy");

    //Set graph to create a new graph tab
    newW = 1;
    plotSetNewWindow(&myPlot, newW);

    //Create data
    x = seqa(0.1, 1, 50);
    y = sin(x)~cos(x);

    //Plot the data in a new graph tab window
    plotXY(myPlot, x, y);
```

Remarks

To open a new graph window once, use **plotOpenWindow**. This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible in the main application window from the **Tools->Graphics>Preferences** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotOpenWindow, plotSetTitle, plotSetLineColor

plotSetTitle

Purpose

Controls the settings for the title for a graph.

Format

```
    plotSetTitle(&myPlot, title, font, fontSize, fontColor);
    plotSetTitle(&myPlot, title, font);
    plotSetTitle(&myPlot, title);
```

Input

&myPlot	A **plotControl** structure pointer.
title	String, the new title. This may contain HTML for the creation of Greek letters, mathematical symbols and text formatting.
font	String, font or font family name.
fontSize	Scalar, font size in points.
fontColor	String, named color or RGB value.

Example

Example 1

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
myPlot = plotGetDefaults("hist");

//Set the title, title font and title font size
plotSetTitle(&myPlot, "GAUSS Example Graph",
"verdana", 10);

//Create data
x = rndn(1e5,1);

//Plot a histogram of the x data spread over 50 bins
plotHist(myPlot, x, 50);
```

Example 2

You may add Greek letters, mathematical symbols, subscript and superscript to your title using HTML. To add HTML to a label, you need to wrap the text to be interpreted as HTML in HTML tags.

```
label_string = "<html>&beta;</html>";
plotSetTitle(&myPlot, label_string);
```

The code above will add the letter β to the graph title. The HTML 'sup' tag will create superscript and the 'sub' tag will create subscript. For example:

```
label_string = "<html>&sigma;<sup>2</sup></html>";
plotSetTitle(&myPlot, label_string);
```

will add σ^2 to your title. While,

```
    label_string = "<html>Y<sub>t-1</sub></html>";
    plotSetTitle(&myPlot, label_string);
```

will create Y_{t-1}.

Remarks

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotSetYLabel, plotSetLineColor, plotSetGrid

plotSetXLabel

Purpose

Controls the settings for the X-axis label on a graph.

Format

```
plotSetXLabel(&myPlot, label, font, fontSize,
fontColor);
plotSetXLabel(&myPlot, label, font, fontSize);
plotSetXLabel(&myPlot, label, font);
plotSetXLabel(&myPlot, label);
```

Input

&myPlot	A **plotControl** structure pointer.
label	String, the new label. This may contain HTML for the creation of Greek letters, mathematical symbols and text formatting.
font	String, font or font family name.
fontSize	Scalar, font size in points.
fontColor	String, named color or RGB value.

Example

Example 1

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
myPlot = plotGetDefaults("hist");

//Set the X-axis label, label font, label font size, and
//label color
plotSetXLabel(&myPlot, "Time (sec)", "verdana", 10, "bl
ack");

//Create data
x = rndn(1e5,1);

//Plot a histogram of the x data spread over 50 bins
plotHist(myPlot, x, 50);
```

Example 2

You may add Greek letters, mathematical symbols, subscript and superscript to your axis labels using HTML. To add HTML to a label, you need to wrap the text to be interpreted as HTML in HTML tags.

```
label_string = "<html>&beta;</html>";
plotSetXLabel(&myPlot, label_string);
```

The code above will add the letter β to the x-axis label. The HTML 'sup' tag will create superscript and the 'sub' tag will create subscript. For example:

```
label_string = "<html>&sigma;<sup>2</sup></html>";
plotSetXLabel(&myPlot, label_string);
```

will add σ^2 to your x-axis label. While,

```
    label_string = "<html>Y<sub>t-1</sub></html>";
    plotSetXLabel(&myPlot, label_string);
```

will create Y_{t-1}.

Remarks

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu.See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotSetXTicInterval, plotSetXTicLabel, plotSetYLabel, plotSetZLabel, plotSetLineColor, plotSetGrid

p

plotSetXTicInterval

Purpose

Controls the interval between X-axis tic labels and also allows the user to specify the first tic to be labeled for 2-D time series graphs.

Format

```
plotSetXTicInterval(&myPlot, ticInterval,
firstLabeled);
plotSetXTicInterval(&myPlot, ticInterval);
```

Input

&myPlot	A **plotControl** structure pointer.
ticInterval	Scalar, the number of X-values between X-axis tic labels.
firstLabeled	Scalar, the value of the first X-value on which to place a tic label.

Example

Time Series Example

```
//Declare and initialize plotControl structure
struct plotControl myPlot;
myPlot = plotGetDefaults("xy");

//Place one tic label every 4 x-values
ticInterval = 4;
plotSetXTicInterval(&myPlot, ticInterval);

//Start the time series in April of 2008
dtstart = 200804;

//Specify quarterly data
frequency = 4;

//Create the multiplicative sequence 1, 2, 4, 8...
y = seqm(1, 2, 10);

//Create a time series plot of the data.
plotTS(myPlot, dtstart, frequency, y);
```

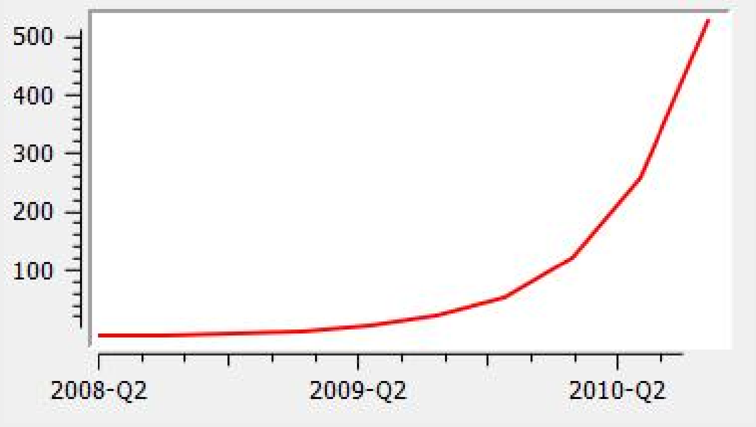

Figure 38.1: Time series plot example

If you would like to change the tic labels so that they start on the first full year, 2009, continuing with the example from above, execute the following lines:

```
    //Set the optional 'firstLabeled' parameter
    plotSetXTicInterval(&myPlot, ticInterval, 2009);
    plotTS(myPlot, dtstart, frequency, y);
```

This new plot should now have tic labels only on the first quarters of each year:

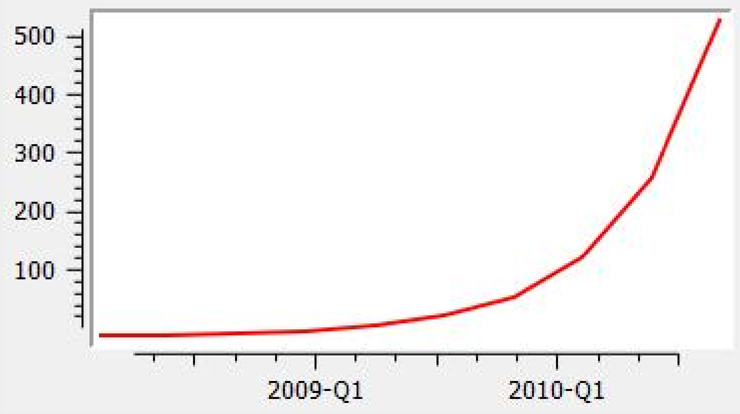

Figure 38.2: Revised time series example

Remarks

`plotSetXTicInterval` is currently only supported for use with time series plots. It is ignored by other plot types.

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

dttostr, strtodt, plotSetXLabel, plotSetXTicLabel

plotSetXTicLabel

Purpose

Controls the formatting and angle of X-axis tic labels for 2-D time series graphs.

Format

```
plotSetXTicLabel(&myPlot, fmt, angle);
plotSetXTicLabel(&myPlot, fmt);
```

Input

&myPlot	A **plotControl** structure pointer.
fmt	String, the desired formatting for the X-axis tic labels. See function **dttostr** for formatting options.
angle	Scalar, the angle in degrees at which to display the X-axis tic labels.

Example

Example 1

```
//Declare and initialize plotControl structure
struct plotControl myPlot;
myPlot = plotGetDefaults("xy");

y = rndn(5, 1);
```

```
    plotSetXTicLabel(&myPlot, "YYYY-MO");

    //Start the series in January 1982
    dtstart = 198201;

    //Specify the data to be monthly
    frequency = 12;

    //Draw the time series plot
    plotTS(myPlot, dtstart, frequency, y);
```

The code above produces a graph with X-tic labels like the image below:

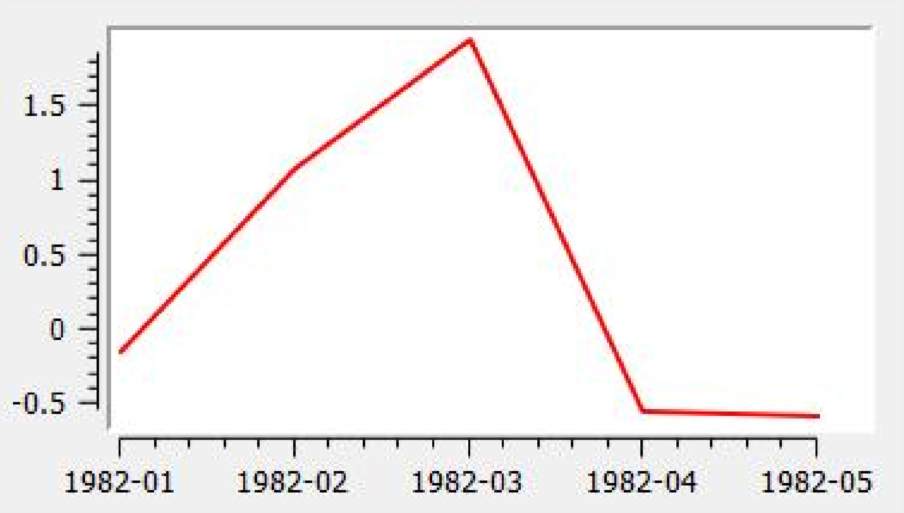

Figure 38.3: Time series plot example

Changing to format string to "MO/YYYY" will change the labels to appear like this:

```
    01/1982
```

Changing to format string to "YYYY-QQ" will change the the labels to appear like this:

```
    1982-Q1
```

Remarks

This function is currently only supported for time series graphs created with **`plotTS`**.

The following format specifiers are supported for the second input to **`plotSetXTicLabel`**, *`fmt`*:

`YYYY`	4 digit year
`YR`	Last two digits of year
`QQ`	Quarter of the year. This is calculated from the month number.
`MO`	Number of month, 01-12
`DD`	Day of month, 01-31
`HH`	Hour of day, 00-23
`MI`	Minute of hour, 00-59
`SS`	Second of minute, 00-59

To learn more about DT scalar format and the using the format specifiers, see **Date and Time Formats**, Section 1.0.1, or the functions **`dttostr`** and **`strtodt`**.

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

dttostr, strtodt, plotSetXLabel, plotSetXTicInterval

plotSetYLabel

Purpose

Controls the settings for the Y-axis label on a graph.

Format

```
plotSetYLabel(&myPlot, label, font, fontSize,
fontColor);
plotSetYLabel(&myPlot, label, font, fontSize);
plotSetYLabel(&myPlot, label, font);
plotSetYLabel(&myPlot, label);
```

Input

&myPlot	A **plotControl** structure pointer.
label	String, the new label. This may contain HTML for the creation of Greek letters, mathematical symbols and text formatting.
font	String, font or font family name.
fontSize	Scalar, font size in points.
fontColor	String, named color or RGB value.

Example

Example 1

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
myPlot = plotGetDefaults("hist");

//Set the Y-axis label, label font, font size, and color
plotSetYLabel(&myPlot, "Time (sec)", "verdana", 10, "bl
ack");

//Create data
x = rndn(1e5,1);

//Plot a histogram of the x data spread over 50 bins
plotHist(myPlot, x, 50);
```

Example 2

You may add Greek letters, mathematical symbols, subscript and superscript to your axis labels using HTML. To add HTML to a label, you need to wrap the text to be interpreted as HTML in HTML tags.

```
label_string = "<html>&beta;</html>";
plotSetYLabel(&myPlot, label_string);
```

The code above will add the letter β to the y-axis label. The HTML 'sup' tag will create superscript and the 'sub' tag will create subscript. For example:

```
label_string = "<html>&sigma;<sup>2</sup></html>";
plotSetYLabel(&myPlot, label_string);
```

will add σ^2 to your y-axis label. While,

```
    label_string = "<html>Y<sub>t-1</sub></html>";
    plotSetYLabel(&myPlot, label_string);
```

will create Y_{t-1}.

Remarks

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotSetXLabel, plotSetXTicInterval, plotSetXTicLabel, plotSetZLabel, plotSetLineColor, plotSetGrid

plotSetZLabel

Purpose

Controls the settings for the Z-axis label on a graph.

Format

```
plotSetZLabel(&myPlot, label, font, fontSize,
fontColor);
plotSetZLabel(&myPlot, label, font, fontSize);
plotSetZLabel(&myPlot, label, font);
plotSetZLabel(&myPlot, label);
```

Input

&myPlot	A **plotControl** structure pointer.
label	String, the new label. This may contain HTML for the creation of Greek letters, mathematical symbols and text formatting. As of the release of this document, HTML is supported for contour plots, but it is not supported for surface plots.
font	String, font or font family name.
fontSize	Scalar, font size in points.
fontColor	String, named color or RGB value.

Example

Example 1

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize plotControl structure
myPlot = plotGetDefaults("surface");

//Set the Z-axis label, label font, font size, and color
plotSetZLabel(&myPlot, "Depth", "verdana", 10, "black")
;

//Create data
x = seqa(-10.6, .3, 71)';
y = seqa(-12.4, .35, 71);
z = sin(sqrt((x/2)^2+(y/2)^2)) ./ sqrt(x^2+y^4);
```

```
    z = z .* sin(x/3);

    //Plot the data
    plotSurface(myPlot, x, y, z);
```

Example 2

You may add Greek letters, mathematical symbols, subscript and superscript to your axis labels using HTML. To add HTML to a label, you need to wrap the text to be interpreted as HTML in HTML tags.

```
    label_string = "<html>&beta;</html>";
    plotSetZLabel(&myPlot, label_string);
```

The code above will add the letter β to the z-axis label. The HTML 'sup' tag will create superscript and the 'sub' tag will create subscript. For example:

```
    label_string = "<html>&sigma;<sup>2</sup></html>";
    plotSetZLabel(&myPlot, label_string);
```

will add σ^2 to your z-axis label. While,

```
    label_string = "<html>Y<sub>t-1</sub></html>";
    plotSetZLabel(&myPlot, label_string);
```

will create Y_{t-1}.

Remarks

This function sets an attribute in a **plotControl** structure. It does not affect an existing graph, or a new graph drawn using the default settings that are accessible from the **Tools->Preferences->Graphics** menu. See **GAUSS Graphics**, Chapter 1, for more information on the methods available for customizing your graphs.

See Also

plotGetDefaults, plotSetXLabel, plotSetXTicInterval, plotSetXTicLabel, plotSetYLabel, plotSetLineColor, plotSetGrid

plotSurface

Purpose

Graphs a 3-D surface.

Format

```
plotSurface(myPlot, x, y,  z);
plotSurface(x, y,  z);
```

Input

myPlot	A **plotControl** structure.
x	1xK vector, the X axis data.
y	Nx1 vector, the Y axis data.
z	NxK matrix, the matrix of height data to be plotted.

See Also

plotPolar, plotSetBkdColor

plotTS

Purpose

Creates a graph of time series data.

Format

`plotTS`(*`myPlot`*, *`dtstart`*, *`frequency`*, *`y`*);
`plotTS`(*`dtstart`*, *`frequency`*, *`y`*);

Input

`myPlot`	A **plotControl** structure.
`dtstart`	Scalar, starting date in DT scalar format.
`frequency`	Scalar, frequency of the data per year. Valid options include:
	1 Yearly
	4 Quarterly
	12 Monthly
`y`	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

Examples

Example 1

```
//Create some data to plot
```

```
    y = rndn(100, 1);

    //The first input starts the series in January of 1982
    //The second input specifies the data to be monthly
    plotTS(1982, 12, y);
```

Example 2

```
    //The first input starts the series in April 2005
    //The second input specifies the data to be monthly
    plotTS(200504, 12, y);
```

Example 3

In DT Scalar format, quarters are represented by supplying the first month of the quarter for the sixth and seventh leading digits. As we see below, 200504 represents April of 2005, but it also represents the second quarter of April 2005.

```
    //The first input starts the series in the second quarter o
    f 2005
    //The second input specifies the data to be quarterly
    plotTS(200504, 4, y);
```

Remarks

Formatting for the X-tic labels can be set with the function **plotSetXTicLabel**. If a **plotControl** structure is not passed in to **plotTS**, or the format specifier is not set with **plotSetXTicLabel** the default formatting: for annual data is "YYYY", for quarterly data "YYYY-QQ" and for monthly data is "YYYY-MO".

By default missing values in the *y* variable will be represented as gaps in the line.

See Also

plotSetXTicLabel, plotSetXTicInterval, plotScatter

plotXY

Purpose

Graphs X vs. Y using Cartesian coordinates.

Format

```
plotXY(myPlot, x, y);
plotXY(x, y);
```

Input

myPlot	A **plotControl** structure.
x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

Remarks

By default missing values in the *y* variable will be represented as gaps in the line.

See Also

plotLogX, plotLogLog, plotScatter

polar

Purpose

Graph data using polar coordinates. NOTE: This function is for use only with the deprecated PQG graphics.

Library

`pgraph`

Format

```
polar(radius, theta);
```

Input

radius	Nx1 or NxM matrix. Each column contains the magnitude for a particular line.
theta	Nx1 or NxM matrix. Each column represents the angle values for a particular line.

Source

`polar.src`

See Also

xy, logx, logy, loglog, scale, xtics, ytics

polychar

Purpose

Computes the characteristic polynomial of a square matrix.

Format

c = **`polychar`**(*x*);

Input

x	NxN matrix.

Output

c	(N+1)x1 vector of coefficients of the Nth order characteristic polynomial of *x*:

```
p(x) = c[1]*x^n + c[2]*x^(n-1) + ... + c[n]*x + c
[n+1];
```

Remarks

The coefficient of x^n is set to unity (*c*[1]=1).

Source

`poly.src`

See Also

polymake, polymult, polyroot, polyeval

polyeval

Purpose

Evaluates polynomials. Can either be one or more scalar polynomials or a single matrix polynomial.

Format

y = **polyeval**(*x*, *c*);

Input

x	1xK or NxN; that is, *x* can either represent K separate scalar values at which to evaluate the (scalar) polynomial(s), or it can represent a single NxN matrix.
c	(P+1)xK or (P+1)x1 matrix of coefficients of polynomials to evaluate. If *x* is 1xK, then *c* must be (P+1)xK. If *x* is NxN, *c* must be (P+1)x1. That is, if *x* is a matrix, it can only be evaluated at a single set of coefficients.

Output

y	Kx1 vector (if *c* is (P+1)xK) or NxN matrix (if *c* is (P+1)x1 and *x* is NxN):

```
y =( c[1,.].*x^p + c[2,.].*x^(p-1) + ... + c[p+
1,.] )';
```

Remarks

In both the scalar and the matrix case, Horner's rule is used to do the evaluation. In the scalar case, the function **recsercp** is called (this implements an elaboration of Horner's rule).

Example

```
x = 2;
let c = 1 1 0 1 1;
y = polyeval(x,c);
```

The result is 27. Note that this is the decimal value of the binary number 11011.

```
y = polyeval(x,1|zeros(n,1));
```

This will raise the matrix *x* to the *n*th power (e.g: `x*x*x*x*...*x`).

Source

poly.src

See Also

polymake, polychar, polymult, polyroot

polygamma

Purpose

Computes the polygamma function of order *n*.

Format

`f = polygamma(z,n);`

Input

z	NxK matrix; *z* may be complex.
n	The order of the function. If *n* is 2 then *f* will be the Digamma function. If *n* = 3,4,5, etc., then *f* will be the tri-, tetra-, penta-, hexa-, hepta-, etc., Gamma function. Real (*n*) must be positive.

Output

f	NxK matrix; *f* may be complex.

Example

```
    polygamma(-45.6-i*29.4, 101);
```

is near 12.5 + 9*i

```
    polygamma(-11.5-i*0.577007813568142,10);
```

is near a root of the decagamma function

Remarks

This program uses the partial fraction expansion of the derivative of the log of the Lanczos series approximation for the Gamma function. Accurate to about 12 digits.

References

1. C. Lanczos, SIAM JNA 1, 1964. pp. 86-96.
2. Y. Luke, "The Special ... approximations," 1969 pp. 29-31.
3. Y. Luke, "Algorithms ... functions," 1977.
4. J. Spouge, SIAM JNA 31, 1994. pp. 931.
5. W. Press, "Numerical Recipes."
6. S. Chang, "Computation of special functions," 1996.
7. Abramowitz & Stegun, section eq 6.4.6
8. Original code by Paul Godfrey

polyint

Purpose

Calculates an Nth order polynomial interpolation.

Format

y = **polyint**(*xa*, *ya*, *x*);

Input

xa	Nx1 vector, *x* values.
ya	Nx1 vector, *y* values.
x	scalar, *x* value to solve for.

Global Input

_poldeg	scalar, the degree of polynomial required, default 6.

Output

y	result of interpolation or extrapolation.

Global Output

_polerr	scalar, interpolation error.

Remarks

Calculates an Nth order polynomial interpolation or extrapolation of *x* on *y* given the vectors *xa* and *ya* and the scalar *x*. The procedure uses Neville's algorithm to determine an up to Nth order polynomial and an error estimate.

Polynomials above degree 6 are not likely to increase the accuracy for most data. Test *_polerr* to determine the required *_poldeg* for your problem.

Source

`polyint.src`

Technical Notes

Press, W.P., B.P. Flannery, S.A. Teukolsky, and W.T. Vettering. *Numerical Recipes: The Art of Scientific Computing*. NY: Cambridge Press, 1986.

polymake

Purpose

Computes the coefficients of a polynomial given the roots.

Format

c = **polymake**(*r*);

Input

r	Nx1 vector containing roots of the desired polynomial.

Output

c	(N+1)x1 vector containing the coefficients of the Nth order polynomial with roots *r*: $p(z)=c[1]*z^{n} + c[2]*z^{(n-1)} + \dots c[n]*z + c[n+1]$

p

Remarks

The coefficient of z^n is set to unity (c[1]=1).

Example

```
//Assign values for the roots of the polynomial
r = { 2, 1, 3 };

//Calculate the coefficients
c = polymake(r);

//Print 3 spaces for each number and 1 digit after the
//decimal place
format /rd 3,1;

//Iterate through each root in 'r'
for i(1, 3, 1);
   rtmp = r[i];
   //Calculate the polynomial
   rout = c[1]*rtmp^3 + c[2]*rtmp^2 + c[3]*rtmp + c[4];
   print "rtmp = " rtmp "rout = " rout;
endfor;
```

Since the values of *r* are roots for this polynomial, `rout` should equal 0. Thus the code above gives the following output:

```
rtmp = 2.0 rout = 0.0
rtmp = 1.0 rout = 0.0
rtmp = 3.0 rout = 0.0
```

This example assigns *c* to be equal to:

```
     1.0
c = -6.0
```

```
        11.0
        -6.0
```

This represents the polynomial:

$$x^3 - 6x^2 + 11x - 6$$

Source

`poly.src`

See Also

polychar, polymult, polyroot, polyeval

polymat

Purpose

Returns a matrix containing the powers of the elements of *x* from 1 to *p*.

Format

y = **`polymat`**(*x*, *p*);

Input

x	NxK matrix.
p	scalar, positive integer.

Output

y	Nx(*p**K) matrix containing powers of the elements of *x* from 1 to *p*. The first K columns will contain first powers, the second K columns second powers, and so on.

Remarks

To do polynomial regression use **ols**:

```
    { vnam,m,b,stb,vc,stderr,sigma,cx,rsq,resid,dwstat } = ols(
    0,y, polymat(x,p));
```

Source

polymat.src

polymroot

Purpose

Computes the roots of the determinant of a matrix polynomial.

Format

r = **polymroot**(*c*);

Input

c	(N+1)*KxK matrix of coefficients of an Nth order

	polynomial of rank K.

Output

r	K*N vector containing the roots of the determinantal equation.

Remarks

c is constructed of N+1 KxK coefficient matrices stacked vertically with the coefficient matrix of the t^n at the top, $t^{(n-1)}$ next, down to the t^0 matrix at the bottom.

Note that this procedure solves the scalar problem as well, that is, the one that POLYROOT solves.

Example

Solve

```
    det(A2*t^2 + A1*t + A0) = 0
```

where:

```
    A2 =  1  2    A1 =  5  8   A0 = 3  4
          2  1         10  7        6  5
```

```
    a2 = { 1 2, 2 1 };
    a1 = { 5 8, 10 7 };
    a0 = { 3 4, 6 5 };

    //The pipe operator '|' provides vertical concatenation
    print  polymroot(a2|a1|a0);
```

```
     -4.3027756
     -.69722436
     -2.6180340
     -.38196601
```

polymult

Purpose

Multiplies polynomials.

Format

c = **polymult**(*c1*, *c2*);

Input

c1	(D1+1)x1 vector containing the coefficients of the first polynomial.
c2	(D2+1)x1 vector containing the coefficients of the second polynomial.

Output

c	(D1+D2)x1 vector containing the coefficients of the product of the two polynomials.

Example

This example multiplies the polynomials:

```
    (2x + 1)(2x^2 + 1)
```

and returns the answer:

```
    4x^3 + 2x^2 + 2x + 1
```

```
    //Assign c1 to represent 2x + 1
    c1 = { 2, 1 };

    //Assign c2 to represent 2x^2 + 1
    c2 = { 2, 0, 1 };
    c = polymult(c1,c2);
```

After the code above:

```
        4
    c = 2
        2
        1
```

Technical Notes

If the degree of *c1* is *D1* (e.g., if *D1=3*, then the polynomial corresponding to *c1* is cubic), then there must be *D1+1* elements in *c1* (e.g., 4 elements for a cubic). Thus, for instance the coefficients for the polynomial

```
    5x^3 + 6x + 3
```

would be:

```
    //Using the pipe operator for vertical concatenation
    c1 = 5|0|6|3;
```

```
  or

//Using an array assignment
c1 = { 5, 0, 6, 3 };
```

(Note that zeros must be explicitly given if there are powers of *x* missing.)

Source

`poly.src`

See Also

polymake, polychar, polyroot, polyeval

polyroot

Purpose

Computes the roots of a polynomial given the coefficients.

Format

y = **`polyroot`**(*c*);

Input

c	(N+1)x1 vector of coefficients of an Nth order polynomial: `p(z) = c[1]*z`^{n}` + c[2]*z`$^{n-1}$` + ... + c[n]*z + c[n+1]`

Output

y	Nx1 vector, the roots of *c*.

Remarks

Zero leading terms will be stripped from *c*. When that occurs the order of *y* will be the order of the polynomial after the leading zeros have been stripped.

c[1] need not be normalized to unity.

Source

`poly.src`

See Also

polymake, polychar, polymult, polyeval

pop

Purpose

Provides access to a last-in, first-out stack for matrices.

Format

```
pop b;
pop a;
```

Remarks

This is used with gosub, goto, and return statements with parameters. It permits passing parameters to subroutines or labels, and returning parameters from subroutines.

The gosub syntax allows an implicit push statement. This syntax is almost the same as that of a standard gosub, except that the matrices to be push'ed "into the subroutine" are in parentheses following the label name. The matrices to be push'ed back to the main body of the program are in parentheses following the return statement. The only limit on the number of matrices that can be passed to and from subroutines in this way is the amount of room on the stack.

No matrix expressions can be executed between the (implicit) push and the pop. Execution of such expressions will alter what is on the stack.

Matrices must be pop'ped in the reverse order that they are push'ed, therefore in the statements:

```
     goto label(x,y,z);
      .
      .
      .
     label:
     pop c;
     pop b;
     pop a;
```

After the code above:

```
     c = z
     b = y
     a = x
```

Note that there must be a separate pop statement for each matrix popped.

See Also

gosub, goto, return

pqgwin

Purpose

Sets the graphics viewer mode. NOTE: This function is for use only with the deprecated PQG graphics.

Library

pgraph

Format

```
pqgwin one;
pqgwin many;
```

Remarks

If you call:

```
pqgwin one
```

only a single viewer will be used. If you call

```
pqgwin many
```

a new viewer will be used for each graph.

pqgwin manual and **pqgwin auto** are supported for backwards compatibility, **manual**=**one**, **auto**=**many**.

Example

```
        pqgwin many;
```

Source

`pgraph.src`

See Also

setvwrmode

previousindex

Purpose

Returns the index of the previous element or subarray in an array.

Format

pi = **previousindex**(*i*, *o*);

Input

i	Mx1 vector of indices into an array, where M <= N.
o	Nx1 vector of orders of an N-dimensional array.

Output

pi	Mx1 vector of indices, the index of the previous element or

subarray in the array corresponding to *o*.

Remarks

previousindex will return a scalar error code if the index cannot be decremented.

Example

```
orders = {3,4,5,6,7};
a = areshape(1,orders);
orders = getorders(a);
ind = { 2,3,1 };
ind = previousindex(ind,orders);
```

After the code above, *ind* is equal to:

```
      2
ind = 2
      5
```

In this example, **previousindex** decremented *ind* to index the previous 6x7 subarray in array *a*.

See Also

nextindex, loopnextindex, walkindex

princomp

Purpose

Computes principal components of a data matrix.

Format

{ *p*, *v*, *a* } = **princomp**(*x*, *j*);

Input

x	NxK data matrix, N>K, full rank.
j	scalar, number of principal components to be computed (*j* <= K).

Output

p	NxJ matrix of the first *j* principal components of *x* in descending order of amount of variance explained.
v	Jx1 vector of fractions of variance explained.
a	JxK matrix of factor loadings, such that:

```
    x = p*a + error.
```

Remarks

Adapted from a program written by Mico Loretan.

The algorithm is based on Theil, Henri "Principles of Econometrics." Wiley, NY, 1971, 46-56.

print

Purpose

Prints matrices, arrays, strings and string arrays to the screen and/or auxiliary output.

Format

`print` [[*/flush*]] [[*/typ*]] [[*/fmted*]] [[*/mf*]] [[*/jnt*]]*list_of_ expressions*[[*;*]];

Input

/typ	literal, symbol type flag.	
	/mat, */sa*, */str*	Indicate which symbol types you are setting the output format for: matrices and arrays (*/mat*), string arrays (*/sa*), and/or strings (*/str*). You can specify more than one */ typ* flag; the format will be set for all types indicated. If no */ typ* flag is listed, `print` assumes */mat*.
/fmted	literal, enable formatting flag.	
	/on, */off*	Enable/disable formatting. When formatting is disabled, the contents of a variable are dumped

p

		to the screen in a "raw" format. `/off` is currently supported only for strings. "Raw" format for strings means that the entire string is printed, starting at the current cursor position. When formatting is enabled for strings, they are handled the same as string arrays. This shouldn't be too surprising, since a string is actually a 1x1 string array.
`/mf`	literal, matrix format. It controls the way rows of a matrix are separated from one another. The possibilities are:	
	`/m0`	no delimiters before or after rows when printing out matrices.
	`/m1 or /mb1`	print 1 carriage return/line feed pair before each row of a matrix with more than 1 row.
	`/m2 or /mb2`	print 2 carriage return/line feed pairs before each row of a matrix with more than 1 row.
	`/m3 or /mb3`	print "Row 1", "Row 2"...before each row of a matrix with more than one row.
	`/ma1`	print 1 carriage return/line feed pair after each row of a matrix with more than 1 row.
	`/ma2`	print 2 carriage return/line feed pairs after each row of a matrix with more than 1 row.
	`/a1`	print 1 carriage return/line feed pair after each row of a matrix.
	`/a2`	print 2 carriage return/line feed pairs after each row of a matrix.

	`/b1`	print 1 carriage return/line feed pair before each row of a matrix.
	`/b2`	print 2 carriage return/line feed pairs before each row of a matrix.
	`/b3`	print "Row 1", "Row 2"... before each row of a matrix.

`/jnt` literal, controls justification, notation, and the trailing character.

Right-Justified

`/rd`	Signed decimal number in the form [[-]] ####.####, where #### is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed.
`/re`	Signed number in the form [[-]]#.##E±###, where # is one decimal digit, ## is one or more decimal digits depending on the precision, and ### is three decimal digits. If precision is 0, the form will be [[-]]#E±### with no decimal point printed.
`/ro`	This will give a format like `/rd` or `/re` depending on which is most compact for the number being printed. A format like `/re` will be used only if the exponent value is less than -4 or greater than the precision. If a `/re` format is used, a decimal point will always appear.

	The precision signifies the number of significant digits displayed.
`/rz`	This will give a format like `/rd` or `/re` depending on which is most compact for the number being printed. A format like `/re` will be used only if the exponent value is less than -4 or greater than the precision. If a `/re` format is used, trailing zeros will be supressed and a decimal point will appear only if one or more digits follow it. The precision signifies the number of significant digits displayed.
Left-Justified	
`/ld`	Signed decimal number in the form [[-]] ####.####, where #### is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after thedecimal point depends on the precision. If the precision is 0, no decimal point will be printed. If the number is positive, a space character will replace the leading minus sign.
`/le`	Signed number in the form [[-]]#.##E±###, where # is one decimal digit, ## is one or more decimal digits depending on the precision, and ### is three decimal digits. If precision is 0, the form will be [[-]]#E±### with no decimal point printed. If the number is positive, a space character will replace the leading minus sign.

`/lo`	This will give a format like `/ld` or `/le` depending on which is most compact for the number being printed. A format like `/le` will be used only if the exponent value is less than -4 or greater than the precision. If a `/le` format is used, a decimal point will always appear. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.
`/lz`	This will give a format like `/ld` or `/le` depending on which is most compact for the number being printed. A format like `/le` will be used only if the exponent value is less than -4 or greater than the precision. If a `/le` format is used, trailing zeros will be supressed and a decimal point will appear only if one or more digits follow it. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.

Trailing Character

The following characters can be added to the **/** *`jnt`* parameters above to control the trailing character if any:

```
                                    format /rdn 1,3;
```

`s`	The number will be followed immediately by a space character. This is the default.

c	The number will be followed immediately by a comma.
t	The number will be followed immediately by a tab character.
n	No trailing character.

The default when **GAUSS** is first started is:

```
format /m1 /ro 16,8;
```

;;	Double semicolons following a `print` statement will suppress the final carriage return/line feed.

list_of_expressions	any **GAUSS** expressions that produce matrices, arrays, stings, or string arrays and/or names of variables to print, separated by spaces.

Remarks

The list of expressions MUST be separated by spaces. In `print` statements, because a space is the delimiter between expressions, NO SPACES are allowed inside expressions unless they are within index brackets, quotes, or parentheses.

The printing of special characters is accomplished by the use of the backslash (\) within double quotes. The options are:

\b	backspace (ASCII 8)
\e	escape (ASCII 27)

`\f`	form feed (ASCII 12)
`\g`	beep (ASCII 7)
`\l`	line feed (ASCII 10)
`\r`	carriage return (ASCII 13)
`\t`	tab (ASCII 9)
`\###`	the character whose ASCII value is "###" (decimal).

Thus, `\13\10` is a carriage return/line feed sequence. The first three digits will be picked up here. So if the character to follow a special character is a digit, be sure to use three digits in the escape sequence. For example: `\0074` will be interpreted as 2 characters (ASCII 7, "4")

An expression with no assignment operator is an implicit `print` statement.

If **`output on`** has been specified, then all subsequent `print` statements will be directed to the auxiliary output as well as the window. (See `output`.) The `locate` statement has no effect on what will be sent to the auxiliary output, so all formatting must be accomplished using tab characters or some other form of serial output.

If the name of the symbol to be printed is prefixed with a `$`, it is assumed that the symbol is a matrix of characters.

```
    print $x;
```

Note that **GAUSS** makes no distinction between matrices containing character data and those containing numeric data, so it is the responsibility of the user to use functions which operate on character matrices only on those matrices containing character data.

These matrices of character strings have a maximum of 8 characters per element. A precision of 8 or more should be set when printing out character matrices or the elements will be truncated.

Complex numbers are printed with the sign of the imaginary half separating them and an "i" appended to the imaginary half. Also, the current field width setting (see `format`) refers to the width of field for each half of the number, so a complex number printed with a field of 8 will actually take (at least) 20 spaces to print.

`print`'ing a sparse matrix results in a table of the non-zero values contained in the sparse matrix, followed by their corresponding row and column indices, respectively.

A `print` statement by itself will cause a blank line to be printed:

```
    print;
```

Example

```
    x = rndn(3,3);
    format /rd 16,8;
    print x;

    format /re 12,2;
    print x;

    print /rd /m3 x;
```

```
          0.14357994  -1.39272762  -0.91942414
          0.51061645  -0.02332207  -0.02511298
         -1.d4675893  -1.04988540   0.07992059

      1.44E-001  -1.39E+000  -9.19E-001
      5.11E-001  -2.33E-002  -2.51E-002
     -1.55E+000  -1.05E+000   7.99E-002

     Row 1
           0.14        -1.39        -0.92
     Row 2
           0.51        -0.02        -0.03
     Row 3
         -1.55        -1.05         0.08
```

In this example, a 3x3 random matrix is printed using 3 different formats. Notice that in the last statement, the format is overridden in the print statement itself but the field and precision remain the same.

```
    let x = AGE PAY SEX;
    format /m1 8,8;
    print $x;
```

```
     AGE
     PAY
     SEX
```

See Also

printfm, printdos

printdos

Purpose

Prints a string to the standard output.

Format

```
printdos s;
```

Input

s	string to be printed to the standard output.

Remarks

This function is useful for printing messages to the screen when **`screen off`** is in effect. The output of this function will not go to the auxiliary output.

This function was used in the past to send escape sequences to the `ansi.sys` device driver on DOS. It still works on some terminals.

Example

```
    printdos "\27[7m"; /* set for reverse video */
    printdos "\27[0m"; /* set for normal text */
```

See Also

print, printfm, screen

p

printfm

Purpose

Prints a matrix using a different format for each column of the matrix.

Format

y = **`printfm`**(*x*, *`mask`*, *`fmt`*);

Input

x	NxK matrix which is to be printed and which may contain both character and numeric data.

mask	LxM matrix, ExE conformable with *x*, containing ones and zeros, which is used to specify whether the particular row, column, or element is to be printed as a character (0) or numeric (1) value.
fmt	Kx3 or 1x3 matrix where each row specifies the format for the respective column of *x*.

Output

y	scalar, 1 if the function is successful and 0 if it fails.

Remarks

The mask is applied to the matrix *x* following the rules of standard element-by-element operations. If the corresponding element of *mask* is 0, then that element of *x* is printed as a character string of up to 8 characters. If mask contains a 1, then that element of *x* is assumed to be a double precision floating point number.

The contents of *fmt* are as follows:

[K,1]	format string,	a string 8 characters maximum.
[K,2]	field width,	a number < 80.
[K,3]	precision,	a number < 17.

The format strings correspond to the `format` slash commands as follows:

/rdn	*"*.*lf"*
/ren	*"*.*lE"*

/ron	*"#*.*lG"*
/rzn	*"*.*lG"*
/ldn	*"- *.*lf"*
/len	*"- *.*lE"*
/lon	*"-# *.*lG"*
/lzn	*"- *.*lG"*

Complex numbers are printed with the sign of the imaginary half separating them and an "i" appended to the imaginary half. The field width refers to the width of field for each half of the number, so a complex number printed with a field of 8 will actually take (at least) 20 spaces to print.

If the precision = 0, the decimal point will be suppressed.

The format string can be a maximum of 8 characters and is appended to a `%` sign and passed directly to the **fprintf** function in the standard C language I/O library. The *lf*, etc., are case sensitive. If you know C, you will easily be able to use this.

If you want special characters to be printed after *x*, then include them as the last characters of the format string. For example:

".*lf,"*	right-justified decimal followed by a comma.
"-.*s "*	left-justified string followed by a space.
".*lf"*	right-justified decimal followed by nothing.

If you want the beginning of the field padded with zeros, then put a "0" before the first "*" in the format string:

"0.*lf"*	right-justified decimal.

Example

Here is an example of **printfm** being used to print a mixed numeric and character matrix:

```
    let x[4,3] = "AGE" 5.12345564 2.23456788
                 "PAY" 1.23456677 1.23456789
                 "SEX" 1.14454345 3.44718234
                 "JOB" 4.11429432 8.55649341;

    let mask[1,3] = 0 1 1;       /* character numeric numeric */
    let fmt[3,3] = "-*.*s " 8 8 /* first column format */
    "*.*lf," 10 3               /* second column format */
    "*.*le " 12 4;              /* third column format */

    d = printfm(x,mask,fmt);
```

The output looks like this:

```
    AGE 5.123, 2.2346E+00
    PAY 1.235, 1.2346E+00
    SEX 1.145, 3.4471E+00
    JOB 4.114, 8.5564E+00
```

When the column of *x* to be printed contains all character elements, use a format string of `"*.*s"` if you want it right-justified, or `"-*.*s"` if you want it left-justified. If the column is mixed character and numeric elements, then use the correct numeric format and **printfm** will substitute a default format string for those elements in the column that are character.

Remember, the mask value controls whether an element will be printed as a number or a character string.

See Also

print, printdos

printfmt

Purpose

Prints character, numeric, or mixed matrix using a default format controlled by the functions **`formatcv`** and **`formatnv`**.

Format

```
y = printfmt(x, mask);
```

Input

x	NxK matrix which is to be printed.
mask	scalar, 1 if *x* is numeric or 0 if *x* is character. - or - 1xK vector of 1's and 0's. The corresponding column of *x* will be printed as numeric where *mask* = 1 and as character where *mask* = 0.

Output

y	scalar, 1 if the function is successful and 0 if it fails.

Remarks

Default format for numeric data is: `''*.*lg '' 16 8`

Default format for character data is: `''*.*s '' 8 8`

Example

```
c1 = { "age", "height", "weight" };
c2 = { 31, 70, 160 };

//Horizontally concatenate c1 and c2
c = c1~c2;

//Print 'c' as numeric data
print c;

//Print 'c' as character data
print $c;

//Print column 1 of 'c' as character data and column 2 as
//numeric data
//Note: call disregards the return value
mask = { 0 1 };
call printfmt(c, mask);
```

The output from the three different print statements will be:

```
+DEN        31.000000
+DEN        70.000000
+DEN        160.00000
```

```
age
height
weight
```

```
    age                 31
    height              70
    weight             160
```

Only the final print statement from **`printfmt`** correctly prints both columns.

Source

`gauss.src`

Globals

__*fmtcv*, __*fmtnv*

See Also

formatcv, formatnv

p

proc

Purpose

Begins the definition of a multi-line recursive procedure. Procedures are user-defined functions with local or global variables.

Format

```
proc nrets = name(arglist);
proc name(arglist);
```

Input

nrets	constant, number of objects returned by the procedure. If *nrets* is not explicitly given, the default is 1. Legal values are 0 to 1023. The **retp** statement is used to return values from a procedure.
name	literal, name of the procedure. This name will be a global symbol.
arglist	a list of names, separated by commas, to be used inside the procedure to refer to the arguments that are passed to the procedure when the procedure is called. These will always be local to the procedure, and cannot be accessed from outside the procedure or from other procedures.

Remarks

A procedure definition begins with the `proc` statement and ends with the `endp` statement.

An example of a procedure definition is:

```
proc dog(x,y,z); /* procedure declaration */
local a,b;        /* local variable declarations */
   a = x .* x;
   b = y .* y;
   a = a ./ x;
   b = b ./ y;
   z = z .* z;
   z = inv(z);
   retp(a'b*z);  /* return with value of a'b*z */
```

```
endp;                  /* end of procedure definition */
```

Procedures can be used just as if they were functions intrinsic to the language. Below are the possible variations depending on the number of items the procedure returns.

Returns 1 item:

```
     y = dog(i,j,k);
```

Returns multiple items:

```
     { x,y,z } = cat(i,j,k);
```

Returns no items:

```
     fish(i,j,k);
```

If the procedure does not return any items or you want to discard the returned items:

```
     call dog(i,j,k);
```

Procedure definitions may not be nested.

For more details on writing procedures, see **Procedures and Keywords**, Chapter 1.

See Also

keyword, call, endp, local, retp

prodc

Purpose

Computes the products of all elements in each column of a matrix.

Format

y = **prodc**(*x*);

Input

x	NxK matrix.

Output

y	Kx1 matrix containing the products of all elements in each column of *x*.

Remarks

To find the products of the elements in each row of a matrix, transpose before applying **prodc**. If *x* is complex, use the bookkeeping transpose (`.'`).

To find the products of all of the elements in a matrix, use the **vecr** function before applying **prodc**.

Example

```
    x = { 1  2  3,
          4  5  6,
          7  8  9 };

    y = prodc(x);
```

The code above assigns *y* to be equal to:

```
        28
y =   80
       162
```

See Also

sumc, meanc, stdc

psi

Purpose

Computes the Psi (or Digamma) function.

Format

f = **psi**(*z*);

Input

z	NxK matrix; *z* may be complex.

Output

f	NxK matrix.

Remarks

This program uses the analytical derivative of the log of the Lanczos series approximation for the Gamma function.

References

1. C. Lanczos, SIAM JNA 1, 1964. pp. 86-96.
2. Y. Luke, "The Special ... approximations," 1969 pp. 29-31.
3. Y. Luke, "Algorithms ... functions," 1977.
4. J. Spouge, SIAM JNA 31, 1994. pp. 931.
5. W. Press, "Numerical Recipes."
6. S. Chang, "Computation of special functions," 1996.
7. Original code by Paul Godfrey

putarray

Purpose

Puts a contiguous subarray into an N-dimensional array and returns the resulting array.

Format

y = **putarray**(*a*, *loc*, *src*);

Input

a	N-dimensional array.
loc	Mx1 vector of indices into the array to locate the subarray of

	interest, where M is a value from 1 to N.
src	[N-M]-dimensional array, matrix, or scalar.

Output

y	N-dimensional array.

Remarks

If *loc* is an Nx1 vector, then *src* must be a scalar. If *loc* is an [N-1]x1 vector, then *src* must be a 1-dimensional array or a 1xL vector, where L is the size of the fastest moving dimension of the array. If *loc* is an [N-2]x1 vector, then *src* must be a KxL matrix, or a KxL 2-dimensional array, where K is the size of the second fastest moving dimension.

Otherwise, if *loc* is an Mx1 vector, then *src* must be an [N-M]-dimensional array, whose dimensions are the same size as the corresponding dimensions of array *a*.

Example

```
//Create a 2x3x4x5x6 dimensional array with unspecified
//contents
a = arrayalloc(2|3|4|5|6,0);

//Create a 4x5x6 dimensional array with all elements equal
//to 5
src = arrayinit(4|5|6,5);

loc = { 2,1 };
a = putarray(a,loc,src);
```

This example sets the contiguous 4x5x6 subarray of *a* beginning at [2,1,1,1,1] to the array *src*, in which each element is set to the specified value 5.

See Also

setarray

putf

Purpose

Writes the contents of a string to a file.

Format

```
ret = putf(filename, str, start, len, mode, append);
```

Input

filename	string, name of output file.
str	string to be written to *filename*. All or part of *str* may be written out.
start	scalar, beginning position in *str* of output string.
len	scalar, length of output string.
mode	scalar, output mode, (0) ASCII or (1) binary.
append	scalar, file write mode, (0) overwrite or (1) append.

Output

ret	scalar, return code.	
	0	normal return
	1	null file name
	2	file open error
	3	file write error
	4	output string too long
	5	null output string, or illegal *mode* value
	6	illegal *append* value
	16	(1) append specified but file did not exist; file was created (warning only)

Remarks

If *mode* is set to (1) binary, a string of length *len* will be written to *filename*. If *mode* is set to (0) ASCII, the string will be output up to length *len* or until **putf** encounters a ^Z (ASCII 26) in *str*. The ^Z will not be written to *filename*.

If *append* is set to (0) overwrite, the current contents of *filename* will be destroyed. If *append* is set to (1) append, *filename* will be created if it does not already exist.

If an error occurs, **putf** will either return an error code or terminate the program with an error message, depending on the `trap` state. If bit 2 (the 4's bit) of the trap flag is

0, **`putf`** will terminate with an error message. If bit 2 of the trap flag is 1, **`putf`** will return an error code. The value of the trap flag can be tested with **`trapchk`**.

Source

`putf.src`

See Also

getf

putvals

Purpose

Inserts values into a matrix or N-dimensional array.

Format

y = **`putvals`**(*x*, *`inds`*, *`vals`*);

Input

x	MxK matrix or N-dimensional array.
`inds`	LxD matrix of indices, specifying where the new values are to be inserted, where D is the number of dimensions in *x*.
`vals`	Lx1 vector, new values to insert.

Output

y	MxK matrix or N-dimensional array, copy of *x* containing the new values in *vals*.

Remarks

If *x* is a vector, *inds* should be an Lx1 vector. If *x* is a matrix, *inds* should be an Lx2 matrix. Otherwise if *x* is an N-dimensional array, *inds* should be an LxN matrix.

putvals allows you to insert multiple values into a matrix or N-dimensional array at one time. This could also be accomplished using indexing inside a `for` loop.

Example

```
x = { -0.8750  0.3616  0.6032 -0.3974,
       0.7644 -1.8509 -0.2703 -0.8190,
       0.7886  1.2678 -1.4998 -0.5876,
       0.6639 -0.7972  1.2713  0.1896,
       0.6303  0.7879 -0.7451 -0.5419 };
inds = { 1 1, 2 4, 3 2, 3 4, 5 3 };
v = seqa(1,1,5);
y = putvals(x,inds,v);
```

After the code above:

```
    1.000  0.362  0.603 -0.397      1.00
    0.764 -1.851 -0.270  2.000      2.00
y = 0.789  3.000 -1.500  4.000  v = 3.00
    0.664 -0.797  1.271  0.190      4.00
    0.630  0.788  5.000 -0.542      5.00
```

pvCreate

Purpose

Returns an initialized instance of structure of type **PV**.

Format

p1 = **pvCreate**();

Output

p1	an instance of structure of type **PV**

Example

```
//Declare 'p1' as an instance of a 'PV' structure
struct PV p1;

//Fill in 'p1' with default values
p1 = pvCreate();
```

Source

pv.src

pvGetIndex

Purpose

Gets row indices of a matrix in a parameter vector.

Format

```
id = pvGetIndex(p1, nm1);
```

Input

p1	an instance of structure of type **PV**.
nm1	name or row number of matrix.

Output

id	Kx1 vector, row indices of matrix described by *nm1* in parameter vector.

Source

`pv.src`

pvGetParNames

Purpose

Generates names for parameter vector stored in structure of type **PV**.

Include

`pv.sdf`

Format

s = **`pvGetParNames`**(*p1*);

Input

p1	an instance of structure of type **PV**.

Output

s	Kx1 string array, names of parameters.

Remarks

If the vector in the structure of type **PV** was generated with matrix names, the parameter names will be concatenations of the matrix name with row and column numbers of the parameters in the matrix. Otherwise the names will have a generic prefix with concatenated row and column numbers.

Example

```
//Define PV structure#include pv.sdf
//Declare 'p1' as an instance of a 'PV' structure
struct PV p1;

//Initialize 'p1' with default values
p1 = pvCreate;
```

```
//Data to pack into the 'PV' struct
x = { 1 2,
      3 4 };

//1's indicate an element to pack into the structure
//0's indicate elements to NOT pack into the structure
mask = { 1 0,
         0 1 };

//Pack values of 'x' selected by 'mask' into 'pi' and name
//this resulting vector, 'P'
p1 = pvPackm(p1,x,"P",mask);

print pvGetParNames(p1);
```

Since *mask* has ones in the [1,1] and [2,2] locations, the code above, produces:

```
 P[1,1]
 P[2,2]
```

Source

pv.src

pvGetParVector

Purpose

Retrieves parameter vector from structure of type **PV**.

Include

pv.sdf

Format

p = **pvGetParVector**(*p1*);

Input

p1	an instance of structure of type **PV**.

Output

p	Kx1 vector, parameter vector.

Remarks

Matrices or portions of matrices (stored using a mask) are stored in the structure of type **PV** as a vector in the *p* member.

Example

```
//Define 'PV' structure
#include pv.sdf
//Declare 'p1' as an instance of a 'PV' structure
struct PV p1;

//Initialize 'p1' with default values
p1 = pvCreate;

x = { 1 2,
      3 4 };

//1's indicate elements to pack into 'p1' parameter vector
```

```
    mask = { 1 1,
             0 0 };

    p1 = pvPackm(p1,x,"X",mask);

    print pvUnpack(p1,"X");
```

pvUnpack returns the entire value of *x* that was packed in. Therefore, the print statement above, produces:

```
     1.000 2.000
     3.000 4.000
```

```
    print pvGetParVector(p1);
```

pvGetParVector returns only those elements indicated by the *mask* variable and therefore the `print` statement above, returns:

```
     1.000
     2.000
```

Source

pv.src

pvLength

Purpose

Returns the length of a parameter vector.

Format

n = **pvLength**(*p1*);

Input

p1	an instance of structure of type **PV**.

Output

n	scalar, length of parameter vector in *p1*.

Source

pv.src

pvList

Purpose

Retrieves names of packed matrices in structure of type **PV**.

Format

n = **pvList**(*p1*);

Input

p1	an instance of structure of type **PV**.

Output

n	Kx1 string vector, names of packed matrices.

Source

`pv.src`

pvPack

Purpose

Packs general matrix into a structure of type **PV** with matrix name.

Include

`pv.sdf`

Format

p1 = **`pvPack`**(*p1*,*x*, *nm*);

Input

p1	an instance of structure of type **PV**.
x	MxN matrix or N-dimensional array.
nm	string, name of matrix/array.

Output

p1	an instance of structure of type **PV**.

Example

```
//Define the 'PV' structure
#include pv.sdf

y = rndn(100,1);
x = rndn(100,5);

//Declare 'p1' as an instance of a 'PV' structure
struct PV p1;

//Initialize 'p1' with default values
p1 = pvCreate;

p1 = pvPack(p1,x,"Y");
p1 = pvPack(p1,y,"X");
```

These matrices can be extracted using the **pvUnpack** command:

```
y = pvUnpack(p1,"Y");
x = pvUnpack(p1,"X");
```

Source

pv.src

See Also

pvPackm, pvPacks, pvUnpack

pvPacki

Purpose

Packs general matrix or array into a **PV** instance with name and index.

Include

`pv.sdf`

Format

p1 = **`pvPacki`**(*p1*, *x*, *nm*, *i*);

Input

p1	an instance of structure of type **PV**.
x	MxN matrix or N-dimensional array.
nm	string, name of matrix or array, or null string.
i	scalar, index of matrix or array in lookup table.

Output

p1	an instance of structure of type **PV**.

Example

```
//Define the 'PV' structure
#include pv.sdf
```

```
y = rndn(100,1);
x = rndn(100,5);

//Declare 'p1' as an instance of a 'PV' structure
struct PV p1;

//Initialize 'p1' with default values
p1 = pvCreate;

//Pack the variables in with a variable name and an index
p1 = pvPacki(p1,y,"Y",1);
p1 = pvPacki(p1,x,"X",2);
```

These matrices can be extracted using the **`pvUnpack`** command, indicating the variable to unpack either by index or by variable name:

```
//Unpack variables by index
y = pvUnpack(p1,1);
x = pvUnpack(p1,2);

//Unpack variables by variable name
y = pvUnpack(p1,"Y");
x = pvUnpack(p1,"X");
```

See Also

pvPack, pvUnpack

pvPackm

Purpose

Packs general matrix into a structure of type **PV** with a mask and matrix name.

Include

`pv.sdf`

Format

p1 = **pvPackm**(*p1*, *x*, *nm*, *mask*);

Input

p1	an instance of structure of type **PV**.
x	MxN matrix or N-dimensional array.
nm	string, name of matrix/array or N-dimensional array.
mask	MxN matrix, mask matrix of zeros and ones.

Output

p1	an instance of structure of type **PV**.

Remarks

The *mask* argument allows storing a selected portion of a matrix into the packed vector. The ones in `mask` indicate an element to be stored in the packed matrix. When the matrix is unpacked (using **pvUnpack**) the elements corresponding to the zeros are restored. Elements corresponding to the ones come from the packed vector which may have been changed.

If the mask is all zeros, the matrix or array is packed with the specified elements in the second argument but no elements of the matrix or array are entered into the parameter vector. When unpacked the matrix or array in the second argment is returned without modification.

Example

```
#include pv.sdf
struct PV p1;
p1 = pvCreate;

x = { 1 2,
      3 4 };

mask = { 1 0,
         0 1 };

p1 = pvPackm(p1,x,"X",mask);

print pvUnpack(p1,1);
```

```
  1.000 2.000
  3.000 4.000
```

```
p1 = pvPutParVector(p1,5|6);
```

```
print pvUnpack(p1,"X");
```

```
      5.000 2.000
      3.000 6.000
```

Source

`pv.src`

pvPackmi

Purpose

Packs general matrix or array into a **PV** instance with a mask, name, and index.

Include

`pv.sdf`

Format

p1 = **pvPackmi**(*p1*, *x*, *nm*, *mask*, *i*);

Input

p1	an instance of structure of type **PV**.
x	MxN matrix or N-dimensional array.
nm	string, matrix or array name.

mask	MxN matrix or N-dimensional array, *mask* of zeros and ones.
i	scalar, index of matrix or array in lookup table.

Output

p1	an instance of structure of type **PV**.

Remarks

The *mask* allows storing a selected portion of a matrix into the parameter vector. The ones in the *mask* matrix indicate an element to be stored in the parameter matrix. When the matrix is unpacked (using **pvUnpackm**) the elements corresponding to the zeros are restored. Elements corresponding to the ones come from the parameter vector.

If the mask is all zeros, the matrix or array is packed with the specified elements in the second argument but no elements of the matrix or array are entered into the parameter vector. When unpacked the matrix or array in the second argment is returned without modification.

Example

```
    #include pv.sdf

    struct PV p1;
    p1 = pvCreate;

    x = { 1 2,
          3 4 };
    mask = { 1 0,
             0 1 };

    p1 = pvPackmi(p1,x,"X",mask,1);

    print pvUnpack(p1,1);
```

```
     1.000 2.000
     3.000 4.000
```

```
    p1 = pvPutParVector(p1,5|6);

    print pvUnpack(p1,1);
```

```
     5.000 2.000
     3.000 6.000
```

See Also

pvPackm, pvUnpack

pvPacks

Purpose

Packs symmetric matrix into a structure of type **PV**.

Include

```
pv.sdf
```

Format

p1 = **pvPacks**(*p1*, *x*, *nm*);

Input

p1	an instance of structure of type **PV**.
x	MxM symmetric matrix.
nm	string, matrix name.

Output

p1	an instance of structure of type **PV**.

Remarks

pvPacks does not support the packing of arrays.

Example

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2,
      2 1 };

p1 = pvPacks(p1,x, "A");
```

```
    p1 = pvPacks(p1, eye(2), "I");
```

These matrices can be extracted using the **pvUnpack** command:

```
    print pvUnpack(p1, "A");
```

```

       1.000 2.000
       2.000 1.000
```

```
    print pvUnpack(p1, "I");
```

```
       1.000 0.000
       0.000 1.000
```

Source

```
pv.src
```

See Also

pvPacksm, pvUnpack

pvPacksi

Purpose

Packs symmetric matrix into a **PV** instance with matrix name and index.

Include

```
pv.sdf
```

Format

p1 = **pvPacksi**(*p1*, *x*, *nm*, *i*);

Input

p1	an instance of structure of type **PV**.
x	MxM symmetric matrix.
nm	string, matrix name.
i	scalar, index of matrix in lookup table.

Output

p1	an instance of structure of type **PV**.

Remarks

pvPacksi does not support the packing of arrays.

Example

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2, 2 1 };

p1 = pvPacksi(p1,x, "A",1);
```

```
    p1 = pvPacksi(p1, eye(2), "I",2);
```

These matrices can be extracted using the **pvUnpack** command.

```
    print pvUnpack(p1,1);
```

```
      1.000 2.000
      2.000 1.000
```

```
    print pvUnpack(p1,2);
```

```
      1.000 0.000
      0.000 1.000
```

See Also

pvPacks, pvUnpack

pvPacksm

Purpose

Packs symmetric matrix into a structure of type **PV** with a mask.

Include

pv.sdf

Format

p1 = **pvPacksm**(*p1*, *x*, *nm*, *mask*);

Input

p1	an instance of structure of type **PV**.
x	MxM symmetric matrix.
nm	string, matrix name.
mask	MxM matrix, mask matrix of zeros and ones.

Output

p1	an instance of structure of type **PV**.

Remarks

pvPacksm does not support the packing of arrays.

The mask allows storing a selected portion of a matrix into the packed vector. The ones in *mask* indicate an element to be stored in the packed matrix. When the matrix is unpacked (using **pvUnpack**) the elements corresponding to the zeros are restored. Elements corresponding to the ones come from the packed vector which may have been changed.

Only the lower left portion of the *mask* matrix is used, and only the lower left portion of the *x* matrix is stored in the packed vector.

If the mask is all zeros, the matrix is packed with the specified elements in the second argument but no elements of the matrix are entered into the parameter vector. When unpacked the matrix in the second argment is returned without modification.

Example

```
    #include pv.sdf
```

```
struct PV p1;
p1 = pvCreate;

x = { 1 2 4,
      2 3 5,
      4 5 6};

mask = { 1 0 1,
         0 1 0,
         1 0 1 };

p1 = pvPacksm(p1, x, "A", mask);

print pvUnpack(p1, "A");
```

```
  1.000 2.000 4.000
  2.000 3.000 5.000
  4.000 5.000 6.000
```

```
p2 = pvGetParVector(p1);

print p2;
```

```
  1.000
  3.000
  4.000
  6.000
```

```
p3 = { 10, 11, 12, 13 };
p1 = pvPutParVector(p1,p3);

print pvUnpack(p1, "A");
```

```
      10.000  2.000 12.000
       2.000 11.000  5.000
      12.000  5.000 13.000
```

Source

`pv.src`

pvPacksmi

Purpose

Packs symmetric matrix into a **PV** instance with a mask, matrix name, and index.

Include

`pv.sdf`

Format

p1 = **`pvPacksmi`**(*p1*, *x*, *nm*, *mask*, *i*);

Input

p1	an instance of structure of type **PV**.
x	MxM symmetric matrix.
nm	string, matrix name.
mask	MxM matrix, symmetric mask matrix of zeros and ones.

i	scalar, index of matrix in lookup table.

Output

p1	an instance of structure of type **PV**.

Remarks

pvPacksmi does not support the packing of arrays.

The *mask* allows storing a selected portion of a matrix into the parameter vector. The ones in the *mask* matrix indicate an element to be stored in the parameter vector. When the matrix is unpacked (using **pvUnpackm**) the elements corresponding to the zeros are restored. Elements corresponding to the ones come from the parameter vector.

Only the lower left portion of the *mask* matrix is used, and only the lower left portion of the *x* matrix is stored in the packed vector.

If the mask is all zeros, the matrix is packed with the specified elements in the second argument but no elements of the matrix are entered into the parameter vector. When unpacked the matrix in the second argment is returned without modification.

Example

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2 4,
      2 3 5,
          4 5 6};

    mask = { 1 0 1,
             0 1 0,
             1 0 1 };

    p1 = pvPacksmi(p1,x, "A",mask,1);

    print pvUnpack(p1,1);
```

```
      1.000 2.000 4.000
      2.000 3.000 5.000
      4.000 5.000 6.000
```

```
    p2 = pvGetParVector(p1);

    print p2;
```

```
      1.000
      3.000
      4.000
      6.000
```

```
    p3 = { 10, 11, 12, 13 };
    p1 = pvPutParVector(p1,p3);

    print pvUnpack(p1,1);
```

```
      10.000  2.000 12.000
       2.000 11.000  5.000
      12.000  5.000 13.000
```

See Also

pvPacksm, pvUnpack

pvPutParVector

Purpose

Inserts parameter vector into structure of type **PV**.

Include

```
pv.sdf
```

Format

p1 = **`pvPutParVector`**(*p1*, *p*);

Input

p1	an instance of structure of type **PV**.
p	Kx1 vector, parameter vector.

Output

p1	an instance of structure of type **PV**.

Remarks

Matrices or portions of matrices (stored using a *mask*) are stored in the structure of type **PV** as a vector in the *p* member.

Example

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2 4,
      2 3 5,
      4 5 6};

mask = { 1 0 1,
         0 1 0,
         1 0 1 };

//Packed as square matrix
p1 = pvPackm(p1,x,"A",mask);

print pvUnpack(p1,"A");
```

```
  1.000 2.000 4.000
  2.000 3.000 5.000
  4.000 5.000 6.000
```

```
p3 = { 10, 11, 12, 13, 14 };
p1 = pvPutParVector(p1,p3);

print pvUnpack(p1,"A");
```

```
  10.000  2.000 11.000
   2.000 12.000  5.000
  13.000  5.000 14.000
```

Source

pv.src

pvTest

Purpose

Tests an instance of structure of type **PV** to determine if it is a proper structure of type **PV**.

Format

i = **pvTest**(*p1*);

Input

p1	an instance of structure of type **PV**.

Output

i	scalar, if 0, *p1* is a proper structure of type **PV**, else if 1, an improper or unitialized structure of type **PV**.

Source

pv.src

pvUnpack

Purpose

Unpacks matrices stored in a structure of type **PV**.

Format

```
x = pvUnpack(p1,  m);
```

Input

p1	an instance of structure of type **PV**.
m	string, name of matrix, or integer, index of matrix.

Output

x	MxN general matrix or MxM symmetric matrix or N-dimensional array.

Source

`pv.src`

QNewton

Purpose

Optimizes a function using the BFGS descent algorithm.

Format

```
{ x, f, g, ret } = QNewton(&fct, start);
```

Input

&fct	pointer to a procedure that computes the function to be minimized. This procedure must have one input argument, a vector of parameter values, and one output argument, the value of the function evaluated at the input vector of parameter values.
start	Kx1 vector, start values.

Global Input

_qn_RelGradTol	scalar, convergence tolerance for relative gradient of

	estimated coefficients. Default = 1e-5.
_qn_GradProc	scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. This procedure must have a single input argument, a Kx1 vector of parameter values, and a single output argument, a Kx1 vector of gradients of the function with respect to the parameters evaluated at the vector of parameter values. If *_qn_GradProc* is 0, **QNewton** uses **gradp**.
_qn_MaxIters	scalar, maximum number of iterations. Default = 1e+5. Termination can be forced by pressing `C` on the keyboard.
_qn_PrintIters	scalar, if 1, print iteration information. Default = 0. Can be toggled during iterations by pressing `P` on the keyboard.
_qn_ParNames	Kx1 vector, labels for parameters.
_qn_PrintResults	scalar, if 1, results are printed.

Output

x	Kx1 vector, coefficients at the minimum of the function.	
f	scalar, value of function at minimum.	
g	Kx1 vector, gradient at the minimum of the function.	
ret	scalar, return code.	
	0	normal convergence
	1	forced termination

2	max iterations exceeded
3	function calculation failed
4	gradient calculation failed
5	step length calculation failed
6	function cannot be evaluated at initial parameter values

Remarks

If you are running in terminal mode, **GAUSS** will not see any input until you press `ENTER`. Pressing `C` on the keyboard will terminate iterations, and pressing `P` will toggle iteration output.

To reset global variables for this function to their default values, call **`QNewtonSet`**.

Example

This example computes maximum likelihood coefficients and standard errors for a Tobit model:

```
/***qnewton.e - a Tobit model***/
//Get data
z = loadd("tobit");
b0 = { 1, 1, 1, 1 };
{b,f,g,retcode} = qnewton(&lpr,b0);

//Covariance matrix of parameters
h = hessp(&lpr,b);
output file = qnewton.out reset;

print "Tobit Model";
print;
```

```
    print "coefficients standard errors";
    print b~sqrt(diag(invpd(h)));

    output off;

    //Log-likelihood proc
    proc lpr(b);
       local s,m,u;
       s = b[4];
       if s <= 1e-4;
          retp(error(0));
       endif;
       m = z[.,2:4]*b[1:3,.];
       u = z[.,1] ./= 0;
       retp(-sumc(u.*lnpdfn2(z[.,1]-m,s) + (1-u).*(ln(cdfnc(
    m/sqrt(s))))));endp;
```

produces:

```
    Tobit Model
    coefficients standard errors

      0.010417884  0.080220019
     -0.20805753   0.094551107
     -0.099749592  0.080006676
      0.65223067   0.099827309
```

Source

qnewton.src

QNewtonmt

Purpose

Minimize an arbitrary function.

Include

```
qnewtonmt.sdf
```

Format

out = **QNewtonmt**(*&fct*, *par*, *data*, *c*);

Input

&fct	pointer to a procedure that computes the function to be minimized. This procedure must have two input arguments, an instance of a **PV** structure containing the parameters, and a **DS** structure containing data, if any. And, one output argument, the value of the function evaluated at the input vector of parameter values.
par	an instance of a **PV** structure. The *par* instance is passed to the user-provided procedure pointed to by **&fct**. *par* is constructed using the **pvPack** functions.
data	an array of instances of a **DS** structure. This array is passed to the user-provided pointed by **&fct** to be used in the objective function. **QNewtonmt** does not look at this structure. Each instance contains the the following members which can be set in whatever way that is convenient for

computing the objective function:

data [i].dataMatrix	NxK matrix, data matrix.
data [i].dataArray	NxKxL [*arg1, arg2...argN*] array, data array.
data[i].vnames	string array, variable names (optional).
data[i].dsname	string, data name (optional).
data[i].type	scalar, type of data (optional).

c an instance of a **QNewtonmtControl** structure. Normally an instance is initialized by calling **QNewtonmtControlCreate** and members of this instance can be set to other values by the user. For an instance named *c*, the members are:

c.CovType	scalar, if 1, ML covariance matrix, else if 2, QML covariance matrix is computed. Default is 0, no covariance matrix.
c.GradProc	scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. Default = ., i.e., no gradient procedure has been provided.
c.MaxIters	scalar, maximum number of iterations. Default = 1e+5.
c.MaxTries	scalar, maximum number of attemps in random search. Default = 100.
c.relGradTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When

	this criterion has been satisifed **`QNewtonmt`** exits the iterations.
`c.randRadius`	scalar, If zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = .001.
`c.output`	scalar, if nonzero, results are printed. Default = 0.
`c.PrintIters`	scalar, if nonzero, prints iteration information. Default = 0.
`c.disableKey`	scalar, if nonzero, keyboard input disabled

Output

`out` an instance of an **QNewtonmtOut** structure. For an instance named out, the members are:

`out.par`	instance of a **PV** structure containing the parameter estimates will be placed in the member matrix *`out.par`*.
`out.fct`	scalar, function evaluated at *`x`*.
`out.retcode`	scalar, return code: 0 normal convergence. 1 forced exit. 2 maximum number of iterations exceeded.

	3	function calculation failed.
	4	gradient calculation failed.
	5	Hessian calculation failed.
	6	line search failed.
	7	error with constraints.
	8	function complex.
out.moment	KxK matrix, covariance matrix of parameters, if *c.covType*> 0.	
out.hessian	KxK matrix, matrix of second derivatives of objective function with respect to parameters.	

Remarks

There is one required user-provided procedure, the one computing the objective function to be minimized, and another optional functions, the gradient of the objective function.

These functions have one input argument that is an instance of type struct **PV** and a second argument that is an instance of type struct **DS**. On input to the call to **`QNewtonmt`**, the first argument contains starting values for the parameters and the second argument any required data. The data are passed in a separate argument because the structure in the first argument will be copied as it is passed through procedure calls which would be very costly if it contained large data matrices. Since **`QNewtonmt`** makes no changes to the second argument it will be passed by pointer thus saving time because its contents aren't copied.

The **PV** structures are set up using the **PV** pack procedures, **pvPack**, **pvPackm**, **pvPacks**, and **pvPacksm**. These procedures allow for setting up a parameter vector in a variety of ways.

For example, we might have the following objective function for fitting a nonlinear curve to data:

```
proc Micherlitz(struct PV par1, struct DS data1);
   local p0,e,s2,x,y;
   p0 = pvUnpack(par1, "parameters");
   y = data1.dataMatrix[.,1];
   x = data1.dataMatrix[.,2];
   e = y - p0[1] - p0[2]*exp(-p0[3] * x);
   retp(-lnpdfmvn(e,e'e/rows(e));
endp;
```

In this example the dependent and independent variables are passed to the procedure as the first and second columns of a data matrix stored in a single **DS** structure. Alternatively these two columns of data can be entered into a vector of **DS** structures one for each column of data:

If the objective function is the negative of a proper log-likelihood, and if *c.covType* is set to 1, the covariance matrix of the parameters is computed and returned in *out.moment*, and standard errors, t-statistics and probabilities are printed if *c.output* = 1.

If the objective function returns the negative of a vector of log-likelihoods, and if *c.covType* is set to 2, the quasi-maximum likelihood (QML) covariance matrix of the parameters is computed.

Example

The following is a complete example for estimating the parameters of the Micherlitz equation in data on the parameters and where an optional gradient procedure has been provided.

```
#include QNewtonmt.sdf

struct DS d0;
d0 = dsCreate;

y = 3.183|
    3.059|
    2.871|
    2.622|
    2.541|
    2.184|
    2.110|
    2.075|
    2.018|
    1.903|
    1.770|
    1.762|
    1.550;

x = seqa(1,1,13);
d0.dataMatrix = y~x;

struct QNewtonmtControl c0;
c0 = QNewtonmtControlCreate;
c0.output = 1;  //Print results
c0.covType = 1; //Compute moment matrix of parameters

struct PV par1;
par1 = pvCreate;
par1 = pvPack(par1,1|1|0, "parameters");

struct QNewtonmt out1;
out1 = QNewtonmt(&Micherlitz,par1,d0,c0);
```

Source

`qnewtonmt.src`

See Also

QNewtonmtControlCreate, QNewtonmtOutCreate

QNewtonmtControlCreate

Purpose

Creates default **QNewtonmtControl** structure.

Include

`qnewtonmt.sdf`

Format

c = **`QNewtonmtControlCreate`**();

Output

c	instance of **QNewtonmtControl** structure with members set to default values.

Source

`qnewtonmt.src`

See Also

QNewtonmt

QNewtonmtOutCreate

Purpose

Creates default **QNewtonmtOut** structure.

Format

c = **`QNewtonmtOutCreate`**();

Output

c	instance of **QNewtonmtOut** structure with members set to default values.

Source

`qnewtonmt.src`

See Also

QNewtonmt

QNewtonSet

Purpose

Resets global variables used by **`QNewton`** to default values.

Format

`QNewtonSet`;

Source

`qnewton.src`

QProg

Purpose

Solves the quadratic programming problem.

Format

{ *x*, *u1*, *u2*, *u3*, *u4*, *u5* } = **`QProg`**(*start*, *q*, *r*, *a*, *b*, *c*, *d*, *bnds*);

Input

start	Kx1 vector, start values.
q	KxK matrix, symmetric model matrix.
r	Kx1 vector, model constant vector.

a	MxK matrix, equality constraint coefficient matrix, or scalar 0, no equality constraints.
b	Mx1 vector, equality constraint constant vector, or scalar 0, will be expanded to Mx1 vector of zeros.
c	NxK matrix, inequality constraint coefficient matrix, or scalar 0, no inequality constraints.
d	Nx1 vector, inequality constraint constant vector, or scalar 0, will be expanded to Nx1 vector of zeros.
bnds	Kx2 matrix, bounds on *x*, the first column contains the lower bounds on *x*, and the second column the upper bounds. If scalar 0, the bounds for all elements will default to ±1e200.

Global Input

_qprog_maxit	scalar, maximum number of iterations. Default = 1000.

Output

x	Kx1 vector, coefficients at the minimum of the function.
u1	Mx1 vector, Lagrangian coefficients of equality constraints.
u2	Nx1 vector, Lagrangian coefficients of inequality constraints.
u3	Kx1 vector, Lagrangian coefficients of lower bounds.
u4	Kx1 vector, Lagrangian coefficients of upper bounds.
ret	scalar, return code.

0	successful termination
1	max iterations exceeded
2	machine accuracy is insufficient to maintain decreasing function values
3	model matrices not conformable
< *0*	active constraints inconsistent

Remarks

`QProg` solves the standard quadratic programming problem:

$$\min \tfrac{1}{2}x'Qx - x'R$$

subject to constraints,

$$Ax = BCx \leq D$$

and bounds,

$$x_{low} \leq x \leq x_{up}$$

Source

`qprog.src`

QProgmt

Purpose

Solves the quadratic programming problem.

Include

qprogmt.sdf

Format

```
qOut = QProgmt(qIn );
```

Input

qIn	instance of a **qprogMTIn** structure containing the following members:	
	qIn.start	Kx1 vector, start values.
	qIn.q	KxK matrix, symmetric model matrix.
	qIn.r	Kx1 vector, model constant vector.
	qIn.a	MxK matrix, equality constraint coefficient matrix, or scalar 0, no equality constraints.
	qIn.b	Mx1 vector, equality constraint constant vector, or scalar 0, will be expanded to Mx1 vector of zeros.
	qIn.c	NxK matrix, inequality constraint coefficient matrix, or scalar 0, no inequality constraints.
	qIn.d	Nx1 vector, inequality constraint constant vector, or scalar 0, will be expanded to Nx1 vector of zeros.

	qIn.bounds	Kx2 matrix, bounds on *qOut.x*, the first column contains the lower bounds on *qOut.x*, and the second column the upper bounds. If scalar 0, the bounds for all elements will default to ±1e200.
	qIn.maxit	scalar, maximum number of iterations. Default = 1000.

Output

qOut	instance of a **qprogMTOut** structure containing the following members:	
	qOut.x	Kx1 vector, coefficients at the minimum of the function.
	qOut.lagrange	instance of a **qprogMTLagrange** structure containing the following members:
	qOut.lagrange.lineq	Mx1 vector, Lagrangian coefficients of equality constraints.
	qOut.lagrange.linineq	Nx1 vector, Lagrangian coefficients of inequality constraints.
	qOut.lagrange.bounds	Kx2 matrix, Lagrangian coefficients of bounds, the first column contains the lower bounds and the second the upper bounds.
	qOut.ret	scalar, return code.

	0 successful termination 1 max iterations exceeded 2 machine accuracy is insufficient to maintain decreasing function values 3 model matrices not conformable < 0 active constraints inconsistent

Remarks

QProgmt solves the standard quadratic programming problem:

$$min \frac{1}{2} x'Qx - x'R$$

subject to constraints,

$$Ax = BCx \le D$$

and bounds,

$$x_{low} \le x \le x_{up}$$

Source

`qprogmt.src`

See Also

QProgmtInCreate

QProgmtInCreate

Purpose

Creates an instance of a structure of type **`QProgmtInCreate`** with the *`maxit`* member set to a default value.

Include

`qprogmt.sdf`

Format

s = **`QProgmtInCreate`**();

Output

s	instance of structure of type **QProgmtIn**.

Source

`qprogmt.src`

See Also

QProgmt

qqr

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix *x*, such that: $X = Q_1 R$

Format

{ *q1*, *r* } = **qqr**(*x*);

Input

x	NxP matrix.

Output

q1	NxK unitary matrix, K = **min**(N,P).
r	KxP upper triangular matrix.

Remarks

Given *X*, there is an orthogonal matrix *Q* such that *Q'x* is zero below its diagonal, i.e.,

$$Q'X = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where *R* is upper triangular. If we partition

$$Q = \left[Q_1 Q_2 \right]$$

where Q_1 has P columns, then

$$X = Q_1 R$$

is the QR decomposition of X. If X has linearly independent columns, R is also the Cholesky factorization of the moment matrix of X, i.e., of $X'X$.

If you want only the R matrix, see the function **qr**. Not computing Q_1 can produce significant improvements in computing time and memory usage.

An unpivoted R matrix can also be generated using **cholup**:

```
r = cholup(zeros(cols(x), cols(x)), x);
```

For linear equation or least squares problems, which require Q_2 for computing residuals and residual sums of squares, see **olsqr** and **qtyr**.

For most problems an explicit copy of Q_1 or Q_2 is not required. Instead one of the following, $Q'Y$, QY, $Q_1'Y$, Q_1Y, $Q_2'Y$, or Q_2Y, for some Y, is required. These cases are all handled by **qtyr** and **qyr**. These functions are available because Q and Q_1 are typically very large matrices while their products with Y are more manageable.

If N < P, the factorization assumes the form:

$$Q'X = [R_1 \quad R_2]$$

where R_1 is a PxP upper triangular matrix and R_2 is Px(N-P). Thus Q is a PxP matrix and R is a PxN matrix containing R_1 and R_2. This type of factorization is useful for the solution of underdetermined systems. However, unless the linearly independent columns happen to be the initial rows, such an analysis also requires pivoting (see **qre** and **qrep**).

Source

`qqr.src`

See Also

qre, qrep, qtyr, qtyre, qtyrep, qyr, qyre, qyrep, olsqr

qqre

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix *x*, such that: $X[.,E] = Q_1R$

Format

{ *q1*, *r*, *e* } = **qqre**(*x*);

Input

x	NxP matrix.

Output

q1	NxK unitary matrix, K = **min**(N,P).
r	KxP upper triangular matrix.
e	Px1 permutation vector.

Remarks

Given $X[.,E]$, where E is a permutation vector that permutes the columns of X, there is an orthogonal matrix Q such that $Q'X[.,E]$ is zero below its diagonal, i.e.,

$$Q'R[.,E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X\begin{bmatrix} ., E \end{bmatrix} = Q_1 R$$

is the QR decomposition of $X[.,E]$.

If you want only the R matrix, see **`qre`**. Not computing Q_1 can produce significant improvements in computing time and memory usage.

If X has rank P, then the columns of X will not be permuted. If X has rank M < P, then the M linearly independent columns are permuted to the front of X by E. Partition the permuted X in the following way:

$$X[., E] = [X_1 X_2]$$

where X is NxM and X_2 is Nx(P-M). Further partition R in the following way:

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

where R_{11} is MxM and R_{12} is Mx(P-M). Then

$$A = R^{-1}{}_{11}R_{12}$$

and

$$X_2 = X_1A$$

that is, A is an Mx(P-N) matrix defining the linear combinations of X_2 with respect to X_1.

If N < P, the factorization assumes the form:

$$Q'X = [R_1R_2]$$

where R_1 is a PxP upper triangular matrix and R_2 is Px(N-P). Thus Q is a PxP matrix and R is a PxN matrix containing R_1 and R_2. This type of factorization is useful for the solution of underdetermined systems. For the solution of

$$X[., E]b = Y$$

it can be shown that

```
b = qrsol(Q'Y, R1)|zeros(N-P,1);
```

The explicit formation here of Q, which can be a very large matrix, can be avoided by using the function **qtyre**.

For further discussion of QR factorizations see the remarks under **qqr**.

Source

qqr.src

See Also

qqr**qqr**, qtyre, olsqr

qqrep

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix *x*, such that: $X[.,E] = Q_1R$

Format

```
{ q1, r, e } = qqrep(x, pvt);
```

Input

x	NxP matrix.
pvt	Px1 vector, controls the selection of the pivot columns: if *pvt*[*i*] > 0, *x*[*i*] is an initial column if *pvt*[*i*] = 0, *x*[*i*] is a free column if *pvt*[*i*] < 0, *x*[*i*] is a final column The initial columns are placed at the beginning of the matrix and the final columns are placed at the end. Only the free columns will be moved during the decomposition.

Output

q1	NxK unitary matrix, K = **min**(N,P).
r	KxP upper triangular matrix.
e	Px1 permutation vector.

Remarks

Given $X[.,E]$, where E is a permutation vector that permutes the columns of X, there is an orthogonal matrix Q such that $Q'X[.,E]$ is zero below its diagonal, i.e.,

$$Q'R[.,E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X\begin{bmatrix} ., E \end{bmatrix} = Q_1 R$$

is the QR decomposition of $X[.,E]$.

qqrep allows you to control the pivoting. For example, suppose that `x` is a data set with a column of ones in the first column. If there are linear dependencies among the columns of `x`, the column of ones for the constant may get pivoted away. This column can be forced to be included among the linearly independent columns using *`pvt`*.

If you want only the `R` matrix, see **qrep**. Not computing Q_1 can produce significant improvements in computing time and memory usage.

Source

```
qqr.src
```

See Also

qqr, qre, olsqr

qr

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix `x`, such that: $X = Q_1 R$

Format

r = **qr**(*x*);

Input

x	NxP matrix.

Output

r	KxP upper triangular matrix, K = **min**(N,P).

Remarks

qr is the same as **qqr** but doesn't return the Q_1 matrix. If Q_1 is not wanted, **qr** will save a significant amount of time and memory usage, especially for large problems.

Given X, there is an orthogonal matrix Q such that $Q'X$ is zero below its diagonal, i.e.,

$$Q'X = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \left[Q_1 Q_2 \right]$$

where Q_1 has P columns, then

$$X = Q_1 R$$

is the QR decomposition of X. If X has linearly independent columns, R is also the Cholesky factorization of the moment matrix of X, i.e., of $X'X$.

qr does not return the Q_1 matrix because in most cases it is not required and can be very large. If you need the Q_1 matrix, see the function **qqr**. If you need the entire Q matrix, call **qyr** with Y set to a conformable identity matrix.

For most problems $Q'Y$, $Q_1'Y$, or QY, Q_1Y, for some Y, are required. For these cases see **qtyr** and **qyr**.

For linear equation or least squares problems, which require Q_2 for computing residuals and residual sums of squares, see **olsqr**.

If N<P, the factorization assumes the form:

$$Q'X = [R_1 R_2]$$

where R_1 is a PxP upper triangular matrix and R_2 is Px(N-P). Thus Q is a PxP matrix and R is a PxN matrix containing R_1 and R_2. This type of factorization is useful for the solution of underdetermined systems. However, unless the linearly independent columns happen to be the initial rows, such an analysis also requires pivoting (see **qre** and **qrep**).

Source

qr.src

See Also

qqr, qrep, qtyre

qre

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix *x*, such that: $X[.,E] = Q_1R$

Format

{ *r*, *e* } = **qre**(*x*);

Input

x	NxP matrix.

Output

r	KxP upper triangular matrix, K = **min**(N,P).
e	Px1 permutation vector.

Remarks

qre is the same as **qqre** but doesn't return the Q_1 matrix. If Q_1 is not wanted, **qre** will save a significant amount of time and memory usage, especially for large problems.

Given $X[.,E]$, where E is a permutation vector that permutes the columns of *x*, there is an orthogonal matrix Q such that $Q'X[.,E]$ is zero below its diagonal, i.e.,

$$Q'X[.,E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X\begin{bmatrix} .,E \end{bmatrix} = Q_1 R$$

is the QR decomposition of $X[.,E]$.

`qre` does not return the Q_1 matrix because in most cases it is not required and can be very large. If you need the Q_1 matrix, see the function **`qqre`**. If you need the entire Q matrix, call **`qyre`** with Y set to a conformable identity matrix. For most problems $Q'Y$, $Q_1'Y$, or QY, Q_1Y, for some y, are required. For these cases see **`qtyre`** and **`qyre`**.

If X has rank P, then the columns of X will not be permuted. If X has rank M<P, then the M linearly independent columns are permuted to the front of X by E. Partition the permuted X in the following way:

$$X[.,E] = [\ X_1\ X_2\ ]$$

where X_1 is NxM and X_2 is Nx(P-M). Further partition R in the following way:

where R_{11} is MxM and R_{12} is Mx(P-M). Then

$$A = R_{11}^{-1} R_{12}$$

and

$$X_2 = X_1 A$$

that is, A is an Mx(P-N) matrix defining the linear combinations of X_2 with respect to X_1

If N<P the factorization assumes the form:

$$Q'X = [R_1 R_2]$$

where R_1 is a PxP upper triangular matrix and R_2 is Px(N-P). Thus Q is a PxP matrix and R is a PxN matrix containing R_1 and R_2. This type of factorization is useful for the solution of underdetermined systems. For the solution of

$$X[.,E]b = Y$$

it can be shown that

```
b = qrsol(Q'Y, R1)|zeros(N-P,1);
```

The explicit formation here of Q, which can be a very large matrix, can be avoided by using the function **qtyre**.

For further discussion of QR factorizations see the remarks under **qqr**.

Source

qr.src

See Also

qqr, olsqr

qrep

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix *X*, such that:

```
    X[.,E] = Q1R
```

Format

{ *r*, *e* } = **qrep**(*X*, *pvt*);

Input

X	NxP matrix.
pvt	Px1 vector, controls the selection of the pivot columns: if *pvt*[*i*] > 0, *X*[*i*] is an initial column. if *pvt*[*i*] = 0, *X*[*i*] is a free column. if *pvt*[*i*] < 0, *X*[*i*] is a final column. The initial columns are placed at the beginning of the matrix and the final columns are placed at the end. Only the free columns will be moved during the decomposition.

Output

r	KxP upper triangular matrix, K = **min**(N,P).

e	Px1 permutation vector.

Remarks

qrep is the same as **qqrep** but doesn't return the Q_1 matrix. If Q_1 is not wanted, **qrep** will save a significant amount of time and memory usage, especially for large problems.

Given *X[.,E]*, where *E* is a permutation vector that permutes the columns of *X*, there is an orthogonal matrix *Q* such that *Q'X[.,E]* is zero below its diagonal, i.e.,

$$Q'X[.,E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where *R* is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X\left[.,E\right] = Q_1 R$$

is the QR decomposition of *X[.,E]*.

qrep does not return the Q_1 matrix because in most cases it is not required and can be very large. If you need the Q_1 matrix, see the function **qqrep**. If you need the entire *Q* matrix, call **qyrep** with *Y* set to a conformable identity matrix. For most problems *Q'Y*, Q_1*'Y*, or *QY*, Q_1*Y*, for some *Y*, are required. For these cases see **qtyrep** and **qyrep**.

qrep allows you to control the pivoting. For example, suppose that *X* is a data set with a column of ones in the first column. If there are linear dependencies among the columns of X, the column of ones for the constant may get pivoted away. This column can be forced to be included among the linearly independent columns using `pvt`.

Source

`qr.src`

See Also

qr, qre, qqrep

qrsol

Purpose

Computes the solution of $Rx = b$ where R is an upper triangular matrix.

Format

x = **qrsol**(*b*, *R*);

Input

b	PxL matrix.
R	PxP upper triangular matrix.

Output

x	PxL matrix.

Remarks

qrsol applies a backsolve to $Rx = b$ to solve for x. Generally R will be the R matrix from a QR factorization. **qrsol** may be used, however, in any situation where R is upper triangular.

Source

qrsol.src

See Also

qqr, qr, qtyr, qrtsol

qrtsol

Purpose

Computes the solution of $R'x = b$ where R is an upper triangular matrix.

Format

x = **qrtsol**(b, R);

Input

b	PxL matrix.
R	PxP upper triangular matrix.

Output

x	PxL matrix.

Remarks

qrtsol applies a forward solve to $R'x = b$ to solve for x. Generally R will be the R matrix from a QR factorization. **qrtsol** may be used, however, in any situation where R is upper triangular. If R is lower triangular, transpose before calling **qrtsol**.

If R is not transposed, use **qrsol**.

Source

qrsol.src

See Also

qqr, **qr**, **qtyr**, **qrsol**

qtyr

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix X and returns $Q'Y$ and R.

Format

{ *qty*, *r* } = **qtyr**(*y*, *X*);

Input

y	NxL matrix.
X	NxP matrix.

Output

qty	NxL unitary matrix.
r	KxP upper triangular matrix, K = **min**(N,P).

Remarks

Given *X*, there is an orthogonal matrix *Q* such that *Q'X* is zero below its diagonal, i.e.,

$$Q'X = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X = Q_1 R$$

is the QR decomposition of *X*. If *X* has linearly independent columns, *R* is also the Cholesky factorization of the moment matrix of *X*, i.e., of *X'X*. For most problems *Q* or Q_1 is not what is required. Rather, we require *Q'Y* or $Q_1'Y$ where *Y* is an NxL matrix (if either *QY* or Q_1Y are required, see **qyr**). Since *Q* can be a very large matrix,

qtyr has been provided for the calculation of $Q'Y$ which will be a much smaller matrix. $Q_1'Y$ will be a submatrix of $Q'Y$. In particular,

$$G = Q_1'Y = \texttt{qty}[1:P,.]$$

and $Q_2'Y$ is the remaining submatrix:

$$H = Q_2'Y = \texttt{qty}[P+1:N,.]$$

Suppose that X is an NxK data set of independent variables, and Y is an Nx1 vector of dependent variables. Then it can be shown that

$$b = R^{-1}G$$

and

$$s_j = \sum_{i=1}^{N-P} H_{i,j} j = 1, 2, ...L$$

where b is a PxL matrix of least squares coefficients and s is a 1xL vector of residual sums of squares. Rather than invert R directly, however, it is better to apply **qrsol** to

$$Rb = Q_1'Y$$

For rank deficient least squares problems, see **qtyre** and **qtyrep**.

Example

The QR algorithm is the numerically superior method for the solution of least squares problems:

```
    loadm x, y;
    { qty, r } = qtyr(y,x);
    q1ty = qty[1:rows(r),.];
    q2ty = qty[rows(r)+1:rows(qty),.];

    //LS coefficients
    b = qrsol(q1ty,r);

    //Residual sums of squares
    s2 = sumc(q2ty^2);
```

Source

qtyr.src

See Also

qqr, qtyre, qtyrep, olsqr

qtyre

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix X and returns $Q'Y$ and R.

Format

{ *qty*, *r*, *e* } = **qtyre**(*y*, *x*);

Input

`y`	NxL matrix.
`x`	NxP matrix.

Output

`qty`	NxL unitary matrix.
`r`	KxP upper triangular matrix, K = **`min`**(N,P).
`e`	Px1 permutation vector.

Remarks

Given *X[.,E]*, where *E* is a permutation vector that permutes the columns of *X*, there is an orthogonal matrix *Q* such that *Q'X[.,E]* is zero below its diagonal, i.e.,

$$Q'X[.,E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X\begin{bmatrix} ., E \end{bmatrix} = Q_1 R$$

is the QR decomposition of *X[.,E]*.

If X has rank P, then the columns of X will not be permuted. If X has rank M<P, then the M linearly independent columns are permuted to the front of X by E. Partition the permuted X in the following way:

$$X[.,E] = [X_1 X_2]$$

where X_1 is NxM and X_2 is Nx(P-M). Further partition R in the following way:

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

where R_{11} is MxM and R_{12} is Mx(P-M). Then

$$A = R^{-1}{}_{11} R_{12}$$

and

$$X_2 = X_1 A$$

that is, A is an Mx(P-N) matrix defining the linear combinations of X_2 with respect to X_1.

For most problems Q or Q_1 is not it is required. Rather, we require $Q'Y$ or $Q_1'Y$ where Y is an NxL matrix. Since Q can be a very large matrix, **`qtyre`** has been provided for the calculation of $Q'Y$ which will be a much smaller matrix. $Q_1'Y$ will be a submatrix of $Q'Y$. In particular,

$$Q_1'Y = \texttt{qty}[1:P,.]$$

and $Q_2'Y$ is the remaining submatrix:

$$Q_2'Y = \texttt{qty}[P+1:N,.]$$

Suppose that X is an NxK data set of independent variables and Y is an Nx1 vector of dependent variables. Suppose further that X contains linearly dependent columns, i.e., X has rank M < P. Then define

$$C = Q_1' Y[1:M, .]$$
$$A = R[1:M, 1:M]$$

and the vector (or matrix of L > 1) of least squares coefficients of the reduced, linearly independent problem is the solution of

$$Ab = C$$

To solve for b use **qrsol**:

```
b = qrsol(C, A);
```

If N < P, the factorization assumes the form:

$$Q' X[., E] = [R_1\ R_2]$$

where R_1 is a PxP upper triangular matrix and R_2 is Px(N-P). Thus Q is a PxP matrix and R is a PxN matrix containing R_1 and R_2. This type of factorization is useful for the solution of underdetermined systems. For the solution of

$$X[., E]b = Y$$

it can be shown that

```
b = qrsol(Q'Y, R1)|zeros(N-P,1);
```

Source

qtyr.src

See Also

qqr, qre, qtyr

qtyrep

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix X using a pivot vector and returns $Q'Y$ and R.

Format

```
{ qty, r,  e } = qtyrep(y, x,  pvt);
```

Input

y	NxL matrix.
x	NxP matrix.
pvt	Px1 vector, controls the selection of the pivot columns: if *pvt*[*i*] > 0, *x*[*i*] is an initial column. if *pvt*[*i*] = 0, *x*[*i*] is a free column. if *pvt*[*i*] < 0, *x*[*i*] is a final column. The initial columns are placed at the beginning of the matrix and the final columns are placed at the end. Only the free columns will be moved during the decomposition.

Output

`qty`	NxL unitary matrix.
`r`	KxP upper triangular matrix, K = **`min`**(N,P).
`e`	Px1 permutation vector.

Remarks

Given *X[.,E]*, where *E* is a permutation vector that permutes the columns of *X*, there is an orthogonal matrix *Q* such that *Q'X[.,E]* is zero below its diagonal, i.e.,

$$Q'X[.,E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where *R* is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X\left[.,E\right] = Q_1 R$$

is the QR decomposition of *X[.,E]*.

`qtyrep` allows you to control the pivoting. For example, suppose that *X* is a data set with a column of ones in the first column. If there are linear dependencies among the columns of *X*, the column of ones for the constant may get pivoted away. This column can be forced to be included among the linearly independent columns using `pvt`.

Source

```
qtyr.src
```

See Also

qrep, qtyre

quantile

Purpose

Computes quantiles from data in a matrix, given specified probabilities.

Format

```
y = quantile(x,   e)
```

Input

x	NxK matrix of data.
e	Lx1 vector, quantile levels or probabilities.

Output

y	LxK matrix, quantiles.

Remarks

`quantile` will not succeed if N***`minc`**(e) is less than 1, or N***`maxc`**(e) is greater than N - 1. In other words, to produce a **`quantile`** for a level of .001, the input matrix must have more than 1000 rows.

Example

```
//Set the rng seed for repeatable random numbers
rndseed 345567;

//Create a 1000x4 random normal matrix
x = rndn(1000,4);

//Quantile levels
e = { .025, .5, .975 };
y = quantile(x,e);

print "medians";
print y[2,.];
print;
print "95 percentiles";
print y[1,.];
print y[3,.];
```

Produces the following output:

```
medians
    -0.037801917   0.029923972  -0.010477829  -0.023937160

95 percentiles
      -2.0074122    -2.0798579    -1.9982702    -1.9605009
       2.0437573     2.0271770     1.9025695     1.9228044
```

Source

`quantile.src`

quantiled

Purpose

Computes quantiles from data in a data set, given specified probabilities.

Format

```
y = quantiled(dataset, e, var);
```

Input

dataset	string, data set name, or NxM matrix of data.
e	Lx1 vector, quantile levels or probabilities.
var	Kx1 vector or scalar zero. If Kx1, character vector of labels selected for analysis, or numeric vector of column numbers in data set of variables selected for analysis. If scalar zero, all columns are selected. If *dataset* is a matrix *var* cannot be a character vector.

Output

y	LxK matrix, quantiles.

Remarks

quantiled will not succeed if N***minc**(e) is less than 1, or N***maxc**(e) is greater than N - 1. In other words, to produce a **quantile** for a level of .001, the input matrix must have more than 1000 rows.

Example:

```
    y = quantiled("tobit",e,0);

     print "medians";
     print y[2,.];
     print;
     print "95 percentiles";
     print y[1,.];
     print y[3,.];
```

produces:

```
    medians

      0.0000 1.0000 -0.0021 -0.1228

    95 percentiles

     -1.1198 1.0000 -1.8139 -2.3143

      2.3066 1.0000  1.4590  1.6954
```

Source

`quantile.src`

qyr

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix X and returns QY and R.

Format

{ *qy*, *r* } = **qyr**(*y*, *x*);

Input

y	NxL matrix.
X	NxP matrix.

Output

qy	NxL unitary matrix.
r	KxP upper triangular matrix, K = **min**(N,P).

Remarks

Given X, there is an orthogonal matrix Q such that $Q'X$ is zero below its diagonal, i.e.,

$$Q'X = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X = Q_1 R$$

is the QR decomposition of *X*. If *X* has linearly independent columns, *R* is also the Cholesky factorization of the moment matrix of *X*, i.e., of *X'X*.

For most problems *Q* or Q_1 is not what is required. Since *Q* can be a very large matrix, **qyr** has been provided for the calculation of *QY*, where *Y* is some NxL matrix, which will be a much smaller matrix.

If either *Q'Y* or $Q_1'Y$ are required, see **qtyr**.

Source

qyr.src

See Also

qqr, qyre, qyrep, olsqr

qyre

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix *x* and returns *QY* and *R*.

Format

{ *qy*, *r*, *e* } = **qyre**(*y*, *x*);

Input

y	NxL matrix.
x	NxP matrix.

Output

qy	NxL unitary matrix.
r	KxP upper triangular matrix, K = **`min`**(N,P).
e	Px1 permutation vector.

Remarks

Given *X[.,E]*, where *E* is a permutation vector that permutes the columns of *X*, there is an orthogonal matrix *Q* such that *Q'X[.,E]* is zero below its diagonal, i.e.,

$$Q'X[.,E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where *R* is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X\begin{bmatrix} ., E \end{bmatrix} = Q_1 R$$

is the QR decomposition of *X[.,E]*.

For most problems *Q* or Q_1 is not what is required. Since *Q* can be a very large matrix, **`qyre`** has been provided for the calculation of *QY*, where *Y* is some NxL matrix, which will be a much smaller matrix.

If either *Q'Y* or $Q_1'Y$ are required, see **`qtyre`**.

If N <P, the factorization assumes the form:

$$Q'X[.,E] = [R_1 R_2]$$

where R_1 is a PxP upper triangular matrix and R_2 is Px(N-P). Thus Q is a PxP matrix and R is a PxN matrix containing R_1 and R_2.

Source

qyr.src

See Also

qqr, qre, qyr

qyrep

Purpose

Computes the orthogonal-triangular (QR) decomposition of a matrix X using a pivot vector and returns QY and R.

Format

```
{ qy, r, e } = qyrep(y, x,  pvt);
```

Input

y	NxL matrix.
x	NxP matrix.
pvt	Px1 vector, controls the selection of the pivot columns:
	if $pvt[i] > 0$, $x[i]$ is an initial column.

if $pvt[i] = 0$, $x[i]$ is a free column.

if $pvt[i] < 0$, $x[i]$ is a final column.

The initial columns are placed at the beginning of the matrix and the final columns are placed at the end. Only the free columns will be moved during the decomposition.

Output

qy	NxL unitary matrix.
r	KxP upper triangular matrix, K = **min**(N,P).
e	Px1 permutation vector.

Remarks

Given *X[.,E]*, where *E* is a permutation vector that permutes the columns of *X*, there is an orthogonal matrix *Q* such that *Q'X[.,E]* is zero below its diagonal, i.e.,

$$Q'X[.,E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X\begin{bmatrix} ., E \end{bmatrix} = Q_1 R$$

is the QR decomposition of *X[.,E]*.

qyrep allows you to control the pivoting. For example, suppose that *X* is a data set with a column of ones in the first column. If there are linear dependencies among the columns of *X*, the column of ones for the constant may get pivoted away. This column can be forced to be included among the linearly independent columns using `pvt`.

For most problems Q or Q_1 is not what is required. Since Q can be a very large matrix, **qyrep** has been provided for the calculation of QY, where Y is some NxL matrix, which will be a much smaller matrix.

If either $Q'Y$ or $Q_1'Y$ are required, see **qtyrep**.

If N<P, the factorization assumes the form:

$$Q'X[.,E] = [R_1 R_2]$$

where R_1 is a PxP upper triangular matrix and R_2 is Px(N-P). Thus Q is a PxP matrix and R is a PxN matrix containing R_1 and R_2.

Source

qyr.src

See Also

qr, qqrep, qrep, qtyrep

rank

Purpose

Computes the rank of a matrix, using the singular value decomposition.

Format

k = **rank**(*x*);

Input

x	NxP matrix.

Global Input

_svdtol	scalar, the tolerance used in determining if any of the singular values are effectively 0. The default value is $10e^{-13}$. This can be changed before calling the procedure.

Output

k	an estimate of the rank of *x*. This equals the

	number of singular values of *x* that exceed a prespecified tolerance in absolute value.

Global Output

_svderr	scalar, if not all of the singular values can be computed **_svderr** will be nonzero.

Source

svd.src

rankindx

Purpose

Returns the vector of ranks of a vector.

Format

y = **rankindx**(*x*, *flag*);

Input

x	Nx1 vector.
flag	scalar, 1 for numeric data or 0 for character data.

Output

y	Nx1 vector containing the ranks of *x*. That is, the rank of the largest element is N and the rank of the smallest is 1. (To get ranks in descending order, subtract *y* from N+1).

Remarks

`rankindx` assigns different ranks to elements that have equal values (ties). Missing values are assigned the lowest ranks.

Example

```
    x = { 12, 4, 15, 7, 8 };
    r = rankindx(x,1);
```

After the code above, *r* is equal to:

```
        4
        1
    r = 5
        2
        3
```

readr

Purpose

Reads a specified number of rows of data from a **GAUSS** data set (`.dat`) file or a **GAUSS** matrix (`.fmt`) file.

Format

```
y = readr(f1, r);
```

Input

f1	scalar, file handle of an open file.
r	scalar, number of rows to read.

Output

y	NxK matrix, the data read from the file.

Remarks

The first time a **readr** statement is encountered, the first *r* rows will be read. The next time it is encountered, the next *r* rows will be read in, and so on. If the end of the data set is reached before *r* rows can be read, then only those rows remaining will be read.

After the last row has been read, the pointer is placed immediately after the end of the file. An attempt to read the file in these circumstances will cause an error message.

To move the pointer to a specific place in the file use **seekr**.

Example

```
open dt = dat1.dat;
m = 0;

do until eof(dt);
```

```
        x = readr(dt,400);
        m = m + moment(x,0);
    endo;

    dt = close(dt);
```

This code reads data from a data set 400 rows at a time. The moment matrix for each set of rows is computed and added to the sum of the previous moment matrices. The result is the moment matrix for the entire data set. **eof**(*dt*) returns 1 when the end of the data set is encountered.

See Also

open, create, writer, seekr, eof

real

Purpose

Returns the real part of *x*.

Format

zr = **real**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

zr	NxK matrix or N-dimensional array, the real part of *x*.

Remarks

If *x* is not complex, *zr* will be equal to *x*.

Example

```
x = {  1 11+2i,
      7i  3,
      2+1i  1 };
zr = real(x);
```

After the code above, *x* and *zr* are equal to:

```
    1+0i  11+2i        1  11
x = 0+7i   3+0i   zr = 0   3
    2+1i   1+0i        2   1
```

See Also

complex, imag

recode

Purpose

Changes the values of an existing vector from a vector of new values. Used in data transformations.

Format

y = **recode**(*x*, *e*, *v*);

Input

x	Nx1 vector to be recoded (changed).
e	NxK matrix of 1's and 0's.
v	Kx1 vector containing the new values to be assigned to the recoded variable.

Output

y	Nx1 vector containing the recoded values of *x*.

Remarks

There should be no more than a single 1 in any row of *e*.

For any given row N of *x* and *e*, if the Kth column of *e* is 1, the Kth element of *v* will replace the original element of *x*.

If every column of *e* contains a 0, the original value of *x* will be unchanged.

Example

```
    x = { 20,
          45,
          32,
          63,
      29 };

//Create 4 column vectors with a 1 where the statement
//evaluates as 'true'
e1 = (20 .lt x) .and (x .le 30);
e2 = (30 .lt x) .and (x .le 40);
e3 = (40 .lt x) .and (x .le 50);
e4 = (50 .lt x) .and (x .le 60);

//Horizontally concatenate the column vectors into a 5x4
//matrix
e = e1~e2~e3~e4;

v = { 1.2,
      2.4,
      3.1,
      4.6 };

//Replace elements of 'x' with elements from 'v' based upon
//the 0's and 1's in 'e'
y = recode(x,e,v);
```

The above code assigns e and y as follows:

```
    0   0   0   0
    0   0   1   0
e = 0   1   0   0
    0   0   0   0
    1   0   0   0

//Since the third column of the second row of 'e' is equal
//to 1, the second row of 'y' is set equal to the third
//element of 'v', etc.
    20.000000
    3.1000000
    y = 2.4000000
        63.000000
        1.2000000
```

Source

```
datatran.src
```

See Also

code, substute

recode (dataloop)

Purpose

Changes the value of a variable with different values based on a set of logical expressions.

Format

```
recode  var with
  or
recode # var with
  or
recode $ var with
    val1 for expression_1,
    val2 for expression_2,
     .
     .
     .
```

```
        valn for expression_n;
```

Input

var	literal, the new variable name.
val	scalar, value to be used if corresponding expression is TRUE.
expression	logical scalar-returning expression that returns nonzero TRUE or zero FALSE.

Remarks

If '$' is specified, the variable will be considered a character variable. If '#' is specified, the variable will be considered numeric. If neither is specified, the type of the variable will be left unchanged.

The logical expressions must be mutually exclusive, that is only one may return TRUE for a given row (observation).

If none of the expressions is TRUE for a given row (observation), its value will remain unchanged.

Any variables referenced must already exist, either as elements of the source data set, as `extern`'s, or as the result of a previous `make`, `vector`, or `code` statement.

Example

```
    recode age with
           1 for age < 21,
           2 for age >= 21 and age < 35,
           3 for age >= 35 and age < 50,
        4 for age >= 50 and age < 65,
        5 for age >= 65;
```

```
    recode $ sex with
        "MALE" for sex =\,= 1,
        "FEMALE" for sex =\,= 0;
```

```
    recode # sex with
        1 for sex $=\,= "MALE",
        0 for sex $=\,= "FEMALE";
```

See Also

code (dataloop)

recserar

Purpose

Computes a vector of autoregressive recursive series.

Format

y = **recserar**(*x*, *y0*, *rho*);

Input

x	NxK matrix
y0	PxK matrix.
rho	PxK matrix.

Output

y	NxK matrix containing the series.

Remarks

recserar is particularly useful in dealing with time series.

Typically, the result would be thought of as K vectors of length N.

y0 contains the first P values of each of these vectors (thus, these are prespecified). The remaining elements are constructed by computing a Pth order "autoregressive" recursion, with weights given by *rho*, and then by adding the result to the corresponding elements of *x*. That is, the *t*th row of *y* is given by:

```
    y[t,.] = x[t,.] + rho[1,.] * y[t-1,.] +...+ rho[P,.] * y[t-
    p,.], t = P + 1,...N
```

and

```
    y[t,.] = y0[t,.], t = 1,...,P
```

Note that the first P rows of *x* are not used.

Example

Example 1

```
    y0 = 0;
    rho = 0.4;
    err = rndn(20, 1);

    y = recserar(err, y0, rho);
```

In this basic example, one autoregressive series is formed. The general form of the series can be written:

```
    y[t] = rho*y[t-1] + err[t]
```

Example 2

```
    n = 10;
    fn multnorm(n,sigma) = rndn(n, rows(sigma))*chol(sigma);

    sig = {    1 -0.3,
            -0.3    1 };
    rho = { 0.5 0.3 };
    y0 = { 0 0 };
    e = multnorm(n,sig);
    x = ones(n,1)~rndn(n,3);
    b = { 1, 2, 3, 4 };

    y = recserar(x*b+e,y0,rho);
```

In this example, two autoregressive series are formed using simulated data. The general form of the series can be written:

```
     y[t,1] = rho[1,1]*y[t-1,1] + x[t,.]*b + e[t,1]
     y[t,2] = rho[2,1]*y[t-1,2] + x[t,.]*b + e[t,2]
```

The error terms (*e[t,1]* and *e[t,2]*) are not individually serially correlated, but they are contemporaneously correlated with each other. The variance-covariance matrix is *sig*.

See Also

recsercp, recserrc

recsercp

Purpose

Computes a recursive series involving products. Can be used to compute cumulative products, to evaluate polynomials using Horner's rule, and to convert from base *b* representations of numbers to decimal representations among other things.

Format

```
y = recsercp(x, z);
```

Input

x	NxK or 1xK matrix
z	NxK or 1xK matrix.

Output

y	NxK matrix in which each column is a series generated by a recursion of the form:

```
y(1) = x(1) + z(1)
y(t) = y(t-1) * x(t) + z(t), t=2,...N
```

Remarks

The following **GAUSS** code could be used to emulate **recsercp** when the number of rows in *x* and *z* is the same:

```
    /* assume here that rows(z) is also n */
    n = rows(x);
    y = zeros(n, 1);
    y[1,.] = x[1,.] + z[1,.];

    i = 2;
    do until i > n;
        y[i,.] = y[i-1,.] .* x[i,.] + z[i,.];
        i = i +1;
    endo;
```

Note that K series can be computed simultaneously, since *x* and *z* can have K columns (they must both have the same number of columns).

recsercp allows either *x* or *z* to have only 1 row.

recsercp(*x*, *0*) will produce the cumulative products of the elements in *x*.

Example

```
    c1 = c[1,.];
    n = rows(c) - 1;
    y = recsercp(x, trimr(c ./ c1,1,0));
    p = c1 .* y[n,.];
```

If *x* is a scalar and *c* is an (N+1)x1 vector, the result *p* will contain the value of the polynomial whose coefficients are given in *c*. That is:

```
    p = c[1,.].*x^n + c[2,.].*x^(n-1) + ... + c[n+1,.];
```

Note that both *x* and *c* could contain more than 1 column, and then this code would evaluate the entire set of polynomials at the same time. Note also that if *x* = 2, and if *c* contains the digits of the binary representation of a number, then *p* will be the decimal representation of that number.

r

See Also

recserar, recserrc

recserrc

Purpose

Computes a recursive series involving division.

Format

y = **recserrc**(*x*, *z*);

Input

x	1xK or Kx1 vector.
z	NxK matrix.

Output

y	NxK matrix in which each column is a series generated by a recursion of the form: $y[1] = x \bmod z[1], \quad x = trunc(x/z[1])$

$$y[2] = x \bmod z[2], \quad x = trunc(x/z[2])$$

$$y[3] = x \bmod z[3], \quad x = trunc(x/z[3])$$

$$\vdots$$

$$y[n] = x \bmod z[n]$$

Remarks

Can be used to convert from decimal to other number systems (radix conversion).

Example

```
x = 2|8|10;
b = 2;
n = maxc(log(x)./log(b) ) + 1;
z = reshape(b, n, rows(x));
y = rev(recserrc(x, z) )';
```

The result, *y*, will contain in its rows (note that it is transposed in the last step) the digits representing the decimal numbers 2, 8, and 10 in base 2:

```
0 0 1 0
1 0 0 0
1 0 1 0
```

Source

`recserrc.src`

See Also

recserar, recsercp

renamefile

Purpose

Changes file name.

Format

```
ret = renamefile("oldname","newname");
```

Input

oldname	string, existing file name.
newname	string, new file name.

Output

ret	scalar, 0 if successful.

Example

```
    ret = renamefile("myfile.gss","mynewfile.gss");
```

In this example, a file in the current working directory with the name "myfile.gss" will be renamed "mynewfile.gss" in the same directory. Full path information may also be included:

```
//On Windows
ret = renamefile("c:\\gauss13\\myfile.gss",
  "c:\\gauss13\\mynewfile.gss");

//On Linux/Mac
ret = renamefile("/home/user/gauss13/myfile.gss",
  "/home/user/gauss13/mynewfile.gss");
```

rerun

Purpose

Displays the most recently created graphics file.

Library

pgraph

Format

```
rerun;
```

Remarks

`rerun` is used by the **endwind** function.

Source

pcart.src

Globals

_pcmdlin, *_pnotify*, *_psilent*, *_ptek*, *_pzoom*

reshape

Purpose

Reshapes a matrix.

Format

y = **reshape**(*x*, *r*, *c*);

Input

x	NxK matrix.
r	scalar, new row dimension.
c	scalar, new column dimension.

Output

y	*r* x *c* matrix created from the elements of *x*.

Remarks

Matrices are stored in row major order.

The first *c* elements are put into the first row of *y*, the second in the second row, and so on. If there are more elements in *x* than in *y*, the remaining elements are discarded. If there are not enough elements in *x* to fill *y*, then when **reshape** runs out of elements, it goes back to the first element of *x* and starts getting additional elements from there.

Example

```
    y = reshape(x,2,6);
```

```
            1  2  3  4
    if x =  5  6  7  8  then y = 1  2  3  4  5  6
            9 10 11 12           7  8  9 10 11 12
```

```
            1  2  3
    if x =  4  5  6  then y = 1  2  3  4  5  6
            7  8  9           7  8  9  1  2  3
```

```
            1  2  3  4  5
    if x =  6  7  8  9 10  then y = 1  2  3  4  5  6
           11 12 13 14 15           7  8  9 10 11 12
```

```
    if x = 1  2  then y = 1 2 3 4 1 2
           3  4           3 4 1 2 3 4
```

```
    if x = 1  then y = 1 1 1 1 1 1
                       1 1 1 1 1 1
```

See Also

submat, vec

retp

Purpose

Returns from a procedure or keyword.

Format

```
retp;
retp(x, y,...);
```

Remarks

For more details, see **Procedures and Keywords**, Chapter 1.

In a **retp** statement 0-1023 items may be returned. The items may be expressions. Items are separated by commas.

It is legal to return with no arguments, as long as the procedure is defined to return 0 arguments.

See Also

proc, keyword, endp

return

Purpose

Returns from a subroutine.

Format

```
    return;
    return(x,  y,...);
```

Remarks

The number of items that may be returned from a subroutine in a `return` statement is limited only by stack space. The items may be expressions. Items are separated by commas.

It is legal to return with no arguments and therefore return nothing.

See Also

gosub, pop

rev

Purpose

Reverses the order of the rows in a matrix.

Format

```
    y = rev(x);
```

Input

x	NxK matrix.

Output

y	NxK matrix containing the reversed rows of *x*.

Remarks

The first row of *y* will be where the last row of *x* was and the last row will be where the first was and so on. This can be used to put a sorted matrix in descending order.

Example

```
//Set the rng seed for repeatable results
rndseed 345345;

//Set print formatting to print 4 spaces for each column
//and 0 numbers after the decimal
format /rd 4,0

//Create some random integers
x = round(rndn(5,3)*10);

//Reverse the order of the columns
y = rev(x);

print "x = " x;
print "y = " y;
```

The code above produces the following output:

```
x =
  10  -14   -7
   3   -1   -5
  -7    4    2
```

```
       1     1     1
       7    -7     2
    y =
       7    -7     2
       1     1     1
      -7     4     2
       3    -1    -5
      10   -14    -7
```

See Also

sortc

rfft

Purpose

Computes a real 1- or 2-D Fast Fourier transform.

Format

y = **`rfft`**(*x*);

Input

x	NxK real matrix.

Output

y	LxM matrix, where L and M are the smallest powers of 2 greater than or equal to N and K, respectively.

Remarks

Computes the RFFT of *x*, scaled by *1/(L*M)*.

This uses a Temperton Fast Fourier algorithm.

If N or K is not a power of 2, *x* will be padded out with zeros before computing the transform.

See Also

rffti, fft, ffti, fftm, fftmi

rffti

Purpose

Computes inverse real 1- or 2-D Fast Fourier transform.

Format

y = **`rffti`**(*x*);

Input

x	NxK matrix.

Output

y	LxM real matrix, where L and M are the smallest prime factor products greater than or equal to N and K.

Remarks

It is up to the user to guarantee that the input will return a real result. If in doubt, use **`ffti`**.

See Also

rfft, fft, ffti, fftm, fftmi

rfftip

Purpose

Computes an inverse real 1- or 2-D FFT. Takes a packed format FFT as input.

Format

y = **`rfftip`**(*x*);

Input

x	NxK matrix or K-length vector.

Output

y	LxM real matrix or M-length vector.

Remarks

`rfftip` assumes that its input is of the same form as that output by **`rfftp`** and

rfftnp.

rfftip uses the Temperton prime factor FFT algorithm. This algorithm can compute the inverse FFT of any vector or matrix whose dimensions can be expressed as the product of selected prime number factors. **GAUSS** implements the Temperton algorithm for any integer power of 2, 3, and 5, and one factor of 7. Thus, **rfftip** can handle any matrix whose dimensions can be expressed as:

$$2^p \times 3^q \times 5^r \times 7^s$$

$$p, q, r \geq 0$$
$$s = 0 \text{ or } 1$$

If a dimension of *x* does not meet this requirement, it will be padded with zeros to the next allowable size before the inverse FFT is computed. Note that **rfftip** assumes the length (for vectors) or column dimension (for matrices) of *x* is K-1 rather than K, since the last element or column does not hold FFT information, but the Nyquist frequencies.

The sizes of *x* and *y* are related as follows: L will be the smallest prime factor product greater than or equal to N, and M will be twice the smallest prime factor product greater than or equal to K-1. This takes into account the fact that *x* contains both positive and negative frequencies in the row dimension (matrices only), but only positive frequencies, and those only in the first K-1 elements or columns, in the length or column dimension.

It is up to the user to guarantee that the input will return a real result. If in doubt, use **ffti**. Note, however, that **ffti** expects a full FFT, including negative frequency information, for input.

Do not pass **rfftip** the output from **rfft** or **rfftn**-it will return incorrect results. Use **rffti** with those routines.

See Also

fft, ffti, fftm, fftmi, fftn, rfft, rffti, rfftn, rfftnp, rfftp

rfftn

Purpose

Computes a real 1- or 2-D FFT.

Format

y = **`rfftn`**(*x*);

Input

x	NxK real matrix.

Output

y	LxM matrix, where L and M are the smallest prime factor products greater than or equal to N and K, respectively.

Remarks

`rfftn` uses the Temperton prime factor FFT algorithm. This algorithm can compute the FFT of any vector or matrix whose dimensions can be expressed as the product of selected prime number factors. **GAUSS** implements the Temperton algorithm for any power of 2, 3, and 5, and one factor of 7. Thus, **`rfftn`** can handle any matrix whose dimensions can be expressed as:

$2^p \times 3^q \times 5^r \times 7^s$

```
p, q, r ≥ 0      -- for rows of matrix
```

```
p > 0. q, r ≥ 0 -- for columns of matrix

p > 0. q, r ≥ 0 -- for length of a vector

s = 0 or 1      -- for all dimensions
```

If a dimension of *x* does not meet these requirements, it will be padded with zeros to the next allowable size before the FFT is computed.

`rfftn` pads matrices to the next allowable size; however, it generally runs faster for matrices whose dimensions are highly composite numbers, i.e., products of several factors (to various powers), rather than powers of a single factor. For example, even though it is bigger, a 33600x1 vector can compute as much as 20 percent faster than a 32768x1 vector, because 33600 is a highly composite number, 2^6x3x5^2x7, whereas 32768 is a simple power of 2, 2^{15}. For this reason, you may want to hand-pad matrices to optimum dimensions before passing them to **`rfftn`**. The **Run-Time Library** includes two routines, **`optn`** and **`optnevn`**, for determining optimum dimensions. Use **`optn`** to determine optimum rows for matrices, and **`optnevn`** to determine optimum columns for matrices and optimum lengths for vectors.

The **Run-Time Library** also includes the **`nextn`** and **`nextnevn`** routines, for determining allowable dimensions for matrices and vectors. (You can use these to see the dimensions to which **`rfftn`** would pad a matrix or vector.)

`rfftn` scales the computed FFT by 1/(L*M).

See Also

fft, ffti, fftm, fftmi, fftn, rfft, rffti, rfftip, rfftnp, rfftp

rfftnp

Purpose

Computes a real 1- or 2-D FFT. Returns the results in a packed format.

Format

y = **`rfftnp`**(*x*);

Input

x	NxK real matrix or K-length real vector.

Output

y	Lx(M/2+1) matrix or (M/2+1)-length vector, where L and M are the smallest prime factor products greater than or equal to N and K, respectively.

Remarks

For 1-D FFT's, **`rfftnp`** returns the positive frequencies in ascending order in the first M/2 elements, and the Nyquist frequency in the last element. For 2-D FFT's, **`rfftnp`** returns the positive and negative frequencies for the row dimension, and for the column dimension, it returns the positive frequencies in ascending order in the first M/2 columns, and the Nyquist frequencies in the last column. Usually the FFT of a real function is calculated to find the power density spectrum or to perform filtering on the waveform. In both these cases only the positive frequencies are required. (See also **`rfft`** and **`rfftn`** for routines that return the negative frequencies as well.)

rfftnp uses the Temperton prime factor FFT algorithm. This algorithm can compute the FFT of any vector or matrix whose dimensions can be expressed as the product of selected prime number factors. **GAUSS** implements the Temperton algorithm for any power of 2, 3, and 5, and one factor of 7. Thus, **rfftnp** can handle any matrix whose dimensions can be expressed as:

```
     2^p x 3^q x 5^r x 7^s

     p, q, r ≥ 0     -- for rows of matrix

     p > 0. q, r ≥ 0 -- for columns of matrix

     p > 0. q, r ≥ 0 -- for length of a vector

     s = 0 or 1      -- for all dimensions
```

If a dimension of *x* does not meet these requirements, it will be padded with zeros to the next allowable size before the FFT is computed.

rfftnp pads matrices to the next allowable size; however, it generally runs faster for matrices whose dimensions are highly composite numbers, i.e., products of several factors (to various powers), rather than powers of a single factor. For example, even though it is bigger, a 33600x1 vector can compute as much as 20 percent faster than a 32768x1 vector, because 33600 is a highly composite number, 2^6x3x5^2x7, whereas 32768 is a simple power of 2, 2^{15}. For this reason, you may want to hand-pad matrices to optimum dimensions before passing them to **rfftnp**. The **Run-Time Library** includes two routines, **optn** and **optnevn**, for determining optimum dimensions. Use **optn** to determine optimum rows for matrices, and **optnevn** to determine optimum columns for matrices and optimum lengths for vectors.

The **Run-Time Library** also includes the **nextn** and **nextnevn** routines, for determining allowable dimensions for matrices and vectors. (You can use these to see the dimensions to which **rfftnp** would pad a matrix or vector.)

rfftnp scales the computed FFT by 1/(L*M).

See Also

fft, ffti, fftm, fftmi, fftn, rfft, rffti, rfftip, rfftn, rfftp

rfftp

Purpose

Computes a real 1- or 2-D FFT. Returns the results in a packed format.

Format

y = **`rfftp`**(*x*);

Input

x	NxK real matrix or K-length real vector.

Output

y	Lx(M/2+1) matrix or (M/2+1)-length vector, where L and M are the smallest powers of 2 greater than or equal to N and K, respectively.

Remarks

If a dimension of *x* is not a power of 2, it will be padded with zeros to the next allowable size before the FFT is computed.

For 1-D FFT's, **rfftp** returns the positive frequencies in ascending order in the first M/2 elements, and the Nyquist frequency in the last element. For 2-D FFT's, **rfftp** returns the positive and negative frequencies for the row dimension, and for the column dimension, it returns the positive frequencies in ascending order in the first M/2 columns, and the Nyquist frequencies in the last column. Usually the FFT of a real function is calculated to find the power density spectrum or to perform filtering on the waveform. In both these cases only the positive frequencies are required. (See also **rfft** and **rfftn** for routines that return the negative frequencies as well.)

rfftp scales the computed FFT by 1/(L*M).

rfftp uses the Temperton FFT algorithm.

See Also

fft, ffti, fftm, fftmi, fftn, rfft, rffti, rfftip, rfftn, rfftnp

rndBernoulli

Purpose

Computes Bernoulli distributed random numbers.

Format

```
{ r, newstate } = rndBernoulli(r, c, prob, state);
r = rndBernoulli(r, c, prob);
```

Input

r	Scalar, number of rows of the output matrix.

c	Scalar, number of columns of the output matrix.
prob	Scalar, probability parameter.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

r	*r* x *c* matrix, Bernoulli random numbers.
newstate	Opaque vector, the updated state.

Example

```
//Bernoulli random numbers can be used to model qualitative
//binary data (i.e., yes/no, true/false), such as marital
//status.

//Set the random seed for repeatable numbers.
rndseed 723940439;

//The percentage of married people in the population we
//would like to model.
prob = 0.7;
```

```
//Create 10,000 Bernoulli random numbers
r = rndBernoulli(10000, 1, prob);

//The mean of 'r' should approximately equal 'prob'
mu = meanc(r);
print mu;
```

```
    0.70270000
```

See Also

rndMVn, rndCreateState

rndBeta

Purpose

Computes beta pseudo-random numbers with a choice of underlying random number generator.

Format

{ *x*, *newstate* } = **rndBeta**(*r*, *c*, *a*, *b*, *state*);
x = **rndBeta**(*r*, *c*, *a*, *b*);

Input

r	Scalar, number of rows of resulting matrix.
c	Scalar, number of columns of resulting matrix.

a	*r* x *c* matrix, or *r* x 1 vector, or 1 x *c* vector, or scalar, first shape argument for beta distribution.
b	*r* x *c* matrix, or *r* x 1 vector, or 1 x *c* vector, or scalar, second shape argument for beta distribution.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

x	*r* x *c* matrix, beta distributed random numbers.
newstate	Opaque vector, the updated state.

Remarks

The properties of the pseudo-random numbers in *x* are:

$$E(x) = a/(a+b)$$
$$Var(x) = a*b/((a+b+1)*(a+b^2))$$
$$0 < x < 1 a > 0 b > 0$$

r and *c* will be truncated to integers if necessary.

Technical Notes

The default generator for **`rndBeta`** is the SFMT Mersenne-Twister 19937. You can specifiy a different underlying random number generator with the function **`rndCreateState`**.

See Also

rndCreateState, rndStateSkip

rndCauchy

Purpose

Computes Cauchy random numbers with a choice of underlying random number generator.

Format

{ *`r`*, *`newstate`* } = **`rndCauchy`**(*`rows`*, *`cols`*, *`location`*, *`scale`*, *`state`*);
`r` = **`rndCauchy`**(*`rows`*, *`cols`*, *`location`*, *`scale`*);

Input

`rows`	Scalar, number of rows of resulting matrix.
`cols`	Scalar, number of columns of resulting matrix.
`location`	Scalar or ExE conformable matrix with *`rows`* and *`cols`*.
`scale`	Scalar or ExE conformable matrix with *`rows`* and *`cols`*.

state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the standard random number functions.

Output

r	*rows* x *cols* matrix, Cauchy distributed random numbers.
newstate	Opaque vector, the updated state.

See Also

rndCreateState, rndStateSkip

rndcon, rndmult, rndseed

Purpose

Resets the parameters of the linear congruential random number generator (RNG) that is the basis for **`rndu`**, **`rndi`** and **`rndn`**.

Format

```
rndcon c;
rndmult a;
rndseed seed;
```

Input

c	scalar, constant for the random number generator.
a	scalar, multiplier for the random number generator.
seed	scalar, initial seed for the random number generator.

Parameter default values and ranges:

```
seed     time(0)        0 < seed < 2^32
a        1664525        0 < a < 2^32
c        1013904223     0 < a < 2^32
```

Remarks

A linear congruential uniform random number generator is used by **rndLCu**, and is also called by **rndLCn**. These statements (`rndcon`, `rndmult` and `rndseed`) allow the parameters of this generator to be changed. Prior to **GAUSS** version 12, the functions **rndn** and **rndu** also used a linear congruential generator. The current version of **rndn** and **rndu** use the Mersenne-Twister by default. `rndseed` will set the seed for the **rndn** and **rndu** if they are used without passing in a state variable. However, **rndcon** and **rndmult** do not change any of their parameters.

The procedure used by the linear congruential RNG's to generate the uniform random numbers is as follows. First, the current "seed" is used to generate a new seed:

```
new_seed = (((a * seed) % 2^32)+ c) % 2^32
```

(where `%` is the mod operator). Then a number between 0 and 1 is created by dividing the new seed by 2^{32}:

```
    x =  new_seed / 2^32
```

`rndcon` resets *c*.

`rndmult` resets *a*.

`rndseed` resets *seed*. This is the initial seed for the generator. The default is that **GAUSS** uses the clock to generate an initial seed when **GAUSS** is invoked.

GAUSS goes to the clock to seed the generator only when it is first started up. Therefore when using the older linear congruential generators, if **GAUSS** is allowed to run for a long time, and if large numbers of random numbers are generated, there is a possibility of recycling (that is, the sequence of "random numbers" will repeat itself). However, the generator used has an extremely long cycle, so that should not usually be a problem. NOTE: The default RNG for **rndn** and **rndu**, the Mersenne-Twister, has an extremely long period of 2^{19937}.

The parameters set by these commands remain in effect until new commands are encountered, or until **GAUSS** is restarted.

See Also

rndu, rndn, rndi, rndLCi, rndKMi

rndCreateState

Purpose

Creates a new random number stream for a specified generator type from a seed value.

Format

state = **rndCreateState**(*brng*, *seed*);

Input

brng	String, generator name. Options include:	
	"mrg32k3a"	L'Ecuyer's MRG32K3A
	"mt19937"	Mersenne-Twister 19937
	"sfmt19937"	optimized Mersenne-Twister 19937
	"mt2203-01"	Mersenne-Twister 2203
	"wh-01"	Wichmann-Hill
seed	Scalar, starting seed value. if -1, **GAUSS** computes the starting seed based on the system clock.	

Output

state	Opaque vector, the newly created state.

Example

```
seed = 123456;
state = rndCreateState("mrg32k3a", seed);
{ r, newstate } = rndn(5, 1, state);
```

```
     0.51489262
     0.14053340
r = 1.2128406
     0.17112172
    -0.18788202
```

```
seed = 123456;

//Create a state from the 1028th substream of the
//Mersenne-Twister 2203 RNG
stateMT = rndCreateState("mt2203-1028", seed);

//Create a state from the 112th substream of the
//Wichmann-Hill RNG
stateWH = rndCreateState("wh-112", seed);

//Generate numbers using the states
{ r1, stateMT } = rndu(4, 1, stateMT);
{ r2, stateWH } = rndu(4, 1, stateWH);
```

Remarks

The states returned from this function may NOT be used with **rndMTu** or any of the **rndKM** or **rndLC** functions.

See Also

rndStateSkip, rndn, rndu, rndBeta

rndExp

Purpose

Computes exponentially distributed random numbers with a choice of underlying random number generator.

Format

```
{ r, newstate } = rndExp(rows,  cols, scale, state);
r = rndExp(rows, cols, scale);
```

Input

rows	Scalar, number of rows of resulting matrix.
cols	Scalar, number of columns of resulting matrix.
location	Scalar or ExE conformable matrix with *rows* and *cols*.
scale	Scalar or ExE conformable matrix with *rows* and *cols*.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

r	*rows* x *cols* matrix, exponentially distributed random numbers.
newstate	Opaque vector, the updated state.

See Also

rndCreateState, rndStateSkip

rndgam

Purpose

Computes pseudo-random numbers with gamma distribution.

Format

```
x = rndgam(r, c, alpha);
```

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
alpha	MxN matrix, ExE conformable with *r* x *c* resulting matrix, shape parameters for gamma distribution.

Output

x	*r* x *c* matrix, gamma distributed pseudo-random numbers.

Remarks

The properties of the pseudo-random numbers in *x* are:

$$E(x) \;=\; alphaVar(x) \;=\; alphax > 0alpha > 0$$

Source

`random.src`

rndGamma

Purpose

Computes gamma pseudo-random numbers with a choice of underlying random number generator.

Format

```
{ x, newstate } = rndGamma(r, c, shape, scale, state);
x = rndGamma(r, c, shape, scale);
```

Input

r	Scalar, number of rows of resulting matrix.

c	Scalar, number of columns of resulting matrix.
shape	*r* x *c* matrix, or *r* x 1 vector, or 1 x *c* vector, or scalar, shape argument for gamma distribution.
scale	*r* x *c* matrix, or *r* x 1 vector, or 1 x *c* vector, or scalar, scale argument for gamma distribution.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

x	*r* x *c* matrix, gamma distributed random numbers.
newstate	Opaque vector, the updated state.

Remarks

The properties of the pseudo-random numbers in *x* are:

$$E(x) \;=\; alphaVar(x) \;=\; alphax > 0alpha > 0$$

Technical Notes

The default generator for **`rndGammma`** is the SFMT Mersenne-Twister 19937. You can specifiy a different underlying random number generator with the function **`rndCreateState`**.

See Also

rndCreateState, rndStateSkip

rndGeo

Purpose

Computes geometric pseudo-random numbers with a choice of underlying random number generator.

Format

{ *y*, *`newstate`* } = **`rndGeo`**(*`r`*, *`c`*, *`prob`*, *`state`*);
y = **`rndGeo`**(*`r`*, *`c`*, *`prob`*);

Input

`r`	Scalar, row dimension.
`c`	Scalar, column dimension.
`prob`	Scalar or matrix: ExE conformatble with *`r`* and *`c`* columns.
`state`	Optional argument - scalar or opaque vector.
	Scalar case:

	`state` = starting seed value. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** `state` = the state vector returned from a previous call to one of the **`rnd`** random number generators.

Output

y	*r* x *c* matrix of geometrically distributed random numbers.
newstate	Opaque vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

See Also

rndCreateState, rndStateSkip

Technical Notes

The default generator for **`rndGeo`** is the SFMT Mersenne-Twister 19937. You can specifiy a different underlying random number generator with the function **`rndCreateState`**.

rndGumbel

Purpose

Computes Gumbel distributed random numbers with a choice of underlying random number generator.

Format

{ *r*, *newstate* } = **rndGumbel**(*rows*, *cols*, *location*, *scale*, *state*);
r = **rndGumbel**(*rows*, *cols*, *scale*);

Input

rows	Scalar, number of rows of resulting matrix.
cols	Scalar, number of columns of resulting matrix.
location	Scalar or ExE conformable matrix with *rows* and *cols*.
scale	Scalar or ExE conformable matrix with *rows* and *cols*.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

r	*rows* x *cols* matrix, Gumbel distributed random numbers.
newstate	Opaque vector, the updated state.

See Also

rndCreateState, rndStateSkip

rndi

Purpose

Returns a matrix of random integers, $0 \le y < 2^{32}$.

Format

y = **rndi**(*r*, *c*);

Input

r	scalar, row dimension.
c	scalar, column dimension.

Output

y	*r* x *c* matrix of random integers between 0 and 2^{32}-1, inclusive.

Remarks

r and *c* will be truncated to integers if necessary.

This generator is automatically seeded using the system clock when **GAUSS** first starts. However, that can be overridden using the `rndseed` statement.

Each seed is generated from the preceding seed, using the formula

$$new\_seed = (((a * seed) \% 2^{32})+c) \% 2^{32}$$

where `%` is the mod operator. The new seeds are the values returned. The multiplicative constant and the additive constant may be changed using `rndmult` and `rndcon` respectively.

See Also

rndu, rndn, rndcon, rndmult

rndKMbeta

Purpose

Computes beta pseudo-random numbers.

Format

{ *x*, *newstate* } = **rndKMbeta**(*r*, *c*, *a*, *b*, *state*);

Input

r	scalar, number of rows of resulting matrix.

c	scalar, number of columns of resulting matrix.
a	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, first shape argument for beta distribution.
b	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, second shape argument for beta distribution.
state	scalar or 500x1 vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **500x1 vector case:** *state* = the state vector returned from a previous call to one of the **rndKM** random number functions.

Output

x	*r* x *c* matrix, beta distributed random numbers.
newstate	500x1 vector, the updated state.

Remarks

The properties of the pseudo-random numbers in *x* are:

$$E(x) = a/(a+b)$$
$$Var(x) = a*b/((a+b+1)*(a+b^2)$$
$$0 < x < 1a > 0b > 0$$

r and *c* will be truncated to integers if necessary.

Source

`randkm.src`

Technical Notes

`rndKMbeta` uses the recur-with-carry KISS+Monster algorithm described in the **`rndKMi`** Technical Notes.

rndKMgam

Purpose

Computes Gamma pseudo-random numbers.

Format

{ *x*, *newstate* } = **`rndKMgam`**(*r*, *c*, *alpha*, *state*);

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
alpha	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, shape argument for gamma distribution.
state	scalar or 500x1 vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock.

	500x1 vector case: `state` = the state vector returned from a previous call to one of the **`rndKM`** random number functions.

Output

`x`	*`r`* x *`c`* matrix, gamma distributed random numbers.
`newstate`	500x1 vector, the updated state.

Remarks

The properties of the pseudo-random numbers in *`x`* are:

$$E(x) = alpha Var(x) = alpha x > 0 alpha > 0$$

To generate **`gamma`**(*`alpha`*, *`theta`*) pseudo-random numbers where *`theta`* is a scale parameter, multiply the result of **`rndKMgam`** by *`theta`*.

Thus

```
z = theta * rndgam(1,1, alpha);
```

has the properties

$$E(z) = alpha * theta Var(z) = alpha * theta^2 z > 0 alpha > 0 theta > 0$$

`r` and *`c`* will be truncated to integers if necessary.

Source

randkm.src

Technical Notes

`rndKMgam` uses the recur-with-carry KISS+Monster algorithm described in the **`rndKMi`** Technical Notes.

rndKMi

Purpose

Returns a matrix of random integers, $0 \leq y < 2^{32}$, and the state of the random number generator.

Format

{ *y*, *`newstate`* } = **`rndKMi`**(*r*, *c*, *`state`*);

Input

r	scalar, row dimension.
c	scalar, column dimension.
`state`	scalar or 500x1 vector. **Scalar case:** *`state`* = starting seed value. If -1, **GAUSS** computes the starting seed based on the system clock. **500x1 vector case:** *`state`* = the state vector returned from a previous call to one of the **`rndKM`** random number generators.

Output

y	*r* x *c* matrix of random integers between 0 and 2^{32} - 1, inclusive.
newstate	500x1 vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Example

This example generates two thousand vectors of random integers, each with one million elements. The state of the random number generator after each iteration is used as an input to the next generation of random numbers.

```
state = 13;
n = 2000;
k = 1000000;
c = 0;
min = 2^32+1;
max = -1;

do while c < n;
   { y,state } = rndKMi(k,1,state);
   min = minc(min | minc(y));
   max = maxc(max | maxc(y));
   c = c + k;
endo;

print "min " min;
print "max " max;
```

See Also

rndKMn, rndKMu

Technical Notes

rndKMi generates random integers using a KISS+Monster algorithm developed by George Marsaglia. KISS initializes the sequence used in the recur-with-carry Monster random number generator. For more information on this generator see http://www.Aptech.com/random.

rndKMn

Purpose

Returns a matrix of standard normal (pseudo) random variables and the state of the random number generator.

Format

{ *y*, *newstate* } = **rndKMn**(*r*, *c*, *state*);

Input

r	scalar, row dimension.
c	scalar, column dimension.
state	scalar or 500x1 vector. **Scalar case:** *state* = starting seed value. If -1, **GAUSS** computes the starting seed based on the system clock.

	500x1 vector case:
	`state` = the state vector returned from a previous call to one of the **`rndKM`** random number generators.

Output

y	*r* x *c* matrix of standard normal random numbers.
newstate	500x1 vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Example

This example generates two thousand vectors of standard normal random numbers, each with one million elements. The state of the random number generator after each iteration is used as an input to the next generation of random numbers.

```
state = 13;
n = 2000;
k = 1000000;
c = 0;
submean = {};

do while c < n;
   { y,state } = rndKMn(k,1,state);
   submean = submean | meanc(y);
   c = c + k;
endo;
```

```
    mean = meanc(submean);
    print mean;
```

See Also

rndKMu, rndKMi

Technical Notes

rndKMn calls the uniform random number generator that is the basis for **rndKMu** multiple times for each normal random number generated. This is the recur-with-carry KISS+Monster algorithm described in the **rndKMi** Technical Notes. Potential normal random numbers are filtered using the fast acceptance-rejection algorithm proposed by Kinderman, A.J. and J.G. Ramage, "Computer Generation of Normal Random Numbers," *Journal of the American Statistical Association*, December 1976, Volume 71, Number 356, pp. 893-896. It employs the error correction from Tirler et al. (2004), "An error in the Kinderman-Ramage method and how to fix it," *Computational and Data Analysis*, Vol. 47, 433-40.

rndKMnb

Purpose

Computes negative binomial pseudo-random numbers.

Format

{ *x*, *newstate* } = **rndKMnb**(*r*, *c*, *k*, *p*, *state*);

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
k	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, "event" argument for negative binomial distribution.
p	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, "probability" argument for negative binomial distribution.
state	scalar or 500x1 vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **500x1 vector case:** *state* = the state vector returned from a previous call to one of the **rndKM** random number functions.

Output

x	*r* x *c* matrix, negative binomial distributed random numbers.
newstate	500x1 vector, the updated state.

Remarks

The properties of the pseudo-random numbers in *x* are:

$$E(x) = (k * p)/(1 - p)$$
$$Var(x) = (k * p)/(1 - p)^2 x = 0, 1,....k > 00 < p < 1$$

r and *c* will be truncated to integers if necessary.

Source

`randkm.src`

Technical Notes

rndKMnb uses the recur-with-carry KISS+Monster algorithm described in the **rndKMi** Technical Notes.

rndKMp

Purpose

Computes Poisson pseudo-random numbers.

Format

{ *x*, *newstate* } = **rndKMp**(*r*, *c*, *lambda*, *state*);

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
lambda	*r* x *c* matrix, or *r* x 1 vector, or 1 x *c* vector, or scalar, shape argument for Poisson distribution.

`state`	scalar or 500x1 vector. **Scalar case:** `state` = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **500x1 vector case:** `state` = the state vector returned from a previous call to one of the **`rndKM`** random number functions.

Output

`x`	`r` x `c` matrix, Poisson distributed random numbers.
`newstate`	500x1 vector, the updated state.

Remarks

The properties of the pseudo-random numbers in `x` are:

$$E(x) \;=\; lambdaVar(x) \;=\; lambdax \;=\; 0,\, 1,....lambda \;>\; 0$$

`r` and `c` will be truncated to integers if necessary.

Source

`randkm.src`

Technical Notes

`rndKMp` uses the recur-with-carry KISS+Monster algorithm described in the **`rndKMi`** Technical Notes.

rndKMu

Purpose

Returns a matrix of uniform (pseudo) random variables and the state of the random number generator.

Format

{ *y*, *newstate* } = **rndKMu**(*r*, *c*, *state*);

Input

r	scalar, row dimension.
c	scalar, column dimension.
state	scalar, 2x1 vector, or 500x1 vector. **Scalar case:** *state* = starting seed value. If -1, **GAUSS** computes the starting seed based on the system clock. **2x1 vector case:** *[1]* the starting seed, uses the system clock if -1 *[2]* 0 for $0 \leq y < 1$ 1 for $0 \leq y \leq 1$ **500x1 vector case:** *state* = the state vector returned from a previous call to one of the **rndKM** random number generators.

Output

y	*r* x *c* matrix of uniform random numbers, $0 \le y < 1$.
newstate	500x1 vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Example

This example generates two thousand vectors of uniform random numbers, each with one million elements. The state of the random number generator after each iteration is used as an input to the next generation of random numbers.

```
state = 13;
n = 2000;
k = 1000000;
c = 0;
submean = {};

do while c < n;
   { y,state } = rndKMu(k,1,state);
   submean = submean | meanc(y);
   c = c + k;
endo;

mean = meanc(submean);
print 0.5-mean;
```

See Also

rndKMn, rndKMi

Technical Notes

`rndKMu` uses the recur-with-carry KISS-Monster algorithm described in the **`rndKMi`** Technical Notes. Random integer seeds from 0 to 2^{32}-1 are generated. Each integer is divided by 2^{32} or 2^{32}-1.

rndKMvm

Purpose

Computes von Mises pseudo-random numbers.

Format

```
{ x, newstate } = rndKMvm(r, c,  m,  k, state);
```

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
m	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, means for vm distribution.
k	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, shape argument for vm distribution.
state	scalar or 500x1 vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock.

	500x1 vector case: *state* = the state vector returned from a previous call to one of the **rndKM** random number functions.

Output

x	*r* x *c* matrix, von Mises distributed random numbers.
newstate	500x1 vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Source

`randkm.src`

Technical Notes

rndKMvm uses the recur-with-carry KISS+Monster algorithm described in the **rndKMi** Technical Notes.

rndLaplace

Purpose

Computes Laplacian pseudo-random numbers with the choice of underlying random number generator.

Format

```
{ x, newstate } = rndLaplace(r, c, loc, scale, state);
x = rndLaplace(r, c, loc, scale);
```

Input

r	Scalar, number of rows of resulting matrix.
c	Scalar, number of columns of resulting matrix.
loc	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, location parameter.
scale	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, scale parameter.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

x	*r* x *c* matrix, Laplacian distributed random numbers.
newstate	Opaque vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Technical Notes

The default generator for **`rndLaplace`** is the SFMT Mersenne-Twister 19937. You can specifiy a different underlying random number generator with the function **`rndCreateState`**.

See Also

rndCreateState, rndStateSkip

rndLCbeta

Purpose

Computes beta pseudo-random numbers. NOTE: This function is deprecated--use **`rndBeta`**--but remains for backward compatibility.

Format

{ *x*, *newstate* } = **`rndLCbeta`**(*r*, *c*, *a*, *b*, *state*);

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
a	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, first shape argument for beta distribution.

b	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, second shape argument for beta distribution.
state	scalar, or 3x1 vector, or 4x1 vector.
	Scalar case:
	state = starting seed value only. System default values are used for the additive and multiplicative constants.
	The defaults are 1013904223, and 1664525, respectively. These may be changed with `rndcon` and `rndmult`.
	If *state* = -1, **GAUSS** computes the starting seed based on the system clock.
	3x1 vector case:
	`[1]` the starting seed, uses the system clock if -1
	`[2]` the multiplicative constant
	`[3]` the additive constant
	4x1 vector case:
	state = the state vector returned from a previous call to one of the **rndLC** random number generators.

Output

x	*r* x *c* matrix, beta distributed random numbers.
newstate	4x1 vector:
	`[1]` the updated seed
	`[2]` the multiplicative constant

`[3]` the additive constant

`[4]` the original initialization seed

Source

```
randlc.src
```

Technical Notes

This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = (((a * seed) \% 2^{32}) + c) \% 2^{32}$$

where `%` is the mod operator and where `a` is the multiplicative constant and `c` is the additive constant.

rndLCgam

Purpose

Computes Gamma pseudo-random numbers. NOTE: This function is deprecated--use rndGamma--but remains for backward compatibility.

Format

{ *x*, *newstate* } = **rndLCgam**(*r*, *c*, *alpha*, *state*);

Input

`r`	scalar, number of rows of resulting matrix.
`c`	scalar, number of columns of resulting matrix.
`alpha`	`r` x `c` matrix, or `r`x1 vector, or 1x`c` vector, or scalar, shape argument for gamma distribution.
`state`	scalar, or 3x1 vector, or 4x1 vector. **Scalar case:** `state` = starting seed value only. System default values are used for the additive and multiplicative constants. The defaults are 1013904223, and 1664525, respectively. These may be changed with `rndcon` and `rndmult`. If `state` = -1, **GAUSS** computes the starting seed based on the system clock. **3x1 vector case:** `[1]` the starting seed, uses the system clock if -1 `[2]` the multiplicative constant `[3]` the additive constant **4x1 vector case:** `state` = the state vector returned from a previous call to one of the **`rndLC`** random number generators.

Output

x	*r* x *c* matrix, gamma distributed random numbers.
newstate	4x1 vector:
	[1] the updated seed
	[2] the multiplicative constant
	[3] the additive constant
	[4] the original initialization seed

Source

randlc.src

Technical Notes

This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = (((a * seed) \% 2^{32}) + c) \% 2^{32}$$

where % is the mod operator and where a is the multiplicative constant and c is the additive constant.

rndLCi

Purpose

Returns a matrix of random integers, $0 \le y < 2^{32}$, and the state of the random number generator. NOTE: This function is deprecated but remains for backward compatibility.

Format

{ *y*, *newstate* } = **rndLCi**(*r*, *c*, *state*);

Input

r	scalar, row dimension.
c	scalar, column dimension.
state	scalar, or 3x1 vector, or 4x1 vector.

Scalar case:

state = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with rndcon and rndmult.

If *state* < 0, **GAUSS** computes the starting seed based on the system clock.

3x1 vector case:

[1]	the starting seed, uses the system clock if < 0
[2]	the multiplicative constant
[3]	the additive constant

4x1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

Output

y	*r* x *c* matrix of random integers between 0 and 2^{32} - 1, inclusive.
newstate	4x1 vector: *[1]* the updated seed *[2]* the multiplicative constant *[3]* the additive constant *[4]* the original initialization seed

Remarks

r and *c* will be truncated to integers if necessary.

Each seed is generated from the preceding seed, using the formula

```
new_seed = (((a * seed) % 2^32)+ c) % 2^32
```

where `%` is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant. The new seeds are the values returned.

Example

```
state = 13;
n = 2000000000;
k = 1000000;
c = 0;
min = 2^32+1;
max = -1;
```

```
do while c < n;
   { y,state } = rndLCi(k,1,state);
   min = minc(min | minc(y));
   max = maxc(max | maxc(y));
   c = c + k;
endo;

print "min " min;
print "max " max;
```

See Also

rndLCn, rndLCu, rndcon, rndmult

rndLCn

Purpose

Returns a matrix of standard normal (pseudo) random variables and the state of the random number generator. NOTE: This function is deprecated--use **rndn**--but remains for backward compatibility.

Format

{ *y*, *newstate* } = **rndLCn**(*r*, *c*, *state*);

Input

r	scalar, row dimension.

c	scalar, column dimension.
state	scalar, or 3x1 vector, or 4x1 vector. **Scalar case:** *state* = starting seed value only. System default values are used for the additive and multiplicative constants. The defaults are 1013904223, and 1664525, respectively. These may be changed with `rndcon` and `rndmult`. **3x1 vector case:** `[1]` the starting seed, uses the system clock if < 0 If *state* < 0, **GAUSS** computes the starting seed based on the system clock. `[2]` the multiplicative constant `[3]` the additive constant **4x1 vector case:** *state* = the state vector returned from a previous call to one of the **`rndLC`** random number generators.

Output

y	*r* x *c* matrix of standard normal random numbers.
newstate	4x1 vector: `[1]` the updated seed `[2]` the multiplicative constant

[3]	the additive constant
[4]	the original initialization seed

Remarks

r and *c* will be truncated to integers if necessary.

Example

```
state = 13;
n = 2000000000;
k = 1000000;
c = 0;
submean = {};

do while c < n;
   { y,state } = rndLCn(k,1,state);
   submean = submean | meanc(y);
   c = c + k;
endo;

mean = meanc(submean);
print mean;
```

See Also

rndLCu, rndLCi, rndcon, rndmult

Technical Notes

The normal random number generator is based on the uniform random number generator, using the fast acceptance-rejection algorithm proposed by Kinderman, A.J. and J.G. Ramage, "Computer Generation of Normal Random Numbers," *Journal of the American*

Statistical Association, December 1976, Volume 71, Number 356, pp. 893-896. This algorithm calls the linear congruential uniform random number generator multiple times for each normal random number generated. See rndLCu for a description of the uniform random number generator algorithm.

rndLCnb

Purpose

Computes negative binomial pseudo-random numbers. NOTE: This function is deprecated--use **`rndNegBinomial`**--but remains for backward compatibility.

Format

{ *x*, *newstate* } = **rndLCnb**(*r*, *c*, *k*, *p*, *state*);

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
k	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, "event" argument for negative binomial distribution.
p	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, "probability" argument for negative binomial distribution.
state	scalar, or 3x1 vector, or 4x1 vector.
	Scalar case:
	state = starting seed value only. System default values are

used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with `rndcon` and `rndmult`.

If *state* = -1, **GAUSS** computes the starting seed based on the system clock.

3x1 vector case:

[1] the starting seed, uses the system clock if -1

[2] the multiplicative constant

[3] the additive constant

4x1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

Output

x	*r* x *c* matrix, negative binomial distributed random numbers.
newstate	4x1 vector:
	[1] the updated seed
	[2] the multiplicative constant
	[3] the additive constant
	[4] the original initialization seed

Source

randlc.src

Technical Notes

This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = ((a * seed) \% 2^{32}) + c) \% 2^{32}$$

where `%` is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant.

rndLCp

Purpose

Computes Poisson pseudo-random numbers. NOTE: This function is deprecated--use **rndPoisson**--but remains for backward compatibility.

Format

{ *x*, *newstate* } = **rndLCp**(*r*, *c*, *lambda*, *state*);

Input

r	scalar, row dimension.
c	scalar, column dimension.
lambda	scalar, mean parameter.
state	scalar, or 3x1 vector, or 4x1 vector.
	Scalar case:

`state` = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with `rndcon` and `rndmult`.

3x1 vector case:

`[1]`	the starting seed, uses the system clock if < 0

If `state` < 0, **GAUSS** computes the starting seed based on the system clock.

`[2]`	the multiplicative constant
`[3]`	the additive constant

4x1 vector case:

`state` = the state vector returned from a previous call to one of the **`rndLC`** random number generators.

Output

`x`	*`r`* x *`c`* matrix of Poisson distributed random numbers.
`newstate`	4x1 vector:
	`[1]` the updated seed
	`[2]` the multiplicative constant
	`[3]` the additive constant
	`[4]` the original initialization seed

Source

`randlc.src`

Technical Notes

This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = (((a * seed) \% 2^{32}) + c) \% 2^{32}$$

where `%` is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant.

rndLCu

Purpose

Returns a matrix of uniform (pseudo) random variables and the state of the random number generator. NOTE: This function is deprecated but remains for backward compatibility.

Format

{ *y*, *newstate* } = **rndLCu**(*r*, *c*, *state*);

Input

r	scalar, row dimension.
c	scalar, column dimension.

`state`	scalar, or 3x1 vector, or 4x1 vector. **Scalar case:** `state` = starting seed value only. System default values are used for the additive and multiplicative constants. The defaults are 1013904223, and 1664525, respectively. These may be changed with `rndcon` and `rndmult`. **3x1 vector case:** `[1]` the starting seed, uses the system clock if < 0 If `state` < 0, **GAUSS** computes the starting seed based on the system clock. `[2]` the multiplicative constant `[3]` the additive constant **4x1 vector case:** `state` = the state vector returned from a previous call to one of the **`rndLC`** random number generators.

Output

`y`	`r` x `c` matrix of uniform (0 < x < 1) random numbers.
`newstate`	4x1 vector: `[1]` the updated seed `[2]` the multiplicative constant `[3]` the additive constant

[4] the original initialization seed

Remarks

r and *c* will be truncated to integers if necessary.

Each seed is generated from the preceding seed, using the formula

$$new\_seed = (((a * seed) \% 2^{32}) + c) \% 2^{32}$$

where % is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant. A number between 0 and 1 is created by dividing *new_seed* by 2^{32}.

Example

```
state = 13;
n = 2000000000;
k = 1000000;
c = 0;
submean = {};

do while c < n;
   { y,state } = rndLCu(k,1,state);
   submean = submean | meanc(y);
   c = c + k;
endo;

mean = meanc(submean);
print 0.5-mean;
```

See Also

rndLCn, rndLCi, rndcon, rndmult

Technical Notes

This function uses a linear congruential method, discussed in Kennedy, W. J. Jr., and J. E. Gentle, *Statistical Computing*, Marcel Dekker, Inc., 1980, pp. 136-147.

rndLCvm

Purpose

Computes von Mises pseudo-random numbers. NOTE: This function is deprecated but remains for backward compatibility.

Format

{ *x*, *newstate* } = **rndLCvm**(*r*, *c*, *m*, *k*, *state*);

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
m	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, means for vm distribution.
k	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, shape argument for vm distribution.
state	scalar, or 3x1 vector, or 4x1 vector.
	Scalar case:
	state = starting seed value only. System default values are

used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with `rndcon` and `rndmult`.

If *`state`* = -1, **GAUSS** computes the starting seed based on the system clock.

3x1 vector case:

`[1]`	the starting seed, uses the system clock if -1
`[2]`	the multiplicative constant
`[3]`	the additive constant

4x1 vector case:

`state` = the state vector returned from a previous call to one of the **`rndLC`** random number generators.

Output

`x`	*`r`* x *`c`* matrix, von Mises distributed random numbers.
`newstate`	4x1 vector:
	`[1]` the updated seed
	`[2]` the multiplicative constant
	`[3]` the additive constant
	`[4]` the original initialization seed

Remarks

r and *c* will be truncated to integers if necessary.

Source

`randlc.src`

Technical Notes

This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = (((a * seed) \% 2^{32}) + c) \% 2^{32}$$

where `%` is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant.

rndLogNorm

Purpose

Computes lognormal pseudo-random numbers with the choice of underlying random number generator.

Format

{ *x*, *newstate* } = **rndLogNorm**(*r*, *c*, *mu*, *sigma*, *state*);
x = **rndLogNorm**(*r*, *c*, *mu*, *sigma*);

Input

r	Scalar, number of rows of resulting matrix.
c	Scalar, number of columns of resulting matrix.
mu	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, mean.
sigma	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, standard deviation.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

x	*r* x *c* matrix, lognormal distributed random numbers.
newstate	Opaque vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Technical Notes

The default generator for **`rndLogNorm`** is the SFMT Mersenne-Twister 19937. You can specifiy a different underlying random number generator with the function **`rndCreateState`**.

See Also

rndCreateState, rndStateSkip

rndMVn

Purpose

Computes multivariate normal random numbers given a covariance matrix.

Format

```
{ r, newstate } = rndMVn(num, mu, cov, state);
r = rndMVn(num, mu, cov);
```

Input

num	Scalar, number of random vectors to create.
mu	Nx1 matrix, mean vector.
cov	NxN covariance matrix.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the

starting seed based on the system clock.

Opaque vector case:

`state` = the state vector returned from a previous call to one of the **`rnd`** random number functions.

Output

`r`	*`num`*xN matrix, multivariate normal random numbers.
`newstate`	Opaque vector, the updated state.

See Also

rndCreateState, rndStateSkip

rndMVt

Purpose

Computes multivariate Student-t distributed random numbers given a covariance matrix.

Format

{ *`r`*, *`newstate`* } = **`rndMVt`**(*`num`*, *`cov`*, *`df`*, *`state`*);
`r` = **`rndMVt`**(*`num`*, *`cov`*, *`df`*);

Input

num	Scalar, number of random vectors to create.
cov	NxN covariance matrix.
df	Scalar, degrees of freedom.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

r	num x N matrix, multivariate student-t distributed random numbers.
newstate	Opaque vector, the updated state.

See Also

rndMVn, rndCreateState

rndn

Purpose

Computes normally distributed pseudo-random numbers with a choice of underlying random number generator.

Format

{ *y*, *newstate* } = **rndn**(*r*, *c*, *state*);
y = **rndn**(*r*, *c*);

Input

r	Scalar, row dimension.
c	Scalar, column dimension.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rndn** random number generators.

Output

y	*r* x *c* matrix of standard normal random numbers.

newstate	Opaque vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Example

This example generates two thousand vectors of standard normal random numbers, each with one million elements. The state of the random number generator after each iteration is used as an input to the next generation of random numbers.

```
  state = 13;
  n = 2000;
  k = 1000000;
  c = 0;
  submean = {};

  do while c < n;
     { y,state } = rndn(k,1,state);
     submean = submean | meanc(y);
     c = c + k;
  endo;

 mean = meanc(submean);
 print mean;
```

Technical Notes

The default generator for **rndn** is the SFMT Mersenne-Twister 19937. You can specify a different underlying random number generator with the function **rndCreateState**.

See Also

rndCreateState, rndStateSkip

rndnb

Purpose

Computes pseudo-random numbers with negative binomial distribution.

Format

```
x = rndnb(r, c, k, p);
```

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
k	MxN matrix, ExE conformable with *r* x *c* resulting matrix, "event" parameters for negative binomial distribution.
p	KxL matrix, ExE conformable with *r* x *c* resulting matrix, "probability" parameters for negative binomial distribution.

Output

x	*r* x *c* matrix, negative binomial distributed pseudo-random numbers.

Remarks

The properties of the pseudo-random numbers in *x* are:

$$\begin{aligned} E(x) &= k * p/(1-p) \\ Var(x) &= k * p/(1-p)^2 \\ x &= 0,1,2,\cdots,k \\ k &> 0 \\ p &> 0 \\ p &< 1 \end{aligned}$$

Source

`random.src`

rndNegBinomial

Purpose

Computes negative binomial pseudo-random numbers with a choice of underlying random number generator.

Format

{ *x*, *newstate* } = **rndNegBinomial**(*r*, *c*, *ns*, *prob*, *state*);
x = **rndNegBinomial**(*r*, *c*, *ns*, *prob*);

Input

r	Scalar, number of rows of resulting matrix.
c	Scalar, number of columns of resulting matrix.

ns	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, "event" argument for negative binomial distribution.
prob	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, "probability" argument for negative binomial distribution.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the state returning random number functions.

Output

x	*r* x *c* matrix, negative binomial distributed random numbers.
newstate	Opaque vector, the updated state.

Remarks

The properties of the pseudo-random numbers in *x* are:

$$E(x)=\frac{k*p}{(1-p)},Var(x)=\frac{k*p}{(1-p)^2}$$
$$x=0,1,\ldots,k>0,0<p<1$$

r and *c* will be truncated to integers if necessary.

Technical Notes

The default generator for **`rndNegBinomial`** is the SFMT Mersenne-Twister 19937. You can specifiy a different underlying random number generator with the function **`rndCreateState`**.

See Also

rndCreateState, rndStateSkip

rndp

Purpose

Computes pseudo-random numbers with Poisson distribution.

Format

x = **rndp**(*r*, *c*, *lambda*);

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
lambda	MxN matrix, ExE conformable with *r* x *c* resulting matrix, shape parameters for Poisson distribution.

Output

x	*r* x *c* matrix, Poisson distributed pseudo-random numbers.

Remarks

The properties of the pseudo-random numbers in *x* are:

$$\begin{aligned} E(x) &= lambda \\ Var(x) &= lambda \\ x &= 0,1,2,\ldots \\ lambda &> 0 \end{aligned}$$

Source

```
random.src
```

rndPoisson

Purpose

Computes Poisson pseudo-random numbers with a choice of underlying random number generator.

Format

{ *x*, *newstate* } = **rndPoisson**(*r*, *c*, *lambda*, *state*);
x = **rndPoisson**(*r*, *c*, *lambda*);

Input

r	Scalar, number of rows of resulting matrix.
c	Scalar, number of columns of resulting matrix.
lambda	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, mean

	parameter for Poisson distribution.
state	Optional argument, scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rndMT** random number functions.

Output

x	*r* x *c* matrix, Poisson distributed random numbers.
newstate	Opaque vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Technical Notes

The default generator for **rndPoisson** is the SFMT Mersenne-Twister 19937. You can specifiy a different underlying random number generator with the function **rndCreateState**.

See Also

rndCreateState, rndStateSkip

rndStateSkip

Purpose

To advance a state vector by a specified number of values.

Format

newState = **rndStateSkip**(*numSkip*, *state*);

Input

numSkip	Scalar, the number of values to skip.
state	Opaque state vector.

Output

newState	Opaque vector, the advanced state.

Example

```
seed = 9192834;

//Create a state from the 118th substream of the
//Wichmann-Hill RNG
state = rndCreateState(wh-118", seed);

//Create a new state that is advanced by 2 numbers.
newState = rndStateSkip(2, state);

//Create and compare numbers from the two state vectors
```

```
{ r, state } = rndu(4, 1, state };
{ r2, newState } = rndu(2, 1, newState);
```

```
        0.54973563
r =   0.81642451
        0.68583300
        0.09105558

r2 = 0.68583300
        0.09105558
```

Technical Notes

This function applies ONLY to the MRG32K3A and Wichmann-Hill random number generators.

See Also

rndCreateState, rndn, rndu, rndBeta, rndGamma

rndu

Purpose

Computes uniform random numbers with a choice of underlying random number generator.

Format

{ *y*, *newstate* } = **rndu**(*r*, *c*, *state*);
y = **rndu**(*r*, *c*);

Input

r	Scalar, row dimension.
c	Scalar, column dimension.
state	Optional argument - scalar, or opaque vector. **Scalar case:** *state* = starting seed value. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number generators.

Output

y	*r* x *c* matrix of uniform random numbers, `0 <= y < 1`.
newstate	Opaque vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Example

This example generates two thousand vectors of uniform random numbers, each with one million elements. The state of the random number generator after each iteration is used as an input to the next generation of random numbers.

```
state = 13;
n = 2000;
k = 1000000;
c = 0;
submean = {};

do while c < n;
   { y,state } = rndu(k,1,state);
   submean = submean | meanc(y);
   c = c + k;
endo;

mean = meanc(submean);
print 0.5-mean;
```

See Also

rndCreateState, rndStateSkip

Technical Notes

The default generator for **rndu** is the SFMT Mersenne-Twister 19937. You can specifiy a different underlying random number generator with the function **rndCreateState**.

rndvm

Purpose

Computes von Mises pseudo-random numbers.

Format

```
x = rndvm(r, c,  m,  k);
```

Input

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
m	NxK matrix, ExE conformable with *r* x *c*, means for von Mises distribution.
k	LxM matrix, ExE conformable with *r* x *c*, shape argument for von Mises distribution.

Output

x	*r* x *c* matrix, von Mises distributed random numbers.

Source

`random.src`

rotater

Purpose

Rotates the rows of a matrix.

Format

y = **rotater**(*x*, *r*);

Input

x	NxK matrix to be rotated.
r	Nx1 or 1x1 matrix specifying the amount of rotation.

Output

y	NxK rotated matrix.

Remarks

The rotation is performed horizontally within each row of the matrix. A positive rotation value will cause the elements to move to the right. A negative rotation value will cause the elements to move to the left. In either case, the elements that are pushed off the end of the row will wrap around to the opposite end of the same row.

If the rotation value is greater than or equal to the number of columns in *x*, then the rotation value will be calculated using (*r* % **cols**(*x*)).

Example

```
        y = rotater(x,r);
```

```
           1   2   3            1            3   1   2
    If x =             and r =      Then y =
           4   5   6           -1            5   6   4
```

```
             1  2  3          0           1  2  3

             4  5  6          1           6  4  5
     If x =            and r =     Then y =
             7  8  9          2           8  9  7

            10 11 12          3          10 11 12
```

See Also

shiftr

rndWeibull

Purpose

Computes Weibull pseudo-random numbers with the choice of underlying random number generator.

Format

{ *x*, *newstate* } = **rndWeibull**(*r*, *c*, *shape*, *scale*, *state*);
x = **rndWeibull**(*r*, *c*, *shape*, *scale*);

Input

r	Scalar, number of rows of resulting matrix.
c	Scalar, number of columns of resulting matrix.
shape	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, shape

	parameter.
scale	*r* x *c* matrix, or *r*x1 vector, or 1x*c* vector, or scalar, scale parameter.
state	Optional argument - scalar or opaque vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **Opaque vector case:** *state* = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

x	*r* x *c* matrix, Weibull distributed random numbers.
newstate	Opaque vector, the updated state.

Remarks

r and *c* will be truncated to integers if necessary.

Technical Notes

The default generator for **rndWeibull** is the SFMT Mersenne-Twister 19937. You can specifiy a different underlying random number generator with the function **rndCreateState**.

See Also

rndCreateState, rndStateSkip

rndWishart

Purpose

Computes Wishart distributed random numbers given a covariance matrix.

Format

{ *r*, *newstate* } = **rndWishart**(*numMats*, *cov*, *df*, *state*);
r = **rndWishart**(*numMats*, *cov*, *df*);

Input

numMats	Scalar, number of Wishart random matrices to create.
cov	NxM covariance matrix.
df	Scalar, degrees of freedom.
state	Optional argument - scalar or opaque vector.

Scalar case:

state = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock.

Opaque vector case:

state = the state vector returned from a previous call to one of the **rnd** random number functions.

Output

r	*numMats* * **rows**(*cov*) x N matrix, wishart random

	matrices.
newstate	Opaque vector, the updated state.

Example

```
cov = { 1 0.5, 0.5 1 };
df = 7;
r = rndWishart(1, cov, df);
```

```
r = 7.6019339 4.7744799
    4.7744799 7.7341260
```

See Also

rndMVn, rndCreateState

round

Purpose

Round to the nearest integer.

Format

y = **round**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing the rounded elements of *x*.

Example

```
    let x = { 77.68 -14.10,
               4.73 -158.88 };
    y = round(x);
    print y;
```

```
       78.00  -14.00
        5.00 -159.00
```

See Also

trunc, floor, ceil

rows

Purpose

Returns the number of rows in a matrix.

Format

y = **rows**(*x*);

Input

x	NxK matrix or sparse matrix.

Output

y	scalar, number of rows in the specified matrix.

Remarks

If *x* is an empty matrix, **rows**(*x*) and **cols**(*x*) return 0.

Example

```
x = ones(3,5);
y = rows(x);
print x;
```

```
   1.00   1.00   1.00
   1.00   1.00   1.00
   1.00   1.00   1.00
```

```
print y;
```

```
   3.00
```

See Also

cols, show

rowsf

Purpose

Returns the number of rows in a **GAUSS** data set (`.dat`) file or **GAUSS** matrix (`.fmt`) file.

Format

y = **`rowsf`**(*f*);

Input

f	file handle of an open file.

Output

y	scalar, number of rows in the specified file.

Example

```
open fp = wilshire.dat;
r = rowsf(fp);
c = colsf(fp);
print r;
```

```
   324.00
```

```
print c;
```

```
        7.00
```

See Also

colsf, open, typef

rref

Purpose

Computes the reduced row echelon form of a matrix.

Format

y = **rref**(*x*);

Input

x	MxN matrix.

Output

y	MxN matrix containing reduced row echelon form of *x*.

Remarks

The tolerance used for zeroing elements is computed inside the procedure using:

```
      tol = maxc(m|n) * eps * maxc(abs(sumc(x')));
```

where *eps* = 2.24e-16.

This procedure can be used to find the rank of a matrix. It is not as stable numerically as the singular value decomposition (which is used in the **rank** function), but it is faster for large matrices.

There is some speed advantage in having the number of rows be greater than the number of columns, so you may want to transpose if all you care about is the rank.

The following code can be used to compute the rank of a matrix:

```
r = sumc(sumc(abs(y')) .> tol);
```

where *y* is the output from **rref**, and *tol* is the tolerance used. This finds the number of rows with any nonzero elements, which gives the rank of the matrix, disregarding numeric problems.

Example

```
// Since (row 2) = 2*(row 1), we do not expect this
// matrix to have full rank
x[3,3] = 1 2 3
         2 4 6
         3 5 2;
y = rref(x);

// compute rank of x
r = sumc(sumc(abs(rref(x)')) .> 1e-15);
print "The rank of x = " r;
```

```
   The rank of x = 2.000
```

Source

rref.src

run

Purpose

Runs a source code or compiled code program.

Format

```
run filename;
run -r filename;
```

Input

filename	literal or ^string, name of file to run.	
	-r	flag, returns control to the calling program.

Remarks

The filename can be any legal file name. Filename extensions can be whatever you want, except for the compiled file extension, `.gcg`. Pathnames are okay. If the name is to be taken from a string variable, then the name of the string variable must be preceded by the ^ (caret) operator.

The `run` statement can be used both from the command line and within a program. If used in a program, once control is given to another program through the `run` statement, there is no return to the original program unless the flag *-r* is used.

If you specify a filename without an extension, **GAUSS** will first look for a compiled code program (i.e., a `.gcg` file) by that name, then a source code program by that name. For example, if you enter

```
            run dog;
```

GAUSS will first look for the compiled code file `dog.gcg`, and run that if it finds it. If **GAUSS** cannot find `dog.gcg`, it will then look for the source code file `dog` with no extension.

If a path is specified for the file, then no additional searching will be attempted if the file is not found.

If a path is not specified, the current directory will be searched first, then each directory listed in *`src_path`*. The first instance found is run. *`src_path`* is defined in `gauss.cfg`.

`run /gauss/myprog.prg;`	No additional search will be made if the file is not found.
`run myprog.prg;`	The directories listed in *`src_path`* will be searched for `myprog.prg` if the file is not found in the current directory.

Programs can also be run by typing the filename on the OS command line when starting **GAUSS**.

Example

Example 1

```
    run myprog.prg;
```

Example 2

```
    name = "myprog.prg";
    run ^name;
```

Example 3

```
    x = rndn(3,3);
    run -r myprog.prg;
    y = inv(x);
    e = x*y;
```

In this case, **GAUSS** will execute the lines after the `run` command. If the the *-r* is omitted, the lines following the `run` command will not be executed within a program.

See Also

#include

satostrC

Purpose

Copies from one string array to another using a C language format specifier string for each element.

Format

```
y = satostrC(sa, fmt);
```

Input

sa	NxM string array.
fmt	1x1, 1xM, or Mx1 format specifier for each element copy.

Output

y	NxM formatted string array.

Source

`strfns.src`

See Also

strcombine

save

Purpose

Saves matrices, strings, or procedures to a disk file.

Format

```
save vflag path=path x, lpath=y;
save path=path x;
save x;
```

Input

vflag	version flag.	
	-v89	not supported
	-v92	supported on UNIX, Windows
	-v96	supported on all platforms
		See also **Foreign Language Interface**, Chapter 1, for details on the various versions. The default format can be specified in `gauss.cfg` by setting the *dat_fmt_version* configuration variable. If *dat_fmt_version* is not set, the default is *v96*.

path	literal or ^string, a default path to use for this and subsequent `save`'s.
x	a symbol name, the name of the file the symbol will be saved in is the same as this with the proper extension added for the type of the symbol.
lpath	literal or ^string, a local path and filename to be used for a particular symbol. This path will override the path previously set and the filename will override the name of the symbol being saved. The extension cannot be overridden.
y	the symbol to be saved to *lpath*.

Remarks

`save` can be used to save matrices, strings, procedures, and functions. Procedures and functions must be compiled and resident in memory before they can be `save`'d.

The following extensions will be given to files that are `save`'d:

matrix	`.fmt`
string	`.fst`
procedure	`.fcg`
function	`.fcg`
keyword	`.fcg`

If the **`path=`** subcommand is used with `save`, the path string will be remembered until changed in a subsequent command. This path will be used whenever none is specified. The `save` path can be overridden in any particular `save` by specifying an explicit path and filename.

Example

```
    spath = "/gauss";
    save path = ^spath x,y,z;
```

Save *x*, *y*, and *z* using `/gauss` as the path. This path will be used for the next `save` if none is specified.

```
    svp = "/gauss/data";
    save path = ^svp n, k, /gauss/quad1=quad;
```

n and *k* will be saved using `/gauss/data` as the `save` path, *quad* will be saved in `/gauss` with the name `quad1.fmt`. On platforms that use the backslash as the path separator, the double backslash is required inside double quotes to produce a backslash because it is the escape character in quoted strings. It is not required when specifying literals.

```
    save path=/procs;
```

Change `save` path to `/procs`.

```
    save path = /miscdata;
    save /data/mydata1 = x, y, hisdata = z;
```

In the above program:

x would be saved in `/data/mydata1.fmt`

y would be saved in `/miscdata/y.fmt`

z would be saved in `/miscdata/hisdata.fmt`

See Also

datasave, load, saveall, saved

saveall

Purpose

Saves the current state of the machine to a compiled file. All procedures, global matrices and strings will be saved.

Format

```
saveall fname;
```

Input

fname	literal or ^string, the path and filename of the compiled file to be created.

Remarks

The file extension will be `.gcg`.

A file will be created containing all your matrices, strings, and procedures. No main code segment will be saved. This just means it will be a `.gcg` file with no main program code (see `compile`). The rest of the contents of memory will be saved, including all global matrices, strings, functions and procedures. Local variables are not saved. This can be used inside a program to take a snapshot of the state of your global variables and procedures. To reload the compiled image, use `run` or `use`.

```
    library pgraph;
    external proc xy,logx,logy,loglog,hist;
    saveall pgraph;
```

This would create a file called `pgraph.gcg`, containing all the procedures, strings and matrices needed to run **Publication Quality Graphics** programs. Other programs could be compiled very quickly with the following statement at the top of each:

```
    use pgraph;
```

See Also

compile, run, use

saved

Purpose

Writes a matrix in memory to a **GAUSS** data set on disk.

Format

y = **saved**(*x*, *dataset*, *vnames*);

Input

x	NxK matrix to save in `.dat` file.
dataset	string, name of data set.
vnames	string or Kx1 character vector, names for the columns of the data set.

Output

y	scalar, 1 if successful, otherwise 0.

Remarks

If *dataset* is null or 0, the data set name will be `temp.dat`.

If *vnames* is a null or 0, the variable names will begin with "X" and be numbered 1-K.

If *vnames* is a string or has fewer elements than *x* has columns, it will be expanded as explained under `create`.

The output data type is double precision.

Example

```
x = rndn(100,3);
dataset = "mydata";
vnames = { height, weight, age };

if not saved(x,dataset,vnames);
   errorlog "Write error";
   end;
endif;
```

Source

`saveload.src`

See Also

loadd, writer, create

savestruct

Purpose

Saves a matrix of structures to a file on the disk.

Format

retcode = **saveStruct**(*instance*, *file_name*);

Input

instance	MxN matrix, instances of a structure.
file_name	string, name of file on disk to contain matrix of structures.

Output

retcode	scalar, 0 if successful, otherwise it will be non-zero.

Remarks

The file on the disk will be given a `.fsr` extension

Example

```
#include ds.sdf
struct DS p0;
p0 = reshape(dsCreate,2,3);
retc = saveStruct(p2, "p2");
```

savewind

Purpose

Save the current graphic panel configuration to a file. Note: This function is for use with the deprecated PQG graphics.

Library

pgraph

Format

```
err = savewind(filename);
```

Input

filename	string, name of file.

Output

err	scalar, 0 if successful, 1 if graphic panel matrix is invalid. Note that the file is written in either case.

Remarks

See the discussion on using graphics panels in **Tiled Graphic Panels**, Section 1.0.1.

Source

`pwindow.src`

See Also

loadwind

scale

Purpose

Fixes the scaling for subsequent graphs. The axes endpoints and increments are computed as a best guess based on the data passed to it. Note: This function is for use with the deprecated PQG graphics.

Library

pgraph

Format

`scale`(*x*, *y*);

Input

x	matrix, the X axis data.
y	matrix, the Y axis data.

Remarks

x and *y* must each have at least 2 elements. Only the minimum and maximum values are necessary.

This routine fixes the scaling for all subsequent graphs until **`graphset`** is called. This also clears **`xtics`** and **`ytics`** whenever it is called.

If either of the arguments is a scalar missing, the main graphics function will set the scaling for that axis using the actual data.

If an argument has 2 elements, the first will be used for the minimum and the last will be used for the maximum.

If an argument has 2 elements, and contains a missing value, that end of the axis will be scaled from the data by the main graphics function.

If you want direct control over the axes endpoints and tick marks, use **`xtics`** or **`ytics`**. If **`xtics`** or **`ytics`** have been called after **`scale`**, they will override **`scale`**.

Source

`pscale.src`

See Also

xtics, ytics, ztics, scale3d

scale3d

Purpose

Fixes the scaling for subsequent graphs. The axes endpoints and increments are computed as a best guess based on the data passed to it. Note: This function is for use with the deprecated PQG graphics.

Library

`pgraph`

Format

`scale3d`(*x*, *y*, *z*);

Input

x	matrix, the X axis data.
y	matrix, the Y axis data.
z	matrix, the Z axis data.

Remarks

x, *y* and *z* must each have at least 2 elements. Only the minimum and maximum values are necessary.

This routine fixes the scaling for all subsequent graphs until **`graphset`** is called. This also clears **`xtics`**, **`ytics`** and **`ztics`** whenever it is called.

If any of the arguments is a scalar missing, the main graphics function will set the scaling for that axis using the actual data.

If an argument has 2 elements, the first will be used for the minimum and the last will be used for the maximum.

If an argument has 2 elements, and contains a missing value, that end of the axis will be scaled from the data by the main graphics function.

If you want direct control over the axes endpoints and tick marks, use **xtics**, **ytics**, or **ztics**. If one of these functions have been called, they will override **scale3d**.

Source

```
pscale.src
```

See Also

scale, xtics, ytics, ztics

scalerr

Purpose

Tests for a scalar error code.

Format

y = **scalerr**(*c*);

Input

c	NxK matrix or sparse matrix or N-dimensional array, generally the return argument of a function or procedure call.

Output

y	scalar or [N-2]-dimensional array, 0 if the argument is not a

scalar error code, or the value of the error code as an integer if the argument is an error code.

Remarks

Error codes in **GAUSS** are NaN's (Not A Number). These are not just scalar integer values. They are special floating point encodings that the math chip recognizes as not representing a valid number. See also **`error`**.

`scalerr` can be used to test for either those error codes that are predefined in **GAUSS** or an error code that the user has defined using **`error`**.

If *c* is an N-dimensional array, *y* will be an [N-2]-dimensional array, where each element corresponds to a 2-dimensional array described by the last two dimensions of *c*. For each 2-dimensional array in *c* that does not contain a scalar error code, its corresponding element in *y* will be set to zero. For each 2-dimensional array in *c* that does contain a scalar error code, its corresponding element in *y* will be set to the value of that error code as an integer. In other words, if *c* is a 5x5x10x10 array, *y* will be a 5x5 array, in which each element corresponds to a 10x10 array in *c* and contains either a zero or the integer value of a scalar error code.

If *c* is an empty matrix, **`scalerr`** will return 65535.

Certain functions will either return an error code or terminate a program with an error message, depending on the trap state. The `trap` command is used to set the trap state. The error code that will be returned will appear to most commands as a missing value code, but the **`scalerr`** function can distinguish between missing values and error codes and will return the value of the error code.

Following are some of the functions that are affected by the trap state:

	trap 1	**trap 0**
function	error code	error message
`chol`	10	Matrix not positive definite

invpd	20	Matrix not positive definite
solpd	30	Matrix not positive definite
/	40	Matrix not positive definite (second argument not square)
	41	Matrix singular (second argument is square)
inv	50	Matrix singular

Example

```
trap 1;
cm = invpd(x);
trap 0;

if scalerr(cm);
   cm = inv(x);
endif;
```

In this example **invpd** will return a scalar error code if the matrix *x* is not positive definite. If **scalerr** returns with a nonzero value, the program will use the **inv** function, which is slower, to compute the inverse. Since the trap state has been turned off, if **inv** fails, the program will terminate with a Matrix singular error message.

See Also

error, trap, trapchk

scalinfnanmiss

Purpose

Returns true if the argument is a scalar infinity, NaN, or missing value.

Format

y = **scalinfnanmiss**(*x*);

Input

x	NxK matrix.

Output

y	scalar, 1 if *x* is a scalar, infinity, NaN, or missing value, else 0.

Example

```
//Create an infinity
x = 1/0;

if scalInfNanMiss(x);
   print "x = " x;
else;
   print "x is Not: a Nan, Infinity, or Missing";
endif;
```

See Also

isinfnanmiss, ismiss, scalmiss

scalmiss

Purpose

Tests to see if its argument is a scalar missing value.

Format

y = **`scalmiss`**(*x*);

Input

x	NxK matrix.

Output

y	scalar, 1 if argument is a scalar missing value, 0 if not.

Remarks

`scalmiss` first tests to see if the argument is a scalar. If it is not scalar, **`scalmiss`** returns a 0 without testing any of the elements.

To test whether any element of a matrix is a missing value, use **`ismiss`**. **`scalmiss`** will execute much faster if the argument is a large matrix, since it will not test each element of the matrix but will simply return a 0.

An element of *x* is considered to be a missing if and only if it contains a missing value in the real part. Thus, **`scalmiss`** and **`scalmiss`** would return a 1 for complex *x* = . + 1i, and a 0 for *x* = 1 + .i.

Example

```
    clear s;
    do until eof(fp);
       y = readr(fp,nr);
       y = packr(y);
       if scalmiss(y);
          continue;
       endif;
       s = s+sumc(y);
    endo;
```

In this example the **`packr`** function will return a scalar missing if every row of its argument contains missing values, otherwise it will return a matrix that contains no missing values. **`scalmiss`** is used here to test for a scalar missing returned from **`packr`**. If the test returns true, then the sum step will be skipped for that iteration of the read loop because there were no rows left after the rows containing missings were packed out.

schtoc

Purpose

Reduces any 2x2 blocks on the diagonal of the real Schur matrix returned from **`schur`**. The transformation matrix is also updated.

Format

{ *schc*, *transc* } = **schtoc**(*sch*, *trans*);

Input

sch	real NxN matrix in Real Schur form, i.e., upper triangular except for possibly 2x2 blocks on the diagonal.
trans	real NxN matrix, the associated transformation matrix.

Output

schc	NxN matrix, possibly complex, strictly upper triangular. The diagonal entries are the eigenvalues.
transc	NxN matrix, possibly complex, the associated transformation matrix.

Remarks

Other than checking that the inputs are strictly real matrices, no other checks are made. If the input matrix *sch* is already upper triangular, it is not changed. Small off-diagonal elements are considered to be zero. See the source code for the test used.

Example

```
    { schc, transc } = schtoc(schur(a));
```

This example calculates the complex Schur form for a real matrix *a*.

Source

```
schtoc.src
```

See Also

schur

schur

Purpose

Computes the Schur form of a square matrix.

Format

{ *s*, *z* } = **schur**(*x*)

Input

x	KxK matrix.

Output

s	KxK matrix, Schur form.
z	KxK matrix, transformation matrix.

Remarks

schur computes the real Schur form of a square matrix. The real Schur form is an upper quasi-triangular matrix, that is, it is block triangular where the blocks are 2x2 submatrices which correspond to complex eigenvalues of *x*. If *x* has no complex eigenvalues, *s* will be strictly upper triangular. To convert *s* to the complex Schur form, use the **Run-Time Library** function **schtoc**.

x is first reduced to upper Hessenberg form using orthogonal similiarity transformations, then reduced to Schur form through a sequence of QR decompositions.

schur uses the ORTRAN, ORTHES and HQR2 functions from EISPACK.

z is an orthogonal matrix that transforms *x* into *s* and vice versa. Thus

```
    s = z'*x*z;
```

and since *z* is orthogonal,

```
    x = z*s*z';
```

Example

```
    //Generate a 5 x 5 matrix of random normal numbers
    x = rndn(5, 5);
    { s, z } = schur(x);

    //From formula above in Remarks section
    newx = z*s*z';

    //Calculate the largest difference between the elements of
    //x and newx
    dif = maxc(maxc(abs(newx-x)));
    print dif;
```

```
       1.33e-14
```

See Also

hess

screen

Purpose

Controls output to the screen.

Format

```
screen on;
screen off;
screen;
```

Remarks

When this is **`on`**, the results of all print statements will be directed to the window. When this is **`off`**, print statements will not be sent to the window. This is independent of the statement **`output on`**, which will cause the results of all print statements to be routed to the current auxiliary output file.

If you are sending a lot of output to the auxiliary output file on a disk drive, turning the window off will speed things up.

The `end` statement will automatically perform **`output off`** and **`screen on`**.

`screen` with no arguments will print "Screen is on" or "Screen is off" on the console.

Example

```
    output file = mydata.asc reset;
    screen off;

    format /m1/rz 1,8;
    open fp = mydata;
    do until eof(fp);
        print readr(fp,200);;
    endo;
    fp = close(fp);
    end;
```

The program above will write the contents of the **GAUSS** file `mydata.dat` into an ASCII file called `mydata.asc`. If `mydata.asc` already exists, it will be overwritten.

Turning the window off will speed up execution. The `end` statement above will automatically perform output off and **`screen on`**.

See Also

output, end, new

searchsourcepath

Purpose

Searches the source path and (if specified) the src subdirectory of the **GAUSS** installation directory for a specified file.

Format

fpath = **searchsourcepath**(*fname*, *srcdir*);

Input

fname	string, name of file to search for.
srcdir	scalar, one of the following: *0* do not search in the `src` subdirectory of the **GAUSS** installation directory. *1* search the `src` subdirectory first. *2* search the `src` subdirectory last.

Output

fpath	string, the path of *fname*, or null string if *fname* is not found.

Remarks

The source path is set by the *src_path* configuration variable in your **GAUSS**configuration file, `gauss.cfg`.

seekr

Purpose

Moves the pointer in a `.dat` or `.fmt` file to a particular row.

Format

```
y = seekr(fh, r);
```

Input

fh	scalar, file handle of an open file.
r	scalar, the row number to which the pointer is to be moved.

Output

y	scalar, the row number to which the pointer has been moved.

Remarks

If *r* = -1, the current row number will be returned.

If *r* = 0, the pointer will be moved to the end of the file, just past the end of the last row.

rowsf returns the number of rows in a file.

```
    seekr(fh,0) == rowsf(fh) + 1;
```

Do NOT try to seek beyond the end of a file.

See Also

open, readr, rowsf

S

select (dataloop)

Purpose

Selects specific rows (observations) in a data loop based on a logical expression.

Format

```
select logical_expression;
```

Remarks

Selects only those rows for which *logical_expression* is TRUE. Any variables referenced must already exist, either as elements of the source data set, as `extern`'s, or as the result of a previous `make`, `vector`, or `code` statement.

Example

```
    select age > 40 AND sex $== 'MALE';
```

See Also

delete (dataloop)

selif

Purpose

Selects rows from a matrix. Those selected are the rows for which there is a 1 in the corresponding row of *e*.

Format

y = **selif**(*x*, *e*);

Input

x	NxK matrix or string array.
e	Nx1 vector of 1's and 0's.

Output

y	MxK matrix or string array consisting of the rows of *x* for which there is a 1 in the corresponding row of *e*.

Remarks

The argument *e* will usually be generated by a logical expression using "dot" operators.

y will be a scalar missing if no rows are selected.

Example

```
    y = selif(x,x[.,2] .gt 100);
```

This example selects all rows of *x* in which the second column is greater than 100.

```
    let x[3,3] = 0 10 20
```

```
                 30 40 50
                 60 70 80;

     e =(x[.,1] .gt 0) .and (x[.,3] .lt 100);
     y = selif(x,e);
```

The resulting matrix *y* is:

```
     30 40 50
     60 70 80
```

All rows for which the element in column 1 is greater than 0 and the element in column 3 is less than 100 are placed into the matrix *y*.

See Also

delif, scalmiss

seqa, seqm

Purpose

seqa creates an additive sequence. **seqm** creates a multiplicative sequence.

Format

y = **seqa**(*start*, *inc*, *n*);
y = **seqm**(*start*, *inc*, *n*);

Input

start	scalar specifying the first element.

inc	scalar specifying increment.
n	scalar specifying the number of elements in the sequence.

Output

y	*n*x1 vector containing the specified sequence.

Remarks

For **seqa**, *y* will contain a first element equal to *start*, the second equal to *start* + *inc*, and the last equal to *start* + *inc**(*n*-1).

For instance,

```
    seqa(1,1,10);
```

will create a column vector containing the numbers 1, 2, ...10.

For **seqm**, *y* will contain a first element equal to *start*, the second equal to *start* * *inc*, and the last equal to *start* * inc^{n-1}.

For instance,

```
    seqm(10,10,10);
```

will create a column vector containing the numbers 10, 100,...10^{10}.

Example

```
    a = seqa(2,2,10)';
    print a;
```

```
        2  4  6  8  10  12  14  16  18  20
```

```
m = seqm(2,2,10)';
print m;
```

```
        2  4  8  16  32  64  128  512  1024
```

Note that the results have been transposed in this example. Both functions return Nx1 (column) vectors.

See Also

recserar, recsercp

setarray

Purpose

Sets a contiguous subarray of an N-dimensional array.

Format

```
setarray a, loc, src;
```

Input

a	N-dimensional array.
loc	Mx1 vector of indices into the array to locate the subarray of interest, where M is a value from 1 to N.
src	[N-M]-dimensional array, matrix, or scalar.

Remarks

setarray resets the specified subarray of *a* in place, without making a copy of the entire array. Therefore, it is faster than **putarray**.

If *loc* is an Nx1 vector, then src must be a scalar. If *loc* is an [N-1]x1 vector, then *src* must be a 1-dimensional array or a 1xL vector, where L is the size of the fastest moving dimension of the array. If *loc* is an [N-2]x1 vector, then *src* must be a KxL matrix, or a KxL 2-dimensional array, where K is the size of the second fastest moving dimension.

Otherwise, if *loc* is an Mx1 vector, then *src* must be an [N-M]-dimensional array, whose dimensions are the same size as the corresponding dimensions of array *a*.

Example

```
    a = arrayalloc(2|3|4|5|6,0);
    src = arrayinit(4|5|6,5);
    loc = { 2,1 };
    setarray a,loc,src;
```

This example sets the contiguous 4x5x6 subarray of a beginning at [2,1,1,1,1] to the array *src*, in which each element is set to the specified value 5.

See Also

putarray

setdif

Purpose

Returns the unique elements in one vector that are not present in a second vector.

Format

y = **setdif**(*v1*, *v2*, *typ*);

Input

v1	Nx1 vector.
v2	Mx1 vector.
typ	scalar, type of data.
	0 character, case sensitive.
	1 numeric.
	2 character, case insensitive.

Output

y	Lx1 vector containing all unique values that are in *v1* and are not in *v2*, sorted in ascending order.

Remarks

Place smaller vector first for fastest operation.

When there are a lot of duplicates, it is faster to remove them first with unique before calling this function.

Example

```
    let v1 = mary jane linda john;
    let v2 = mary sally;
    typ = 0;
    y = setdif(v1,v2,typ);
```

Now, *y* is equal to:

```
    jane
    linda
    john
```

Source

```
setdif.src
```

See Also

setdifsa

setdifsa

Purpose

Returns the unique elements in one string vector that are not present in a second string vector.

Format

sy = **setdifsa**(*sv1*, *sv2*);

Input

sv1	Nx1 or 1xN string vector.
sv2	Mx1 or 1xM string vector.

Output

sy	Lx1 vector containing all unique values that are in *sv1* and are not in *sv2*, sorted in ascending order.

Remarks

Place smaller vector first for fastest operation.

When there are a lot of duplicates it is faster to remove them first with **unique** before calling this function.

Example

```
string sv1 = { "mary", "jane", "linda", "john" };
string sv2 = { "mary", "sally" };

sy = setdifsa(sv1,sv2);
```

Now *sy* is equal to:

```
    jane
    john
    linda
```

Source

`setdif.src`

See Also

setdif

setvars

Purpose

Reads the variable names from a data set header and creates global matrices with the same names.

Format

```
nvec = setvars(dataset);
```

Input

dataset	string, the name of the **GAUSS** data set. Do not use a file extension.

Output

nvec	Nx1 character vector, containing the variable names defined in the data set.

Remarks

setvars is designed to be used interactively.

Example

```
    nvec = setvars("freq");
```

Source

vars.src

See Also

makevars

setvwrmode

Purpose

Sets the graphics viewer mode. NOTE: This function is for use with the deprecated PQG graphics.

Library

pgraph

Format

oldmode = **setvwrmode**(*mode*);

Input

mode	string, new mode or null string. "one" Use only one viewer. "many" Use a new viewer for each graph.

Output

oldmode	string, previous *mode*.

Remarks

If *mode* is a null string, the current *mode* will be returned with no changes made.

If "*one*" is set, the viewer executable will be `vwr.exe`.

Example

```
    oldmode = setvwrmode("one");
    call setvwrmode(oldmode);
```

Source

`pgraph.src`

See Also

pqgwin

setwind

Purpose

Sets the current graphic panel to a previously created graphic panel number. NOTE: This function is for use with the deprecated PQG graphics. Use **`plotLayout`** instead.

Library

pgraph

Format

`setwind`(*n*);

Input

n	scalar, graphic panel number.

Remarks

This function selects the specified graphic panel to be the current graphic panel. This is the graphic panel in which the next graph will be drawn.

See the discussion on using graphic panels in **Graphic Panels**, Section 1.1.

Source

`pwindow.src`

See Also

begwind, endwind, getwind, nextwind, makewind, window

shell

Purpose

Executes an operating system command.

Format

```
shell stmt;
```

Input

stmt	literal or ^string, the command to be executed.

Remarks

`shell` lets you run shell commands and programs from inside **GAUSS**. If a command is specified, it is executed; when it finishes, you automatically return to **GAUSS**. If no command is specified, the shell is executed and control passes to it, so you can issue commands interactively. You have to type **`exit`** to get back to **GAUSS** in that case.

If you specify a command in a string variable, precede it with the ^ (caret) as shown in the examples below.

Example

```
    comstr = "ls ./src";
    shell ^comstr;
```

This lists the contents of the ./src subdirectory, then returns to **GAUSS**.

```
    shell cmp n1.fmt n1.fmt.old;
```

This compares the matrix file `n1.fmt` to an older version of itself, n1.fmt.old, to see if it has changed. When **`cmp`** finishes, control is returned to **GAUSS**.

```
    shell;
```

This executes an interactive shell. The OS prompt will appear and OS commands or other programs can be executed. To return to **GAUSS**, type **`exit`**.

See Also

exec

shiftr

Purpose

Shifts the rows of a matrix.

Format

y = **`shiftr`**(*x*, *s*, *f*);

Input

x	NxK matrix to be shifted.
s	scalar or Nx1 vector specifying the amount of shift.
f	scalar or Nx1 vector specifying the value to fill in.

Output

y	NxK shifted matrix.

Remarks

The shift is performed within each row of the matrix, horizontally. If the shift value is positive, the elements in the row will be moved to the right. A negative shift value causes the elements to be moved to the left. The elements that are pushed off the end of the row are lost, and the fill value will be used for the new elements on the other end.

Example

```
x = { 1 2,
      3 4 };
s = { 1,
      1 };
f = { 99,
     999 };
y = shiftr(x,s,f);
```

Now *y* is equal to:

```
    99   1
    4   999
```

```
    x = { 1 2 3,
          4 5 6,
          7 8 9 };
    s = { 0,
          1,
          2 };
    f = 0;
    y2 = shiftr(x,s,f);
```

Now *y2* is equal to:

```
    1   2   3
    0   4   5
    0   0   7
```

See Also

rotater

show

Purpose

Displays the global symbol table.

Format

```
    show -flagssymbol;
    show -flags;
    show symbol;
    show;
```

Input

flags	flags to specify the symbol type that is shown.	
	k	keywords
	p	procedures
	f	`fn` functions
	m	matrices
	s	strings
	g	show only symbols with global references
	l	show only symbols with all local references
symbol	the name of the symbol to be shown. If the last character is an asterisk (`*`), all symbols beginning with the supplied characters will be shown.	

Remarks

If there are no arguments, the entire symbol table will be displayed.

`show` is directed to the auxiliary output if it is open.

Here is an example listing with an explanation of the columns. Note that `show` does not display the column titles shown here:

```
Memory used Name Cplx Type References Info
128 bytes a MATRIX 4,4
672 bytes add KEYWORD global refs 0=1
192 bytes area FUNCTION local refs 1=1
256 bytes c C MATRIX 4,4
296 bytes p1 PROCEDURE local refs 1=1
384 bytes p2 PROCEDURE global refs 0=1
8 bytes ps1 STRUCT sdat *
16 bytes s STRING 8 char
312 bytes s1 STRUCT sdat 1,1
40 bytes sa STRING ARRAY 3,1
56 bytes sm SPARSE MATRIX 15,15
2104 bytes token PROCEDURE local refs 2=1
216 bytes y ARRAY 3 dims 2,3,4
672 bytes program space used
12 global symbols, 2000 maximum, 12 shown
0 active locals, 2000 maximum
1 active structure
```

The 'Memory used' column gives the amount of memory used by each item.

The 'Name' column gives the name of each symbol.

The 'Cplx' column contains a 'C' if the symbol is a complex matrix.

The 'Type' column specifies the type of the symbol. It can be ARRAY, FUNCTION, KEYWORD, MATRIX, PROCEDURE, STRING, STRING ARRAY, or STRUCT.

If the symbol is a procedure, keyword or function, the 'References' column will show if it makes any global references. If it makes only local references, the procedure or function can be saved to disk in an `.fcg` file with the `save` command. If the function or procedure makes any global references, it cannot be saved in an `.fcg` file.

If the symbol is a structure, the 'References' column will contain the structure type. A structure pointer is indicated by a `*` following the structure type.

The 'Info' column depends on the type of the symbol. If the symbol is a procedure or a function, it gives the number of values that the function or procedure returns and the number of arguments that need to be passed to it when it is called. If the symbol is a matrix, sparse matrix, string array or array of structures, then the 'Info' column gives the number of rows and columns. If the symbol is a string, then it gives the number of characters in the string. If the symbol is an N-dimensional array, then it gives the orders of each dimension. As follows:

Rets=Args	if procedure, keyword, or function
Row,Col	if matrix, sparse matrix, string array, or structure
Length	if string
OrdN,...,Ord2,Ord1	if array, where N is the slowest moving dimension of the array, and Ord is the order (or size) of a dimension

If the symbol is an array of structures, the 'Info' column will display the size of the array. A scalar structure instance is treated as a 1x1 array of structures. If the symbol is a structure pointer, the 'Info' column will be blank.

The program space is the area of space reserved for all nonprocedure, nonfunction program code. The maximum program space can be controlled by the `new` command.

The maximum number of global and local symbols is controlled by the *`maxglobals`* and *`maxlocals`* configuration variables in `gauss.cfg`.

Example

```
show -fpg eig*;
```

This command will show all functions and procedures that have global references and begin with **`eig`**.

```
show -m;
```

This command will show all matrices.

See Also

new, delete

sin

Purpose

Returns the sine of its argument.

Format

y = **`sin`**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing the sine of *x*.

Remarks

For real data, *x* should contain angles measured in radians.

To convert degrees to radians, multiply the degrees by *π/180*.

Example

```
    let x = { 0, .5, 1, 1.5 };
    y = sin(x);
    print y;
```

```
        0.000000
        0.479426
        0.841471
        0.997495
```

See Also

atan, cos, sinh, pi

singleindex

Purpose

Converts a vector of indices for an N-dimensional array to a scalar vector index.

Format

si = **singleindex**(*i*, *o*);

Input

i	Nx1 vector of indices into an N-dimensional array.
o	Nx1 vector of orders of an N-dimensional array.

Output

si	scalar, index of corresponding element in 1-dimensional array or vector.

Remarks

This function and its opposite, **arrayindex**, allow you to convert between an N-dimensional index and its corresponding location in a 1-dimensional object of the same size.

Example

```
orders = { 2,3,4 };

a = arrayalloc(orders,0);
ai = { 2,1,3 };
setarray a, ai, 49;
v = vecr(a);
vi = singleindex(ai,orders);

print "ai = " ai;
```

```
    print "vi = " vi;
    print "getarray(a,ai) = " getarray(a,ai);
    print "v[vi] = " v[vi];
```

produces:

```
    ai =
       2.0000000
       1.0000000
       3.0000000
    vi = 15.000000
    getarray(a,ai) = 49.000000
    v[vi] = 49.000000
```

This example allocates a 3-dimensional array *a* and sets the element corresponding to the index vector ai to 49. It then creates a vector, *v*, with the same data. The element in the array *a* that is indexed by *ai* corresponds to the element of the vector *v* that is indexed by *vi*.

See Also

arrayindex

sinh

Purpose

Computes the hyperbolic sine.

Format

y = **sinh**(*x*);

Input

x	NxK matrix.

Output

y	NxK matrix containing the hyperbolic sines of the elements of *x*.

Example

```
    let x = { -0.5, -0.25, 0, 0.25, 0.5, 1 };
    x = x * pi;
    y = sinh(x);
```

The above statement produces, *y* equal to:

```
    -2.301299
    -0.868671
     0.000000
     0.868671
     2.301299
    11.548739
```

Source

`trig.src`

sleep

Purpose

Sleeps for a specified number of seconds.

Format

unslept = **sleep**(*secs*);

Input

secs	scalar, number of seconds to sleep.

Output

unslept	scalar, number of seconds not slept.

Remarks

secs does not have to be an integer. If your system does not permit sleeping for a fractional number of seconds, *secs* will be rounded to the nearest integer, with a minimum value of 1.

If a program sleeps for the full number of *secs* specified, **sleep** returns 0; otherwise, if the program is awakened early (e.g., by a signal), **sleep** returns the amount of time not slept.

A program may sleep for longer than *secs* seconds, due to system scheduling.

solpd

Purpose

Solves a set of positive definite linear equations.

Format

`x = solpd(b, A);`

Input

b	NxK matrix or M-dimensional array where the last two dimensions are NxK.
A	NxN symmetric positive definite matrix or M-dimensional array where the NxN 2-dimensional arrays described by the last two dimensions are symmetric and positive definite.

Output

x	NxK matrix or M-dimensional array where the last two dimensions are NxK, the solutions for the system of equations, $Ax = b$.

Remarks

b can have more than one column. If so, the system of equations is solved for each column, i.e., *A***x*[., *i*] = *b*[., *i*].

This function uses the Cholesky decomposition to solve the system directly. Therefore it is more efficient than using **inv**(*A*)**b*.

If *b* and *A* are M-dimensional arrays, the sizes of their corresponding M-2 leading dimensions must be the same. The resulting array will contain the solutions for the system of equations given by each of the corresponding 2-dimensional arrays described by the two trailing dimensions of *b* and *A*. In other words, for a 10x4x2 array *b* and a 10x4x4 array *A*, the resulting array *x* will contain the solutions for each of the 10 corresponding 4x2 arrays contained in *b* and 4x4 arrays contained in *A*. Therefore, *A*[*n*,.,.]**x*[*n*,.,.] = *b*[*n*,.,.], for $1 \leq n \leq 10$.

solpd does not check to see that the matrix *A* is symmetric. **solpd** will look only at the upper half of the matrix including the principal diagonal.

If the *A* matrix is not positive definite:

trap 1	return scalar error code 30.
trap 0	terminate with an error message.

One obvious use for this function is to solve for least squares coefficients. The effect of this function is thus similar to that of the **/** operator.

If *X* is a matrix of independent variables, and *Y* is a vector containing the dependent variable, then the following code will compute the least squares coefficients of the regression of *Y* on *X*:

```
b = solpd(X'Y,X'X);
```

Example

```
n = 5;
format /lo 16,8;
```

```
    A = rndn(n,n);
    A = A'A;
    x = rndn(n,1);
    b = A*x;

    x2 = solpd(b,A);

    print " X solpd(b,A) Difference";
    print x~x2~x-x2;
```

produces:

```
    X solpd(b,A)  Difference

     0.32547881   0.32547881  -4.9960036e-16
     1.5190182    1.5190182   -1.7763568e-15
     0.88099266   0.88099266   1.5543122e-15
     1.8192784    1.8192784   -2.2204460e-16
    -0.060848175 -0.060848175 -1.4710455e-15
```

See Also

chol, invpd, trap

sortc, sortcc

Purpose

Sorts a matrix of numeric or character data.

Format

y = **sortc**(*x*, *c*);
y = **sortcc**(*x*, *c*);

Input

x	NxK matrix.
c	scalar specifying one column of *x* to sort on.

Output

y	NxK matrix equal to *x* and sorted on the column *c*.

Remarks

These functions will sort the rows of a matrix with respect to a specified column. That is, they will sort the elements of a column and will arrange all rows of the matrix in the same order as the sorted column.

sortc assumes that the column to sort on is numeric. **sortcc** assumes that the column to sort on contains character data.

The matrix may contain both character and numeric data, but the sort column must be all of one type. Missing values will sort as if their value is below -∞.

The sort will be in ascending order. This function uses the Quicksort algorithm.

If you need to obtain the matrix sorted in descending order, you can use:

```
    rev(sortc(x, c))
```

Example

```
    let x[3,3]= 4 7 3
```

```
                 1  3  2
                 3  4  8;
     y = sortc(x,1);
```

The above example code produces, *y* equal to:

```
     1  3  2
     3  4  8
     4  7  3
```

See Also

rev

sortd

Purpose

Sorts a data file on disk with respect to a specified variable.

Format

```
sortd(infile, outfile, keyvar, keytyp);
```

Input

infile	string, name of input file.
outfile	string, name of output file, must be different.
keyvar	string, name of key variable.

keytyp	scalar, type of key variable.	
	1	numeric key, ascending order.
	2	character key, ascending order.
	-1	numeric key, descending order.
	-2	character key, descending order.

Remarks

The data set `infile` will be sorted on the variable `keyvar`, and will be placed in `outfile`.

If the inputs are null ("" or 0), the procedure will ask for them.

Source

`sortd.src`

See Also

sortmc, sortc, sortcc, sorthc, sorthcc

sorthc, sorthcc

Purpose

Sorts a matrix of numeric or character data, or a string array.

Format

```
y = sorthc(x, c);
y = sorthcc(x, c);
```

Input

x	NxK matrix or string array.
c	scalar specifying one column of *x* to sort on.

Output

y	NxK matrix or string array equal to *x* and sorted on the column *c*.

Remarks

These functions will sort the rows of a matrix or string array with respect to a specified column. That is, they will sort the elements of a column and will arrange all rows of the object in the same order as the sorted column.

sorthc assumes that the column to sort on is numeric. **sorthcc** assumes that the column to sort on contains character data.

If *x* is a matrix, it may contain both character and numeric data, but the sort column must be all of one type. Missing values will sort as if their value is below $-\infty$.

The sort is in ascending order. This function uses the heap sort algorithm.

If you need to obtain the matrix sorted in descending order, you can use:

```
    rev(sorthc(x, c))
```

Example

```
let x[3,3]= 4 7 3
            1 3 2
            3 4 8;

//Sort x based upon the values in the third column
y = sorthc(x,3);
```

This produces *y* equal to:

```
1 3 2
4 7 3
3 4 8
```

See Also

sortc, rev

sortind, sortindc

Purpose

Returns the sorted index of *x*.

Format

ind = **sortind**(*x*);
ind = **sortindc**(*x*);

Input

x	Nx1 column vector.

Output

ind	Nx1 vector representing sorted index of *x*.

Remarks

sortind assumes that *x* contains numeric data. **sortindc** assumes that *x* contains character data.

This function can be used to sort several matrices in the same way that some other reference matrix is sorted. To do this, create the index of the reference matrix, then use **submat** to rearrange the other matrices in the same way.

Example

```
//Create uniform random integers between 0 and 10
x = round(10*rndu(10, 1));

ind = sortind(x);
y = x[ind];
```

After running the above code:

```
            9.00
            8.00
x    =      0.00
            4.00
            6.00
```

```
        3.00
        4.00
ind  =  5.00
        2.00
        1.00

        0.00
        4.00
y    =  6.00
        8.00
        9.00
```

sortmc

Purpose

Sorts a matrix on multiple columns.

Format

y = **sortmc**(*x*,   *v*);

Input

x	NxK matrix to be sorted.
v	Lx1 vector containing integers specifying the columns, in order, that are to be sorted. If an element is negative, that column will be interpreted as character data.

Output

y	NxK sorted matrix.

Example

sortmc keeps all rows together. After it sorts on the first specified column, it will continue to sort the rows of the matrix using the other specified columns ONLY when there is a tie in the first column. For example:

```
x = { 9 2 5 6,
      3 6 1 9,
      3 7 4 1,
      1 2 8 9 };

s1 = sortc(x,1);

sm = sortmc(x, 1|2);
```

will return:

```
          1       2       8       9
s1  =     3       7       4       1
          3       6       1       9
          9       2       5       6

          1       2       8       9
sm  =     3       6       1       9
          3       7       4       1
          9       2       5       6
```

In the output above, we see that the difference between *s1* and *sm* is that the second and third rows have been switched. This is because **sortmc** first sorted the matrix based upon row one like **sortc**. Then **sortmc** sorted the rows in which the first column was the same (in our example they are both threes), based upon the values in the second column.

Source

```
sortmc.src
```

See Also

sortd, sortc, sortcc, sorthc, sorthcc

sortr, sortrc

Purpose

Sorts the columns of a matrix of numeric or character data, with respect to a specified row.

Format

y = **`sortr`**(*x*, *r*);
y = **`sortrc`**(*x*, *r*);

Input

x	NxK matrix.
r	scalar, row of *x* on which to sort.

Output

y	NxK matrix equal to *x* and sorted on row *r*.

Remarks

These functions sort the columns of a matrix with respect to a specified row. That is, they sort the elements of a row and arrange all rows of the matrix in the same order as the sorted column.

sortr assumes the row on which to sort is numeric. **sortrc** assumes that the row on which to sort contains character data.

The matrix may contain both character and numeric data, but the sort row must be all of one type. Missing values will sort as if their value is below $-\infty$.

The sort will be in left to right ascending order. This function uses the Quicksort algorithm. If you need to obtain the matrix sorted left to right in descending order (i.e., ascending right to left), use:

```
rev(sortr(x, r)')'
```

Example

```
//Create a 5 x 3 matrix of random integers
//between 1 and 30
x = ceil(30*rndu(5, 3));

//Sort the columns based upon the first row
y = sortr(x,1);
```

Examine the variables after the code above. Notice that the columns remain the same, but their order has changed.

```
          10.000 21.000 18.000
          11.000 30.000 20.000
     x =  10.000 23.000  7.000
           6.000  9.000 20.000
           7.000  4.000 30.000

          10.000 18.000 21.000
          11.000 20.000 30.000
     y =  10.000  7.000 23.000
           6.000 20.000  9.000
           7.000 30.000  4.000
```

If we were to use the same *x*, but sort on the 5th row:

```
     y2 = sortr(x, 5);
```

We get the following result:

```
           21.000 10.000 18.000
           30.000 11.000 20.000
     y2 =  23.000 10.000  7.000
            9.000  6.000 20.000
            4.000  7.000 30.000
```

spBiconjGradSol

Purpose

Attempts to solve the system of linear equations $Ax = b$ using the biconjugate gradient method where A is a sparse matrix.

Format

```
x = spBiconjGradSol(a, b, epsilon, maxit);
```

Input

a	NxN, sparse matrix.
b	Nx1, dense vector.
epsilon	Method tolerance: If epsilon is set to 0, the default tolerance is set to 1e-6.
maxit	Maximum number of iterations. If maxit is set to 0, the default setting is 300 iterations.

Output

x	Nx1 dense vector.

Example

```
      nz = { 33.446  82.641 -12.710 -25.062   0.000,
              0.000 -26.386  17.016  21.576 -45.273,
              0.000 -42.331 -47.902   0.000   0.000,
              0.000 -26.517 -22.135 -76.827  31.920,
             10.364 -29.843 -20.277   0.000  65.816 };
      b = { 10.349,
            -3.117,
             4.240,
             0.013,
             2.115 };
      sparse matrix a;
      a = densetosp(nz,0);

      //Setting the third and fourth arguments to 0 employs the
      //default tolerance and maxit settings
      x = spBiconjGradSol(a,b,0,0);

      //Solve the system of equations using the '/' operator for
      //comparison
      x2 = b/a;
```

The output from the above code:

```
            0.135
            0.055
     x =   -0.137
            0.018
           -0.006

            0.135
            0.055
     x2 =  -0.137
            0.018
           -0.006
```

Remarks

If convergence is not reached within the maximum number of iterations allowed, the function will either terminate the program with an error message or return an error code which can be tested for with the **scalerr** function. This depends on the trap state as follows:

trap 1	return error code: 60
trap 0	terminate with error message: Unable to converge in allowed number of iterations.

If matrix A is not well conditioned use the `/` operator to perform the solve. If the matrix is symmetric, **`spConjGradSol`** will be approximately twice as fast as **`spBiconjGradSol`**.

See Also

spConjGradSol

spChol

Purpose

Computes the LL' decomposition of a sparse matrix A.

Format

`l` = **`spChol`**(*`a`*);

Input

`a`	NxN, symmetric, positive definite sparse matrix.

Output

`l`	NxN lower-triangular sparse matrix.

Example

```
sparse matrix A;
sparse matrix L;

//Create a small, simple positive-definite matrix
let x = { 9.53984224e+001 -5.84272701e+000 1.99970335e+001,
         -5.84272701e+000  1.09765831e+002 2.52038945e+000,
          1.99970335e+001  2.52038945e+000 4.71834812e+000
};

//Create the sparse matrix A from x, keeping all elements
A = denseToSp(x, 0);

//Create matrix factorization
L = spChol(A);
```

See Also

spLDL, spLU

Technical Notes

spChol implements functions from the TAUCS library: TAUCS Version 2.2. Copyright ©2001, 2002, 2003 by Sivan Toledo, Tel-Aviv University, stoledo@tau.ac.il. All Rights Reserved.

spConjGradSol

Purpose

Attempts to solve the system of linear equations $Ax = b$ using the conjugate gradient method where A is a symmetric sparse matrix.

Format

x = **`spConjGradSol`**(*`a`*, *`b`*, *`epsilon`*, *`maxit`*);

Input

`a`	NxN, symmetric sparse matrix.
`b`	Nx1, dense vector.
`epsilon`	Method tolerance: If epsilon is set to 0, the default tolerance is set to 1e-6.
`maxit`	Maximum number of iterations. If maxit is set to 0, the default setting is 300 iterations.

Output

`x`	Nx1 dense vector

Example

```
nz = {    0.000    2845.607      0.000      0.000      0.000,
       2845.607   10911.430      0.000      0.000      0.000,
        0.000       0.000  3646.798  2736.338 -2674.440,
        0.000       0.000  2736.338  7041.526 -3758.528,
        0.000       0.000 -2674.440 -3758.528  7457.899 };
 sparse matrix a;

//Set 'a' to be a sparse matrix with the same contents as
//the dense matrix 'nz'
a = densetosp(nz,0);

//Create our right-hand-side
b = { 10.349,
      -3.117,
       4.240,
       0.013,
       2.115 };

//Setting the third and fourth arguments to 0 employs the
//default tolerance maxit settings
x = spConjGradSol(a,b,0,0);

newb = a*x;
```

The results from the above code are:

```
           -0.01504075
            0.00363683
     x   =  0.00203504
           -0.00033936
            0.00084234

           10.34900000
           -3.11700000
     newb = 4.24000000
```

```
          0.01300000
          2.11500000
```

Remarks

If convergence is not reached within the maximum number of iterations allowed, the function will either terminate the program with an error message or return an error code which can be tested for with the **`scalerr`** function. This depends on the trap state as follows:

trap 1	return error code: 60
trap 0	terminate with error message: Unable to converge in allowed number of iterations.

If matrix A is not symmetric or well conditioned use the **`/`** operator to perform the solve. For a nonsymmetric, but well conditioned matrix A, use **`spBiconjGradSol`**.

See Also

spBiconjGradSol

spCreate

Purpose

Creates a sparse matrix from vectors of non-zero values, row indices, and column indices.

Format

y = **`spCreate`**(*`r`*, *`c`*, *`vals`*, *`rinds`*, *`cinds`*);

Input

r	scalar, rows of output matrix.
c	scalar, columns of output matrix.
vals	Nx1 vector, non-zero values.
rinds	Nx1 vector, row indices of corresponding non-zero values.
cinds	Nx1 vector, column indices of corresponding non-zero values.

Output

y	*r* x *c* sparse matrix.

Remarks

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **`spCreate`**.

Example

```
    //Declare 'y' to be a sparse matrix
    sparse matrix y;

    //Create the non-zero values to place in the sparse matrix
    vals = { 1.7, 2.4, 3.2, 4.5 };

    //Set the row and column indices for the location in which
```

```
//to place each successive element of 'vals' into the new
//matrix
rinds = { 2,5,8,13 };
cinds = { 4,1,9,5 };

y = spCreate(15,10,vals,rinds,cinds);
```

This example creates a 15x10 sparse matrix *y*, containing the following non-zero values:

Non-zero value	Index
1.7	(2,4)
2.4	(5,1)
3.2	(8,9)
4.5	(13,5)

See Also

packedToSp, denseToSp, spEye

spDenseSubmat

Purpose

Returns a dense submatrix of a sparse matrix.

Format

y = **spDenseSubmat**(*x*, *rinds*, *cinds*);

Input

x	MxN sparse matrix.
rinds	Kx1 vector, row indices.
cinds	Lx1 vector, column indices.

Output

y	KxL dense matrix, the intersection of *rinds* and *cinds*.

Remarks

If *rinds* or *cinds* are scalar zeros, all rows or columns will be returned.

Example

```
sparse matrix y;
x = { 0  0  0 10,
      0  2  0  0,
      0  0  0  0,
      5  0  0  0,
      0  0  0  3 };

//Set 'y' to be a sparse matrix with the same values as 'x'
y = denseToSp(x,0);

//Extract a submatrix from 'y' with all rows of 'y' and
//columns 1, 3 and 4
d = spDenseSubmat(y,0,1|3|4);
```

Now *d* is equal to:

```
     0   0 10
     0   0  0
     0   0  0
     5   0  0
     0   0  3
```

See Also

spSubmat

spDiagRvMat

Purpose

Inserts submatrices along the diagonal of a sparse matrix.

Format

y = **spDiagRvMat**(*x*, *inds*, *size*, *a*);

Input

x	MxN sparse matrix.
inds	Kx2 vector or scalar 0, row and column indices into *x* at which to place the corresponding submatrices in *a*.
size	Kx2 vector or scalar 0, sizes of the corresponding submatrices in *a*.
a	KxLxP array, containing the submatrices to insert into *x*.

Output

y	MxN sparse matrix, a copy of *x* containing the specified insertions.

Remarks

Each row of *inds* must contain the row and column indices, respectively, that form the starting point for the insertion of the corresponding submatrix in *a*. If *inds* is a scalar 0, the starting point for the insertion of each submatrix will be one row and one column past the ending point of the previous insertion. The first insertion will begin at the [1,1] element.

Each row of *size* must contain the number of rows and columns in the corresponding submatrix in *a*. This allows you to insert submatrices of different sizes L_ixP_i by inserting them into the planes of an array that is KxMAX(L)xMAX(P) and padding the submatrices with zeros to MAX(L)xMAX(P). For each plane in *a*, **spDiagRvMat** extracts the submatrix *a*[*i*,1:*size*[*i*,1], 1:*size*[*i*,2]] and inserts that into *x* at the location indicated by the corresponding row of *inds*. If *size* is a scalar 0, then each LxP plane of *a* is inserted into *x* as is.

Example

```
declare sparse matrix x,y;

//Create a 10x10 sparse identity matrix
x = spEye(10);

sx1 = { 2 3, 5 8 };
sx2 = { 8 2 3 4, 7 9 5 6, 3 2 8 4 };
sx3 = { 4 7 2, 6 5 3 };
```

```
sx4 = { 9, 3 };

//Create a 4x3x4 dimensional array with every element set
//to 0
a = arrayinit(4|3|4,0);

//Set some of the array values
a[1,1:2,1:2] = sx1;
a[2,.,.] = sx2;
a[3,1:2,1:3] = sx3;
a[4,1:2,1] = sx4;
```

The value of *a* is now:

```
Plane [1,.,.]

    2.00000000    3.00000000    0.00000000    0.00000000
    5.00000000    8.00000000    0.00000000    0.00000000
    0.00000000    0.00000000    0.00000000    0.00000000

Plane [2,.,.]

    8.00000000    2.00000000    3.00000000    4.00000000
    7.00000000    9.00000000    5.00000000    6.00000000
    3.00000000    2.00000000    8.00000000    4.00000000

Plane [3,.,.]

    4.00000000    7.00000000    2.00000000    0.00000000
    6.00000000    5.00000000    3.00000000    0.00000000
    0.00000000    0.00000000    0.00000000    0.00000000

Plane [4,.,.]

    9.00000000    0.00000000    0.00000000    0.00000000
    3.00000000     0.00000000     0.00000000     0.00000000
    0.00000000     0.00000000     0.00000000     0.00000000
```

```
inds = 0;
siz = { 2 2, 3 4, 2 3, 2 1 };

y = spDiagRvMat(x,inds,siz,a);
```

The output, in variable *y*, is:

```
2   3   0   0   0   0   0   0   0   0
5   8   0   0   0   0   0   0   0   0
0   0   8   2   3   4   0   0   0   0
0   0   7   9   5   6   0   0   0   0
0   0   3   2   8   4   0   0   0   0
0   0   0   0   0   1   4   7   2   0
0   0   0   0   0   0   6   5   3   0
0   0   0   0   0   0   0   1   0   9
0   0   0   0   0   0   0   0   1   3
0   0   0   0   0   0   0   0   0   1
```

spEigv

Purpose

Computes a specified number of eigenvalues and eigenvectors of a square, sparse matrix *a*.

Format

{ *va*, *ve* } = **spEigv**(*a*, *nev*, *which*, *tol*, *maxit*, *ncv*);

Input

a	NxN square, sparse matrix.
nev	Scalar, number of eigenvalues to compute.
which	String, may be one of the following: "LM" largest magnitude, "LR" largest real, "LI" largest imaginary, "SR" smallest real, or "SI" smallest imaginary. Default input 0, sets *which* to "LM."
tol	Scalar, tolerance for eigenvalues. Default input 0, sets *tol* to 1e-15.
maxit	Scalar, maximum number of iterations. Default input 0, sets *maxit* to *nev*x(columns of *a*)x100.
ncv	Scalar, size of Arnoldi factorization. The minimum setting is the greater of *nev*+2 and 20. See Remarks on how to set *ncv*. Default input 0, sets *ncv* to 2x *nev*+1.

Output

va	*nev*x1 dense vector containing the computed eigenvalues of input matrix *a*.
ve	Nx *nev* dense matrix containing the corresponding eigenvectors of input matrix *a*.

Example

```
    rndseed 3456;
```

```
      sparse matrix a;
      x = 10*rndn(5,5);
      a = densetosp(x,4);
```

```
        21.276135  5.4078872 -19.817044  9.6771132 -19.211952
        0.0000000 -4.4011007  10.445221 -5.1742289 -16.336474
    a = 0.0000000 -20.853017  7.6285434  0.0000000 -15.626397
       -12.637055  8.1227002  0.0000000 -8.7817892  0.0000000
        0.0000000 -7.8181517  15.326816  0.0000000  0.0000000
```

```
     { va, ve } = spEigv(a,2,0,0,0,0);
     /* equivalent to call { va, ve } = spEigv(a,2,"LM",1e-15,2
     *5*100,5); */
```

```
     va = 21.089832
         -3.4769986 + 20.141970i

     ve = -0.92097057   0.29490584 - 0.38519280i
          -0.10091920  -0.18070330 - 0.38405816i
          0.061241324   0.24121182 - 0.56419722i
           0.36217049  0.017643612 + 0.26254313i
          0.081917964  -0.31466284 - 0.19936942i
```

Below we show that the first eigenvalue times the corresponding eigenvector (1) equals the input matrix times the first eigenvector (2).

```
     (1) va[1]*ve[.,1]     =      (2) a*ve[.,1] =
      -19.423115                  -19.423115
      -2.1283690                  -2.1283690
       1.2915693                   1.2915693
       7.6381149                   7.6381149
       1.7276361                   1.7276361
```

Remarks

The ideal setting for input *ncv* is problem dependent and cannot be easily predicted ahead of time. Increasing *ncv* will increase the amount of memory used during computation. For a large, sparse matrix, *ncv* should be small compared to the order of input matrix *a*. **spEigv** is *not* thread-safe.

Technical Notes

spEigv implements functions from the ARPACK library.

spEye

Purpose

Creates a sparse identity matrix.

Format

y = **spEye**(*n*);

Input

n	scalar, order of identity matrix.

Output

y	*n* x *n* sparse identity matrix.

Remarks

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spEye**.

Example

```
    //Declare 'y' a sparse matrix
    sparse matrix y;

    //Create 3x3 sparse identity matrix
    y = spEye(3);
```

y is now equal to:

```
    1   0   1
    0   1   0
    0   0   1
```

See Also

spCreate, spOnes, denseToSp

spGetNZE

Purpose

Returns the non-zero values in a sparse matrix, as well as their corresponding row and column indices.

Format

{ *vals*, *rowinds*, *colinds* } = **spNumNZE**(*x*);

Input

x	MxN sparse matrix.

Output

vals	Nx1 vector, non-zero values in *x*.
rinds	Nx1 vector, row indices of corresponding non-zero values.
cinds	Nx1 vector, column indices of corresponding non-zero values.

Example

```
sparse matrix y;
x = { 0 0 0 10,
      0 2 0  0,
      0 0 0  0,
      5 0 0  0,
      0 0 0  3 };

//Create sparse matrix from 'x'
y = denseToSp(x,0);

//Get non-zero values, row indices and column indices
{ v,r,c } = spGetNZE(y);
```

v, the non-zero values, is equal to:

```
     10
      2
      5
      3
```

r, the row indices, is equal to:

```
      1
      2
      4
      5
```

c, the column indices, is equal to:

```
      4
      2
      1
      4
```

See Also

spNumNZE

spline

Purpose

Computes a two-dimensional interpolatory spline.

Format

{ *u*, *v*, *w* } = **spline**(*x*, *y*, *z*, *sigma*, *g*);

Input

x	1xK vector, x-abscissae (x-axis values).
y	Nx1 vector, y-abscissae (y-axis values).
z	KxN matrix, ordinates (z-axis values).
sigma	scalar, tension factor.
g	scalar, grid size factor.

Output

u	1x(K**g*) vector, x-abscissae, regularly spaced.
v	(N**g*)x1 vector, y-abscissae, regularly spaced.
w	(K**g*)x(N**g*) matrix, interpolated ordinates.

Remarks

sigma contains the tension factor. This value indicates the curviness desired. If *sigma* is nearly zero (e.g., .001), the resulting surface is approximately the tensor product of cubic splines. If *sigma* is large (e.g., 50.0), the resulting surface is approximately bi-linear. If *sigma* equals zero, tensor products of cubic splines result. A standard value for *sigma* is approximately 1.

g is the grid size factor. It determines the fineness of the output grid. For *g* = 1, the output matrices are identical to the input matrices. For *g* = 2, the output grid is twice as fine as the input grid, i.e., *u* will have twice as many columns as *x*, *v* will have twice as many rows as *y*, and *w* will have twice as many rows and columns as *z*.

Source

`spline.src`

spLDL

Purpose

Computes the LDL decomposition of a symmetric sparse matrix A.

Format

{ *l*, *d* } = **spLDL**(*a*);

Input

a	N x N, symmetric sparse matrix.

Output

l	NxN lower-triangular sparse matrix.
d	NxN diagonal sparse matrix.

Example

```
declare sparse matrix a, l, d;
nz = { 142 13 56 57 0,
        13   0   0   0 0,
        56   0 94 47 0,
        57   0 47 35 0,
         0   0   0   0 0 };

a = densetosp(nz,0);
{ l, d } = spLDL(a);
```

Remarks

spLDL will not check to see if the input matrix is symmetric. The function looks only at the lower triangular portion of the input matrix.

See Also

spLU

Technical Notes

spLDL implements functions from the TAUCS library:

TAUCS Version 2.2 Copyright ©2003, by Sivan Toledo, Tel-Aviv University, stoledo@tau.ac.il. All Rights Reserved.

spLU

Purpose

Computes the LU decomposition of a sparse matrix A with partial pivoting.

Format

{ *l*, *u* } = **spLU**(*a*);

Input

a	N x N, non-singular sparse matrix.

Output

l	NxN "scrambled" lower-triangular sparse matrix. This is a lower triangular matrix that has been reordered based upon the row pivoting.
u	NxN "scrambled" upper-triangular sparse matrix. This is an upper triangular matrix that has been reordered based upon column pivoting to preserve sparsity.

Example

```
declare sparse matrix a, l, u;

nz = {-5.974 0 -13.37 6.136                0,
           0   5.932   7.712       0  -6.549,
           0  -5.728       0  14.227       0,
           0 -12.164   9.916  13.902   6.182,
      13.425       0 -12.654 -16.534       0 };

a = densetosp(nz,0);
{ l, u } = spLU(a);
```

Remarks

If the input matrix or either of the factors L and U are singular, the function will either terminate the program with an error message or return an error code which can be tested for with the `scalerr` function. This depends on the trap state as follows:

trap 1	return error code: 50
trap 0	terminate with error message: Matrix singular

See Also

spLDL

Technical Notes

spLU implements functions from the SuperLU 4.0 library written by James W. Demmel, John R. Gilbert and Xiaoye S. Li.

Copyright ©2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy). All rights reserved.

spNumNZE

Purpose

Returns the number of non-zero elements in a sparse matrix.

Format

n = **spNumNZE**(*x*);

Input

x	MxN sparse matrix.

Output

n	scalar, the number of non-zero elements in *x*.

Example

```
     sparse matrix y;
     x = { 0  0  0 10,
           0  2  0   0,
           0  0  0   0,
           5  0  0   0,
           0  0  0   3 };

      y = denseToSp(x,0);
      n = spNumNZE(y);
      print "The number of nonzeros is" n;
```

```
        4.00
```

See Also

spGetNZE

spOnes

Purpose

Generates a sparse matrix containing only ones and zeros

Format

y = **spOnes**(*r*, *c*, *rinds*, *cinds*);

Input

r	scalar, rows of output matrix.
c	scalar, columns of output matrix.
rinds	Nx1 vector, row indices of ones.
cinds	Nx1 vector, column indices of ones.

Output

y	*r* x *c* sparse matrix of ones.

Remarks

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spOnes**.

Example

```
    //declare sparse matrix
    sparse matrix y;

    //Set row indices and column indices
    rinds = { 1, 3, 5 };
    cinds = { 2, 1, 3 };

    //Create a 5x4 sparse matrix with ones at the intersection
    //of the 'rind' and 'cind'
    y = spOnes(5,4,rinds,cinds);
```

The resulting *y* is equal to:

```
    0  1  0  0
    0  0  0  0
    1  0  0  0
    0  0  0  0
    0  0  1  0
```

See Also

spCreate, spEye, spZeros, denseToSp

SpreadsheetReadM

Purpose

Reads and writes Excel files.

Format

xlsmat = **SpreadsheetReadM**(*file*, *range*, *sheet*);

Input

file	string, name of `.xls` file.
range	string, range to read or write; e.g., "a1:b20".
sheet	scalar, sheet number.

Output

xlsmat	matrix of numbers read from Excel.

Portability

Windows, **Linux** and **Mac**

Remarks

If the read functions fail, they will return a scalar error code which can be decoded with **`scalerr`**. If the write function fails, it returns a non-zero error number.

See Also

scalerr, error, SpreadsheetReadSA, SpreadsheetWrite

SpreadsheetReadSA

Purpose

Reads and writes Excel files.

Format

```
xlssa = SpreadsheetReadSA(file, range, sheet);
```

Input

file	string, name of `.xls` file.
range	string, range to read or write; e.g., "a1:b20".
sheet	scalar, sheet number.

Output

xlssa	string array read from Excel.

Portability

Windows, **Linux** and **Mac**

Remarks

If the read functions fail, they will return a scalar error code which can be decoded with **`scalerr`**. If the write function fails, it returns a non-zero error number.

See Also

scalerr, error, SpreadsheetReadM, SpreadsheetWrite

S

SpreadsheetWrite

Purpose

Reads and writes Excel files.

Format

```
xlsret = SpreadsheetWrite(data, file, range, sheet);
```

Input

data	matrix, string or string array, data to write.
file	string, name of `.xls` file.
range	string, range to read or write; e.g., "a1:b20".
sheet	scalar, sheet number.

Output

xlsret	success code, 0 if successful, else error code.

Portability

Windows, **Linux** and **Mac**

Remarks

If the read functions fail, they will return a scalar error code which can be decoded with **`scalerr`**. If the write function fails, it returns a non-zero error number.

See Also

scalerr, error, SpreadsheetReadM, SpreadsheetReadSA

spScale

Purpose

Scales a sparse matrix.

Format

{ *a*, *r*, *s* } = **`spScale`**(*x*);

Input

x	MxN sparse matrix.

Output

a	MxN scaled sparse matrix.
r	Mx1 vector, row scale factors.
s	Nx1 vector, column scale factors.

Remarks

spScale scales the elements of the matrix by powers of 10 so that they are all within (-10,10).

Example

```
    x = { 25  -12    0,
           3    0  -11,
           8 -100    0 };

    declare sparse matrix sm, smsc;
    sm = denseToSp(x,0);

    { smsc, r, c } = spScale(sm);
```

The results:

```
             2.50  -0.12   0.00
    smsc =   0.30   0.00  -0.11
             0.80  -1.00   0.00

             1.00
    c =      0.10
             0.10

             0.10
    r =      0.10
             0.10
```

spSubmat

Purpose

Returns a sparse submatrix of a sparse matrix.

Format

y = **`spSubmat`**(*x*, *`rinds`*, *`cinds`*);

Input

x	MxN sparse matrix.
`rinds`	Kx1 vector, row indices.
`cinds`	Lx1 vector, column indices.

Output

s	KxL sparse matrix, the intersection of *`rinds`* and *`cinds`*.

Remarks

If *`rinds`* or *`cinds`* are scalar zeros, all rows or columns will be returned.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **`spSubmat`**.

Example

```
    sparse matrix y;
    sparse matrix z;

    x = { 0 0 0 10,
          0 2 0 0,
          0 0 0 0,
          5 0 0 0,
          0 0 0 3 };

    y = denseToSp(x,0);

    //Extract all columns; rows 1, 3 and 4
    z = spSubmat(y,1|3|4,0);

    //Extract all values from 'z' into a dense matrix 'd'
    d = spDenseSubmat(z,0,0);
```

Now *d* is equal to:

```
    0.00    0.00    0.00   10.00
    0.00    0.00    0.00    0.00
    5.00    0.00    0.00    0.00
```

See Also

spDenseSubmat

spToDense

Purpose

Converts a sparse matrix to a dense matrix.

Format

y = **spToDense**(*x*);

Input

x	MxN sparse matrix.

Output

y	MxN dense matrix.

Remarks

A dense matrix is just a normal format matrix.

Example

```
sparse matrix y;

//Create a 4x4 sparse identity matrix
y = spEye(4);

//Create a dense matrix with the same values as 'y'
d = spToDense(y);
```

The dense matrix *d* is equal to:

```
1   0   0   0
0   1   0   0
```

```
    0   0   1   0
    0   0   0   1
```

See Also

spDenseSubmat, denseToSp

spTrTDense

Purpose

Multiplies a sparse matrix transposed by a dense matrix.

Format

y = **spTrTDense**(*s*, *d*);

Input

s	NxM sparse matrix.
d	NxL dense matrix.

Output

y	MxL dense matrix, the result of *s*'**d*.

Remarks

This may also be accomplished by the following code:

```
    y = s'*d;
```

However, **spTrTDense** will be more efficient.

See Also

spTScalar

spTScalar

Purpose

Multiplies a sparse matrix by a scalar.

Format

y = **spTScalar**(*s*, *scal*, *rinds*, *cinds*);

Input

s	NxM sparse matrix.
scal	scalar.
rinds	Kx1 vector of row indices.
cinds	Lx1 vector of column indices.

Output

y	KxL sparse matrix.

Remarks

Only the elements of *s* specified by *rinds* and *cinds* will be multiplied by *scal*. All other elements will be unchanged in the result.

To select all rows or all columns, input a scalar 0 for *rinds* or *cinds*.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spTScalar**.

Example

```
sparse matrix y;
x = { 3 0 2 1,
      0 4 0 0,
      5 0 0 3,
      0 1 2 0 };

rinds = 0;
cinds = { 2,4 };

//Multiply all elements in the second and fourth column
//by 'scal'
y = spTScalar(x,10,rinds,cinds);
d = spDenseSubmat(y,0,0);
```

The result, in *d* is:

```
3 0  2 1
0 40 0 0
```

```
     5  0   0  3
     0  10  2  0
```

See Also

spTrTDense

spZeros

Purpose

Creates a sparse matrix containing no non-zero values.

Format

y = **spZeros**(*r*, *c*);

Input

r	scalar, rows of output matrix.
c	scalar, columns of output matrix.

Output

y	*r* x *c* sparse matrix.

Remarks

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spZeros**.

Example

```
sparse matrix y;

//Create a 4x3 sparse matrix with all elements set to 0
y = spZeros(4,3);

//Create a dense matrix with the same values as 'y'
d = spToDense(y);
```

The contents of *d* are equal to:

```
0  0  0
0  0  0
0  0  0
0  0  0
```

See Also

spOnes, spEye

sqpSolve

Purpose

Solves the nonlinear programming problem using a sequential quadratic programming method.

Format

```
{ x, f, lagr, retcode } = sqpSolve(&fct, start);
```

Input

&fct	pointer to a procedure that computes the function to be minimized. This procedure must have one input argument, a vector of parameter values, and one output argument, the value of the function evaluated at the input vector of parameter values.
start	Kx1 vector of start values.

Global Input

_sqp_A	MxK matrix, linear equality constraint coefficients.
_sqp_B	Mx1 vector, linear equality constraint constants.

These globals are used to specify linear equality constraints of the following type:

$$\_sqp\_A * x = \_sqp\_B$$

where x is the Kx1 unknown parameter vector.

_sqp_EqProc scalar, pointer to a procedure that computes the nonlinear equality constraints. For example, the statement:

```
_sqp_EqProc = &eqproc;
```

tells **sqpSolve** that nonlinear equality constraints are to be placed on the parameters and where the procedure computing them is to be found. The procedure must have one input argument, the Kx1 vector of parameters, and one output argument, the Rx1 vector of computed constraints that are to be equal to zero. For example, suppose that you wish to place the following constraint:

```
    p[1] * p[2] = p[3]
```

The procedure for this is:

```
    proc eqproc(p);
      retp(p[1]*p[2]-p[3]);
    endp;
```

_sqp_C	MxK matrix, linear inequality constraint coefficients.
_sqp_D	Mx1 vector, linear inequality constraint constants.

These globals are used to specify linear inequality constraints of the following type:

```
    _sqp_C * X >= _sqp_D
```

where x is the Kx1 unknown parameter vector.

_sqp_IneqProc scalar, pointer to a procedure that computes the nonlinear inequality constraints. For example the statement:

```
    _sqp_EqProc = &ineqproc;
```

tells **sqpSolve** that nonlinear equality constraints are to be placed on the parameters and where the procedure computing them is to be found. The procedure must have one input argument, the Kx1 vector of parameters, and one output argument, the Rx1 vector of computed constraints that are to be equal to zero. For example, suppose that you wish to place the following constraint:

```
    p[1] * p[2] >= p[3]
```

The procedure for this is:

```
    proc ineqproc(p);
      retp(p[1]*[2]-p[3]);
    endp;
```

_sqp_Bounds

Kx2 matrix, bounds on parameters. The first column contains the lower bounds, and the second column the upper bounds. If the bounds for all the coefficients are the same, a 1x2 matrix may be used. Default is:

[1] -1e256   [2] 1e256

_sqp_GradProc

scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. For example, the statement:

```
    _sqp_GradProc = &gradproc;
```

tells **sqpSolve** that a gradient procedure exists and where to find it. The user-provided procedure has two input arguments, a Kx1 vector of parameter values and an NxP matrix of data. The procedure returns a single output argument, an NxK matrix of gradients of the log-likelihood function with respect to the parameters evaluated at the vector of parameter values.

Default = 0, i.e., no gradient procedure has been provided.

_sqp_HessProc

scalar, pointer to a procedure that computes the Hessian, i.e.,

	the matrix of second order partial derivatives of the function with respect to the parameters. For example, the instruction:
	`_sqp_HessProc = &hessproc;`
	will tell **sqpSolve** that a procedure has been provided for the computation of the Hessian and where to find it. The procedure that is provided by the user must have two input arguments, a Px1 vector of parameter values and an NxK data matrix. The procedure returns a single output argument, the PxP symmetric matrix of second order derivatives of the function evaluated at the parameter values.
_sqp_MaxIters	scalar, maximum number of iterations. Default = 1e+5. Termination can be forced by pressing C on the keyboard.
_sqp_DirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisifed, **sqpSolve** will exit the iterations.
_sqp_ParNames	Kx1 character vector, parameter names.
sqp PrintIters	scalar, if nonzero, prints iteration information. Default = 0. Can be toggled during iterations by pressing P on the keyboard.
sqp FeasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off.
sqp RandRadius	scalar, if zero, no random search is attempted. If nonzero it is the radius of random search which is invoked whenever the usual line search fails. Default = .01.
__output	scalar, if nonzero, results are printed. Default = 0.

Output

x	Kx1 vector of parameters at minimum.
f	scalar, function evaluated at *x*.
lagr	vector, created using **vput**. Contains the Lagrangean for the constraints. They may be extracted with the **vread** command using the following strings:
	"lineq" — Lagrangeans of linear equality constraints,
	"nlineq" — Lagrangeans of nonlinear equality constraints
	"linineq" — Lagrangeans of linear inequality constraints
	"nlinineq" — Lagrangeans of nonlinear inequality constraints
	"bounds" — Lagrangeans of bounds
	Whenever a constraint is active, its associated Lagrangean will be nonzero.
retcode	return code:
	0 — normal convergence
	1 — forced exit
	2 — maximum number of iterations exceeded

3	function calculation failed
4	gradient calculation failed
5	Hessian calculation failed
6	line search failed
7	error with constraints

Remarks

Pressing `C` on the keyboard will terminate iterations, and pressing `P` will toggle iteration output.

sqpSolve is recursive, that is, it can call itself with another function and set of global variables,

Example

```
//Reset all sqpSolve global variables
sqpSolveSet;

proc fct(x);
  retp( (x[1] + 3*x[2] + x[3])^2 + 4*(x[1] - x[2])^2);
endp;

proc ineqp(x);
 retp(6*x[2] + 4*x[3] - x[1]^3 - 3);
endp;

proc eqp(x);
 retp(1-sumc(x));
endp;
```

```
    _sqp_Bounds = { 0 1e256 };

    start = { .1, .7, .2 };

    _sqp_IneqProc = &ineqp;
    _sqp_EqProc = &eqp;

    { x,f,lagr,ret } = sqpSolve(&fct,start);
```

Source

sqpsolve.src

sqpSolveMT

Purpose

Solves the nonlinear programming problem.

Include

sqpsolvemt.sdf

Format

out1 = **sqpSolveMT**(&*fct*, *par1*, *data1*, *c1*);

Input

&fct	pointer to a procedure that computes the function to be minimized. This procedure must have two input arguments, an instance of structure of type **PV** and an instance of a structure of type **DS**, and one output argument, either a 1x1 scalar or an Nx1 vector of function values evaluated at the parameters stored in the **PV** instance using data stored in the **DS** instance.
par1	an instance of structure of type **PV**. The *par1* instance is passed to the user-provided procedure pointed to by &*fct*. *par1* is constructed using the "**pack**" functions.
data1	an array of instances of a **DS** structure. This array is passed to the user-provided pointed by &*fct* to be used in the objective function. **sqpSolveMT** does not look at this structure. Each instance contains the the following members which can be set in whatever way that is convenient for computing the objective function:

data1 [i].dataMatrix	NxK matrix, data matrix.
data1 [i].dataArray	NxKxL.. array, data array.
data1[i].vnames	string array, variable names (optional).
data1[i].dsname	string, data name (optional).
data1[i].type	scalar, type of data (optional).

c1	an instance of an **sqpSolveMTControl** structure. Normally an

instance is initialized by calling **`sqpSolveMTControlCreate`** and members of this instance can be set to other values by the user. For an instance named *`c1`*, the members are:

`c1.A`	MxK matrix, linear equality constraint coefficients: *`c1`*.A * p = *`c1`*.B where p is a vector of the parameters.
`c1.B`	Mx1 vector, linear equality constraint constants: *`c1`*.A * *`p`* = *`c1`*.B where *`p`* is a vector of the parameters.
`c1.C`	MxK matrix, linear inequality constraint coefficients: *`c1`*.C * p >=*`c1`*.D where p is a vector of the parameters.
`c1.D`	Mx1 vector, linear inequality constraint constants: *`c1`*.C * *`p`*>=*`c1`*.D where *`p`* is a vector of the parameters.
`c1.eqProc`	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has one input argument, a structure of type SQPdata, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = ., i.e., no equality procedure.
`c1.weights`	vector, weights for objective function returning a vector. Default = 1.

`c1.ineqProc`	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has one input argument, a structure of type SQPdata, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = ., i.e., no inequality procedure.
`c1.bounds`	1x2 or Kx2 matrix, bounds on parameters. If 1x2 all parameters have same bounds. Default = -1e256 1e256 .
`c1.covType`	scalar, if 2, QML covariance matrix, else if 0, no covariance matrix is computed, else ML covariance matrix is computed.
`c1.gradProc`	scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. Default = ., i.e., no gradient procedure has been provided.
`c1.hessProc`	scalar, pointer to a procedure that computes the Hessian, i.e., the matrix of second order partial derivatives of the function with respect to the parameters. Default = ., i.e., no Hessian procedure has been provided.
`c1.maxIters`	scalar, maximum number of iterations. Default = 1e+5.
`c1.dirTol`	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5.

	When this criterion has been satisfied SQPSolve exits the iterations.
c1.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
c1.randRadius	scalar, If zero, no random search is attempted. If nonzero, it is the radius of random search which is invoked whenever the usual line search fails. Default = .01.
c1.output	scalar, if nonzero, results are printed. Default = 0.
c1.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out1 an instance of an **sqpSolveMTout** structure. For an instance named *out1*, the members are:

out1x.par	an instance of structure of type PV containing the parameter estimates will be placed in the member matrix *out1*.par.
out1.fct	scalar, function evaluated at x.
out1.lagr	an instance of a SQPLagrange structure

containing the Lagrangeans for the constraints. The members are:

`out1.lagr.lineq`	Mx1 vector, Lagrangeans of linear equality constraints.
`out1.lagr.nlineq`	Nx1 vector, Lagrangeans of nonlinear equality constraints.
`out1.lagr.linineq`	Px1 vector, Lagrangeans of linear inequality constraints.
`out1.lagr.nlinineq`	Qx1 vector, Lagrangeans of nonlinear inequality constraints.
`out1.lagr.bounds`	Kx2 matrix, Lagrangeans of bounds.

Whenever a constraint is active, its associated Lagrangean will be nonzero. For any constraint that is inactive throughout the iterations as well as at convergence, the corresponding Lagrangean matrix will be set to a scalar missing value.

`out1.retcode` return code:

0	normal convergence.
1	forced exit.
2	maximum number of iterations exceeded.

3	function calculation failed.
4	gradient calculation failed.
5	Hessian calculation failed.
6	line search failed.
7	error with constraints.
8	function complex.

Remarks

There is one required user-provided procedure, the one computing the objective function to be minimized, and four other optional functions, one each for computing the equality constraints, the inequality constraints, the gradient of the objective function, and the Hessian of the objective function.

All of these functions have one input argument that is an instance of a structure of type struct **PV** and a second argument that is an instance of a structure of type struct **DS**. On input to the call to **`sqpSolveMT`**, the first argument contains starting values for the parameters and the second argument any required data. The data are passed in a separate argument because the structure in the first argument will be copied as it is passed through procedure calls which would be very costly if it contained large data matrices. Since **`sqpSolveMT`** makes no changes to the second argument it will be passed by pointer thus saving time because its contents aren't copied.

Both of the structures of type **PV** are set up using the **PV** "**`pack`**" procedures, **`pvPack`**, **`pvPackm`**, **`pvPacks`**, and **`pvPacksm`**. These procedures allow for setting up a parameter vector in a variety of ways.

For example, we might have the following objective function for fitting a nonlinear curve to data:

```
    proc Micherlitz(struct PV par1, struct DS data1);
       local p0,e,s2,x,y;
       p0 = pvUnpack(par1, "parameters");
       y = data1.dataMatrix[.,1];
       x = data1.dataMatrix[.,2];
       e = y - p0[1] - p0[2]*exp(-p0[3] * x);
      retp(e'*e);
    endp;
```

In this example the dependent and independent variables are passed to the procedure as the first and second columns of a data matrix stored in a single **DS** structure. Alternatively these two columns of data can be entered into a vector of **DS** structures, one for each column of data:

```
    proc Micherlitz(struct PV par1, struct DS data1);
       local p0,e,s2,x,y;
       p0 = pvUnpack(par1, "parameters");
       y = data1[1].dataMatrix;
       x = data1[2].dataMatrix;
       e = y - p0[1] - p0[2]*exp(-p0[3]*x);
      retp(e'*e);
    endp;
```

The syntax is similar for the optional user-provided procedures. For example, to constrain the squared sum of the first two parameters to be greater than one in the above problem, provide the following procedure:

```
    proc ineqConst(struct PV par1, struct DS data1);
       local p0;
       p0 = pvUnpack(p0, "parameters");
      retp( (p0[2]+p0[1])^2 - 1);
    endp;
```

The following is a complete example for estimating the parameters of the Micherlitz equation in data with bounds constraints on the parameters and where an optional gradient procedure has been provided:

```
#include sqpSolveMT.sdf
struct DS d0;
d0 = dsCreate;

y =  3.183|
     3.059|
     2.871|
     2.622|
     2.541|
     2.184|
     2.110|
     2.075|
     2.018|
     1.903|
     1.770|
     1.762|
     1.550;

x = seqa(1,1,13);
d0.dataMatrix = y~x;

//Declare control structure
struct sqpSolveMTControl c0;

//Initialize structure to default values
c0 = sqpSolveMTControlCreate;

//Constrain parameters to be positive
c0.bounds = 0~100;

struct PV par1;
```

```
par1 = pvCreate;
par1 = pvPack(par1,.92|2.62|.114, "parameters");
struct sqpSolveMTout out1;
out1 = sqpSolveMT(&Micherlitz,par1,d0,c0);

print " parameter estimates ";
print pvUnPack(out1.par, "parameters");

proc Micherlitz(struct PV par1, struct DS data1);
   local p0,e,s2,x,y;
   p0 = pvUnpack(par1, "parameters");
   y = data1.dataMatrix[.,1];
   x = data1.dataMatrix[.,2];
   e = y - p0[1] - p0[2]*exp(-p0[3] * x);
  retp(e'*e);
endp;

proc grad(struct PV par1, struct DS data1);
   local p0,e,w,g,r,x,y;
   p0 = pvUnpack(par1, "parameters");
   y = data1.dataMatrix[.,1];
   x = data1.dataMatrix[.,2];
   g = zeros(3,1);
   w = exp(-p0[3] * x);
   e = y - p0[1] - p0[2]*w;
   r = e'*w;
   g[1] = -2*sumc(e);
   g[2] = -2*r;
   g[3] = 2*p0[1]*p0[2]*r;
  retp(g);
endp;
```

Source

sqpsolvemt.src

See Also

sqpSolveMTControlCreate, sqpSolveMTlagrangeCreate, CR-sqpSolveMToutCreate

sqpSolveMTControlCreate

Purpose

Creates an instance of a structure of type **sqpSolveMTcontrol** set to default values.

Include

sqpsolvemt.sdf

Format

s = **sqpSolveMTControlCreate**();

Output

s	instance of structure of type **sqpSolveMTControl**.

Example

```
//Declare instance of structure
struct sqpSolveMTControl s;

//Initialize the structure to default values
```

```
    s = sqpSolveMTControlCreate();
```

Source

sqpsolvemt.src

See Also

sqpSolve

sqpSolveMTlagrangeCreate

Purpose

Creates an instance of a structure of type **sqpSolveMTlagrange** set to default values.

Include

sqpsolvemt.sdf

Format

```
    s = sqpSolveMTlagrangeCreate();
```

Output

s	instance of structure of type **sqpSolveMTlagrange**.

Example

```
//Declare instance of structure
struct sqpSolveMTlagrange sla;

//Initialize the structure to default values
sla = sqpSolveMTlagrangeCreate();
```

Source

`sqpsolvemt.src`

See Also

sqpSolve

sqpSolveMToutCreate

Purpose

Creates an instance of a structure of type **sqpSolveMTout** set to default values.

Include

`sqpsolvemt.sdf`

Format

s = **`sqpSolveMToutCreate`**();

Output

s	instance of structure of type **sqpSolveMTout**.

Example

```
//Declare instance of structure
struct sqpSolveMTout out;

//Initialize the structure to default values
out = sqpSolveMToutCreate();
```

Source

sqpsolvemt.src

See Also

sqpSolve

sqpSolveSet

Purpose

Resets global variables used by **sqpSolve** to default values.

Format

```
sqpSolveSet;
```

Source

`sqpsolve.src`

sqrt

Purpose

Computes the square root of every element in *x*.

Format

y = **`sqrt`**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array, the square roots of each element of *x*.

Remarks

If *x* is negative, complex results are returned by default. You can turn the generation of complex numbers for negative inputs on or off in the **GAUSS** configuration file, and with the **`sysstate`** function, case 8. If you turn it off, **`sqrt`** will generate an error for negative inputs.

If *x* is already complex, the complex number state does not matter; **`sqrt`** will compute a complex result.

Example

```
let x[2,2] = 1 2 3 4;
y = sqrt(x);
```

The output, in variable *y* is equal to:

```
1.00000000
1.41421356
1.73205081
2.00000000
```

stdc

Purpose

Computes the standard deviation of the elements in each column of a matrix.

Format

y = **`stdc`**(*x*);

Input

x	NxK matrix.

Output

y	Kx1 vector, the standard deviation of each column of *x*.

Remarks

This function essentially computes:

```
sqrt(1/(N-1)*sumc((x-meanc(x)')^2))
```

Thus, the divisor is N-1 rather than N, where N is the number of elements being summed. To convert to the alternate definition, multiply by

```
sqrt((N-1)/N)
```

Example

```
//Set the rng seed so that the random numbers produced will
//be repeatable
rndseed 94243524;

//Create a vector of random normal numbers
y = rndn(8100,1);

//Compute the standard deviation of the column vector 'y'
std = stdc(y);
```

The standard deviation, in variable *std*, is equal to:

```
1.00183907
```

See Also

meanc

stdsc

Purpose

Computes the standard deviation of the elements in each column of a matrix.

Format

y = **stdsc**(*x*);

Input

x	NxK matrix.

Output

y	Kx1 vector, the standard deviation of each column of *x*.

Remarks

This function essentially computes:

```
sqrt(1/(N)*sumc((x-meanc(x)')^2))
```

Thus, the divisor is N rather than N-1, where N is the number of elements being summed. See **stdc** for the alternate definition.

Example

```
    //Create 3 columns of random normal numbers
    y = rndn(8100,3);

    //Calculate the standard deviation of each column
    std = stdsc(y);
```

The return, in variable *std* is equal to:

```
    1.00095980
    0.99488832
    1.00201375
```

See Also

stdc, astds, meanc

stocv

Purpose

Converts a string to a character vector.

Format

v = **stocv**(*s*);

Input

s	string, to be converted to character vector.

Output

v	Nx1 character vector, contains the contents of *s*.

Remarks

stocv breaks *s* up into a vector of 8-character length matrix elements. Note that the character information in the vector is not guaranteed to be null-terminated.

Example

```
      s = "Now is the time for all good men";
      v = stocv(s);
```

```
          "Now is t"

          "the time "
     v =
          "for all "

          "good men"
```

See Also

cvtos, vget, vlist, vput, vread

stof

Purpose

Converts a string to floating point.

Format

```
y = stof(x);
```

Input

x	string or NxK matrix containing character elements to be converted.

Output

y	matrix, the floating point equivalents of the ASCII numbers in *x*.

Remarks

If *x* is a string containing "1 2 3", then **`stof`** will return a 3x1 matrix containing the numbers 1, 2 and 3.

If *x* is a null string, **`stof`** will return a 0.

This uses the same input conversion routine as `loadm` and `let`. It will convert character elements and missing values. **`stof`** also converts complex numbers in the same manner as `let`.

See Also

ftos, ftocv, chrs

stop

Purpose

Stops a program and returns to the command prompt. Does not close files.

Format

```
stop;
```

Remarks

This command has the same effect as `end`, except it does not close files or the auxiliary output.

It is not necessary to put a `stop` or an `end` statement at the end of a program. If neither is found, an implicit `stop` is executed.

See Also

end, new, system

strcombine

Purpose

Converts an NxM string array to an Nx1 string vector by combining each element in a column separated by a user-defined delimiter string.

Format

y = **strcombine**(*sa*, *delim*, *qchar*);

Input

sa	NxM string array.
delim	1x1, 1xM, or Mx1 delimiter string.
qchar	scalar, 2x1, or 1x2 string vector containing quote characters as required:

scalar:	Use this character as quote character.
	If this is 0, no quotes are added.
2x1 or 1x2 string vector:	Contains left and right quote characters.

Output

y	Nx1 string vector result.

Example

```
//Create strings with directory names
projDir = "myProject";
homeDir = "C:";
gaussDir = "gauss";

//Horizonally concatenate the 2 strings into a 1 x 3 string
//array
projDir = homeDir$~gaussDir$~projDir;

//Reshape projDir from a 1 x 3 string array into a 2 x 3
```

```
    //string array
    projDir = reshape(projDir, 2, 3);

    //Create 2 x 1 string array with the names of the final
    //directory, using vertical concatenation. Then add them
    //onto the end of projDir
    endDir = "data"$|"src";
    projDir = projDir$~endDir;

    //Convert the 2 x 4 string array into a 2 x 1 array with
    //each column combined and separated by backslashes
    projDir = strcombine(projDir, "\\", 0);
    print projDir;
```

The above example will give the following output:

```
    projDir = C:\gauss\myProject\data\
    C:\gauss\myProject\src\
```

Source

strfns.src

See Also

satostrC

strindx

Purpose

Finds the index of one string within another string.

Format

```
y = strindx(where, what, start);
```

Input

where	string or scalar, the data to be searched.
what	string or scalar, the substring to be searched for in *where*.
start	scalar, the starting point of the search in *where* for an occurrence of *what*. The index of the first character in a string is 1.

Output

y	scalar containing the index of the first occurrence of *what*, within *where*, which is greater than or equal to *start*. If no occurrence is found, it will be 0.

Remarks

An example of the use of this function is the location of a name within a string of names:

```
    z = "nameagepaysex";
    x = "pay";
    y = strindx(z,x,1);
```

The above code will set *y* equal to:

```
    8.00
```

This function is used with **`strsect`** for extracting substrings.

See Also

strrindx, strlen, strsect, strput

strlen

Purpose

Returns the length of a string.

Format

y = **`strlen`**(*x*);

Input

x	string, NxK matrix of character data, or NxK string array.

Output

y	scalar containing the exact length of the string *x*, or NxK matrix or string array containing the lengths of the elements in *x*.

Remarks

The null character (ASCII 0) is a legal character within strings and so embedded nulls will be counted in the length of strings. The final terminating null byte is not counted, though.

For character matrices, the length is computed by counting the characters (maximum of 8) up to the first null in each element of the matrix. The null character, therefore, is not a valid character in matrices containing character data and is not counted in the lengths of the elements of those matrices.

Example

```
    x1 = "How long?";
    x2 = "Classification";
    len1 = strlen(x1);
    len2 = strlen(x2);
```

After running the code above:

```
    len1 = 9

    len2 = 14
```

See Also

strsect, strindx, strrindx

strput

Purpose

Lays a substring over a string.

Format

y = **strput**(*substr*, *str*, *off*);

Input

substr	string, the substring to be laid over the other string.
str	string, the string to receive the substring.
off	scalar, the offset in *str* to place *substr*. The offset of the first byte is 1.

Output

y	string, the new string.

Example

```
str = "max";
sub = "imum";
loc = 4;
y = strput(sub,str,loc);
print y;
```

produces:

```
maximum
```

Source

`strput.src`

strrindx

Purpose

Finds the index of one string within another string. Searches from the end of the string to the beginning.

Format

y = **`strrindx`**(*`where`*, *`what`*, *`start`*);

Input

`where`	string or scalar, the data to be searched.
`what`	string or scalar, the substring to be searched for in *`where`*.
`start`	scalar, the starting point of the search in *`where`* for an occurrence of *`what`*. *`where`* will be searched from this point backward for *`what`*.

Output

y	scalar containing the index of the last occurrence of *what*, within *where*, which is less than or equal to *start*. If no occurrence is found, it will be 0.

Remarks

A negative value for *start* causes the search to begin at the end of the string. An example of the use of **strrindx** is extracting a file name from a complete path specification:

```
path = "/gauss/src/ols.src";
ps = "/";
pos = strrindx(path,ps,-1);
if pos;
   name = strsect(path,pos+1,strlen(path)-pos);
else;
   name = "";
endif;
```

The above code makes the following assignments:

```
pos = 11

name = ols.src
```

See Also

strindx, strlen, strsect, strput

strsect

Purpose

Extracts a substring of a string.

Format

y = **strsect**(*str*, *start*, *len*);

Input

str	string or scalar from which the segment is to be obtained.
start	scalar, the index of the substring in *str*. The index of the first character is 1.
len	scalar, the length of the substring.

Output

y	string, the extracted substring, or a null string if *start* is greater than the length of *str*.

Remarks

If there are not enough characters in a string for the defined substring to be extracted, then a short string or a null string will be returned.

If *str* is a matrix containing character data, it must be scalar.

Example

```
strng = "This is an example string.";
y = strsect(strng,12,7);
```

The above code assigns the variable *y* to be:

```
example
```

See Also

strlen, strindx, strrindx

strsplit

Purpose

Splits an Nx1 string vector into an NxK string array of the individual tokens.

Format

```
sa = strsplit(sv);
```

Input

sv	Nx1 string array.

Output

sa	NxK string array.

Remarks

Each row of *sv* must contain the same number of tokens. The following characters are considered delimiters between tokens:

space	ASCII 32
tab	ASCII 9
comma	ASCII 44
newline	ASCII 10
carriage return	ASCII 13

Tokens containing delimiters must be enclosed in single or double quotes or parentheses. Tokens enclosed in single or double quotes will NOT retain the quotes upon translation. Tokens enclosed in parentheses WILL retain the parentheses after translation. Parentheses cannot be nested.

Example

```
    //Create a 2x1 string array
    let string sv = {
        "kingdom phylum class",
        "order family genus" };

    //Split 'sv' into a string array in which each token is an
    //element in a new string array
    sa = strsplit(sv);

    //Print the [1,1] element of 'sa' followed by the [1,2],
    //[1,3], [2,1]...
    for i(1, 2, 1);
       for j(1, 3, 1);
```

```
        print sa[i,j];
    endfor;
endfor;
```

The above code sets *sa* to be equal to:

```
"kingdom" "phylum" "class" "order" "family" "genus"
```

and produces the output:

```
kingdom
phylum
class
order
family
genus
```

Elements that contain spaces may be grouped with single tics, like this:

```
let string ss = { "classification 'scientific
    taxonomy'" };
ss2 = strsplit(ss);

print "ss2[1] = " ss2[1];
print "ss2[2] = " ss2[2];
```

In this program, 'scientific taxonomy' is kept as one token, and thus the output from the above code is:

```
ss2[1] = classification
ss2[2] = scientific taxonomy
```

See Also

strsplitPad

strsplitPad

Purpose

Splits a string vector into a string array of the individual tokens. Pads on the right with null strings.

Format

```
sa = strsplitPad(sv, cols);
```

Input

sv	Nx1 string array.
cols	scalar, number of columns of output string array.

Output

sa	Nx *cols* string array.

Remarks

Rows containing more than *cols* tokens are truncated and rows containing fewer than *cols* tokens are padded on the right with null strings. The following characters are considered delimiters between tokens:

space	ASCII 32
tab	ASCII 9

comma	ASCII 44
newline	ASCII 10
carriage return	ASCII 13

Tokens containing delimiters must be enclosed in single or double quotes or parentheses. Tokens enclosed in single or double quotes will NOT retain the quotes upon translation. Tokens enclosed in parentheses WILL retain the parentheses after translation. Parentheses cannot be nested.

Example

```
let string sv = {
   "alpha beta gamma",
   "delta, epsilon, zeta, eta",
   "theta iota kappa"
};

 sa = strsplitPad(sv, 4);
```

After the code above, *sa* will be equal to:

```
"alpha""beta""gamma""""delta""epsilon""zeta""eta""the
ta""iota""kappa"""
```

See Also

strsplit

strtodt

Purpose

Converts a string array of dates to a matrix in DT scalar format.

Format

x = **strtodt**(*sa*, *fmt*);

Input

sa	NxK string array containing dates.
fmt	string containing date/time format characters.

Output

x	NxK matrix of dates in DT scalar format.

Remarks

The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number:

```
20120921223505
```

represents 22:35:05 or 10:35:05 PM on September 21, 2012.

The following formats are supported:

YYYY	Four digit year
YR	Last two digits of year
MO	Number of month, 01-12
DD	Day of month, 01-31
HH	Hour of day, 00-23
MI	Minute of hour, 00-59
SS	Second of minute, 00-59

Example

```
    x = strtodt("2012-07-12 10:18:32", "YYYY-MO-DD HH:MI:
    SS");
    print x;
```

produces:

```
    20120712101832.0
```

```
    x = strtodt("2012-07-12 10:18:32", "YYYY-MO-DD");
    print x;
```

produces:

```
    20120712000000.0
```

```
    x = strtodt("10:18:32", "HH:MI:SS");
    print x;
```

produces:

```
    101832.0
    x = strtodt("05-28-10", "MO-DD-YR");
    print x;
```

produces:

```
    20100528000000.0
```

See Also

dttostr, dttoutc, utctodt

strtof

Purpose

Converts a string array to a numeric matrix.

Format

x = **strtof**(*sa*);

Input

sa	NxK string array containing numeric data.

Output

x	NxK matrix.

Remarks

Elements with more than one numerical character separated by a delimiter such as a comma or a space will be interpreted as complex data. For example, the string:

```
    "1.2 1.9"
```

will be converted into the number:

```
    1.2 + 1.9i
```

Parentheses surrounding the numerical elements in the string will be ignored as will be a following *i*. The following strings will be interpreted as the same by **strtof**.

```
    "(2.31 4.72)""2.31 4.73""2.31,4.73i"
```

Example

```
    //Create a string array
    string sa = { "1.1""2.2""3.3", "4.4""5.5""6.6" };
    num = strtof(sa);
```

After the code above, *num* is a numeric matrix with the following values:

```
    1.100  2.200 3.300
    4.400  5.500 6.600
```

See Also

strtofcplx, ftostrC

strtofcplx

Purpose

Converts a string array to a complex numeric matrix.

Format

x = **strtofcplx**(*sa*);

Input

sa	NxK string array containing numeric data.

Output

x	NxK complex matrix.

Remarks

strtofcplx supports both real and complex data. It is slower than **strtof** for real matrices. **strtofcplx** requires the presence of the real part. The imaginary part can be absent.

See Also

strtof, ftostrC

strtriml

Purpose

Strips all whitespace characters from the left side of each element in a string array.

Format

y = **strtriml**(*sa*);

Input

sa	NxM string array.

Output

y	NxM string array.

Source

`strfns.src`

See Also

strtrimr, strtrunc, strtruncl, strtruncpad, strtruncr

strtrimr

Purpose

Strips all whitespace characters from the right side of each element in a string array.

Format

y = **strtrimr**(*sa*);

Input

sa	NxM string array.

Output

y	NxM string array.

Source

`strfns.src`

See Also

strtriml, strtrunc, strtruncl, strtruncpad, strtruncr

strtrunc

Purpose

Truncates all elements of a string array to not longer than the specified number of characters.

Format

y = **strtrunc**(*sa*, *maxlen*);

Input

sa	NxK string array.
maxlen	1xK or 1x1 matrix, maximum length.

Output

y	NxK string array result.

Example

```
string s = { "best", "linear", "unbiased", "estimator"
};
ss = strtrunc(s, 6);
```

After the code above, the variables **s** and **ss** are equal to:

```
      best
    linear
    s = unbiased
       estimator

            best
          linear
    ss =  unbias
          estima
```

See Also

strtriml, strtrimr, strtruncl, strtruncpad, strtruncr

strtruncl

Purpose

Truncates the left side of all elements of a string array by a user-specified number of characters.

Format

y = **strtruncl**(*sa*, *ntrunc*);

Input

sa	NxM, Nx1, 1xM, or 1x1 string array.
ntrunc	NxM, Nx1, 1xM, or 1x1 matrix containing the number of characters to strip.

Output

y	string array result.

Source

`strfns.src`

See Also

strtriml, strtrimr, strtrunc, strtruncpad, strtruncr

strtruncpad

Purpose

Truncates all elements of a string array to the specified number of characters, adding spaces on the end as needed to achieve the exact length.

Format

y = **strtruncpad**(*sa*, *maxlen*);

Input

sa	NxK string array.
maxlen	1xK or 1x1 matrix, maximum length.

Output

y	NxK string array result.

See Also

strtriml, strtrimr, strtrunc, strtruncl, strtruncr

strtruncr

Purpose

Truncates the right side of all elements of a string array by a user-specified number of characters.

Format

y = **strtruncr**(*sa*, *ntrunc*);

Input

sa	NxM, Nx1, 1xM, or 1x1 string array.
ntrunc	NxM, Nx1, 1xM, or 1x1 matrix containing the number of characters to strip.

Output

y	String array result.

Source

`strfns.src`

See Also

strtriml, strtrimr, strtrunc, strtruncl, strtruncpad

submat

Purpose

Extracts a submatrix of a matrix, with the appropriate rows and columns given by the elements of vectors.

Format

y = **submat**(*x*, *r*, *c*);

Input

x	NxK matrix.
r	LxM matrix of row indices.
c	PxQ matrix of column indices.

Output

y	(L*M)x(P*Q) submatrix of *x*, *y* may be larger than *x*.

Remarks

If $r = 0$, then all rows of x will be used. If $c = 0$, then all columns of x will be used.

Example

```
//Create 12x1 vector with consecutive numbers
x = seqa(1, 1, 12);

//Reshape the 12x1 vector into a 3x4 matrix
x = reshape(x, 3, 4);

v1 = 1 3;
v2 = 2 4;

//Extract sub-matrices
y = submat(x,v1,v2);
z = submat(x,0,v2);
```

After the code above, the matrix values are:

```
    1  2  3  4
x = 5  6  7  8
    9 10 11 12

y =  2  4
    10 12

     2  4
z =  6  8
    10 12
```

See Also

diag, vec, reshape

subscat

Purpose

Changes the values in a vector depending on the category a particular element falls in.

Format

y = **`subscat`**(*x*, *v*, *s*);

Input

x	Nx1 vector.
v	Px1 numeric vector, containing breakpoints specifying the ranges within which substitution is to be made. This MUST be sorted in ascending order. *v* can contain a missing value as a separate category if the missing value is the first element in *v*. If *v* is a scalar, all matches must be exact for a substitution to be made.
s	Px1 vector, containing values to be substituted.

Output

y	Nx1 vector, with the elements in *s* substituted for the original elements of *x* according to which of the regions the elements of *x* fall into:

```
    x ≤ v[1] → s[1]
        v[1] < x ≤ v[2] → s[2]
                  . . .
    v[p - 1] < x ≤ v[p] → s[p]
           x > v[p] → the original value of x
```

If missing is not a category specified in *v*, missings in *x* are passed through without change.

Example

```
//Create an additive sequence from 1-10
x = seqa(1, 1, 10);

//Set the breakpoints which indicate where to apply the
//substitution such that elements 1-4 in 'x' will be set to
//the first value of 'sub', the 5th and 6th values will be
//set to the second element of 'sub' and the 7th-10th
//elements will be set to the third element of 'sub'
bp = { 4, 6, 10 };

//The substitution values
sub = { 3.14, 6.28, 9.42 };

y = subscat(x,bp,sub);
```

The above code assigns the following values:

```
          1       3.14
          2       3.14
          3       3.14
          4       3.14
    x =   5   y = 6.28
          6       6.28
          7       9.42
          8       9.42
          9       9.42
         10       9.42
```

substute

Purpose

Substitutes new values for old values in a matrix, depending on the outcome of a logical expression.

Format

y = **substute**(*x*, *e*, *v*);

Input

x	NxK matrix containing the data to be changed.
e	LxM matrix, ExE conformable with *x* containing 1's and 0's.
v	PxQ matrix, ExE conformable with *x* and *e*, containing the values to be substituted for the original values of *x* when the corresponding element of *e* is 1.

Output

y	max(N,L,P) by max(K,M,Q) matrix.

Remarks

The *e* matrix is usually the result of an expression or set of expressions using dot conditional and boolean operators.

Example

```
//Create a matrix with character elements for the
//first column
x = { Y 55 30,
      N 57 18,
      Y 24 3,
      N 63 38,
      Y 55 32,
      N 37 11 };

//Create a rows(x) by 1 vector with a '1' for each row
// that:
//  1) The first element is a Y
//  2) The second element is greater than or equal to 55
//  3) The third element is greater than or equal to 30
//If the row does not meet ALL of these conditions a 0 will
//be returned.
e = x[.,1] .$== "Y" .and x[.,2] .>= 55 .and x[.,3] .>= 30;

//Substitute an 'R' for the first element in every row that
//meets the conditions specified in the assignment to 'e'
x[.,1] = substute(x[.,1],e, "R");
```

The vector e is equal to:

```
    1
    0
    0
    0
    1
    0
```

Here is what x looks like after substitution:

```
    R 55 30
    N 57 18
    Y 24  3
    N 63 38
    R 55 32
    N 37 11
```

Source

`datatran.src`

See Also

code, recode

subvec

Purpose

Extracts an Nx1 vector of elements from an NxK matrix.

Format

y = **subvec**(*x*, *ci*);

Input

x	NxK matrix.
ci	Nx1 vector of column indices.

Output

y	Nx1 vector containing the elements in *x* indicated by *ci*.

Remarks

Each element of *y* is from the corresponding row of *x* and the column set by the corresponding row of *ci*. In other words, *y*[*i*] = *x*[*i*, *ci*[*i*]].

Example

```
//Create an additive sequence from 1-12, i.e. 1, 2, 3,...12
x = seqa(1, 1, 12);

//Reshape the sequential vector 'x' into a 4x3 matrix
x = reshape(x,4,3);

//The column indices (one per row of 'x') indicating which
//values to extract from 'x'
ci = { 2, 3, 1, 3 };
```

```
//Extract subvector from 'x' and assign it to 'y'
y = subvec(x,ci);
```

After the above code, *x* and *y* are equal to:

```
      1  2  3
x =   4  5  6
      7  8  9
     10 11 12

      2
  y = 6
      7
     12
```

sumc

Purpose

Computes the sum of each column of a matrix or the sum across the second-fastest moving dimension of an L-dimensional array.

Format

y = **sumc**(*x*);

Input

x	NxK matrix or L-dimensional array where the last two dimensions are NxK.

Output

y	Kx1 vector or L-dimensional array where the last two dimensions are Kx1.

Example

```
//Create a 12x1 vector containing an additive sequence
//counting by twos, from 0-22, i.e. 2, 4, 6, 8...22
x = seqa(0,2,12);

//Reshape the 12x1 vector 'x' into a 3x4 matrix
x = reshape(x,3,4));

//Sum the columns
y = sumc(x);
```

After the above code, the variables *x* and *y* are equal to:

```
      0  2  4  6
x =   8 10 12 14
     16 18 20 22

     24
y =  30
     36
     42
```

```
//Create an additive sequence from 1-24 and reshape it into
//a 2x3x4 array
a = areshape(seqa(1,1,24),2|3|4);

//Sum the columns across the second fastest moving
```

```
//dimension
z = sumc(a);
```

a is a 2x3x4 array such that:

```
Plane [1,.,.]

      1.0000000     2.0000000     3.0000000     4.0000000
      5.0000000     6.0000000     7.0000000     8.0000000
      9.0000000     10.000000     11.000000     12.000000

Plane [2,.,.]

      13.000000     14.000000     15.000000     16.000000
      17.000000     18.000000     19.000000     20.000000
      21.000000     22.000000     23.000000     24.000000
```

Variable *z* is a 2x4x1 array equal to:

```
Plane [1,.,.]

      15.000000
      18.000000
      21.000000
      24.000000

Plane [2,.,.]

      51.000000
      54.000000
      57.000000
      60.000000
```

See Also

cumsumc, meanc, stdc

sumr

Purpose

Computes the sum of each row of a matrix or the sum of the fastest moving dimension of an L-dimensional array.

Format

y = **sumr**(*x*);

Input

x	NxK matrix or L-dimensional array where the last two dimensions are NxK.

Output

y	Nx1 vector or L-dimensional array where the last two dimensions are Nx1.

Example

```
//Create an additive sequence from 1-12 and reshape it into
//a 3x4 matrix
x = reshape(seqa(1,1,12),3,4);

//Sum the rows
y = sumr(x);
```

After the above code, the variables *x* and *y* will be:

```
        1  2  3  4        10
    x = 5  6  7  8    y = 26
        9 10 11 12        42
```

```
    //Reshape an additive sequence from 1-24 into a 2x3x4
    //dimensional array
    a = areshape(seqa(1,1,24),2|3|4);
    z = sumr(a);
```

a is a 2x3x4 array such that:

```
    Plane [1,.,.]

          1.0000000     2.0000000     3.0000000     4.0000000
          5.0000000     6.0000000     7.0000000     8.0000000
          9.0000000     10.000000     11.000000     12.000000

    Plane [2,.,.]

          13.000000     14.000000     15.000000     16.000000
          17.000000     18.000000     19.000000     20.000000
          21.000000     22.000000     23.000000     24.000000
```

The variable *z* is equal to:

```
    Plane [1,.,.]

          10.000000
          26.000000
          42.000000
    Plane [2,.,.]

          58.000000
          74.000000
          90.000000
```

See Also

sumc

surface

Purpose

Graphs a 3-D surface. NOTE: This function is for use with the deprecated PQG graphics. Use **`plotSurface`** instead.

Library

pgraph

Format

`surface`(*x*, *y*, *z*);

Input

x	1xK vector, the X axis data.
y	Nx1 vector, the Y axis data.

z	NxK matrix, the matrix of height data to be plotted.

Global Input

_psurf 2x1 vector, controls 3-D surface characteristics.

[1]	if 1, show hidden lines. Default 0.
[2]	color for base (default 7). The base is an outline of the X-Y plane with a line connecting each corner to the surface. If 0, no base is drawn.

_pticout scalar, if 0 (default), tick marks point inward, if 1, tick marks point outward.

_pzclr Z level color control.

There are 3 ways to set colors for the Z levels of a surface graph.

1. To specify a single color for the entire surface plot, set the color control variable to a scalar value 1-15. For example:

```
    _pzclr = 15;
```

2. To specify multiple colors distributed evenly over the entire Z range, set the color control variable to a vector containing the desired colors only. **GAUSS** will automatically calculate the required corresponding Z values for you. The following example will produce a three color surface plot, the Z ranges being lowest=blue, middle=light blue,

highest=white:

```
    _pzclr = { 1, 10, 15 };
```

3. To specify multiple colors distributed over selected ranges, the Z ranges as well as the colors must be manually input by the user. The following example assumes -0.2 to be the minimum value in the *z* matrix:

```
    _pzclr = { -0.2 1,
    /* z >= -0.2 blue */
     0.0 10,
    /* z >= 0.0 light blue */
     0.2 15 };
    /* z >= 0.2 white */
```

Since a Z level is required for each selected color, the user must be responsible to compute the minimum value of the *z* matrix as the first Z range element. This may be most easily accomplished by setting the `_pzclr` matrix as shown above (the first element being an arbitrary value), then resetting the first element to the minimum *z* value as follows:

```
    _pzclr = { 0.0 1,
               0.0 10,
               0.2 15 };
    _pzclr[1,1] = minc(minc(z));
```

See **PQG Graphics Colors**, Section 1, for the list of available colors.

Remarks

`surface` uses only the minimum and maximum of the X axis data in generating the graph and tick marks.

Source

`psurface.src`

See Also

volume, view

svd

Purpose

Computes the singular values of a matrix.

Format

s = **`svd`**(*x*);

Input

x	NxP matrix whose singular values are to be computed.

Output

s	Mx1 vector, where M = **`min`**(N,P), containing

	the singular values of *x* arranged in descending order.

Global Input

_svderr	scalar, if not all of the singular values can be computed, *_svderr* will be nonzero. The singular values in *s*[*_svderr*+1], ... *s*[M] will be correct.

Remarks

Error handling is controlled with the low bit of the trap flag.

trap 0	set *_svderr* and terminate with message
trap 1	set *_svderr* and continue execution

Example

```
//Create a 5x5 random normal matrix
x = rndn(5,5);

//Calculate the singular values of matrix 'x'
y = svd(x);
```

Source

`svd.src`

See Also

svd2, svds

svd1

Purpose

Computes the singular value decomposition of a matrix so that: $x = u * s * v'$.

Format

{ *u*, *s*, *v* } = **svd1**(*x*);

Input

x	NxP matrix whose singular values are to be computed.

Output

u	NxN matrix, the left singular vectors of *x*.
s	NxP diagonal matrix, containing the singular values of *x* arranged in descending order on the principal diagonal.
v	PxP matrix, the right singular vectors of *x*.

Global Output

_svderr	scalar, if all of the singular values are correct, *_svderr* is 0. If not all of the singular values can be computed, *_svderr* is

set and the diagonal elements of *s* with indices greater than _ *svderr* are correct.

Remarks

Error handling is controlled with the low bit of the trap flag.

trap 0	set _*svderr* and terminate with message
trap 1	set _*svderr* and continue execution

Example

```
//Create 10x10 random normal matrix
x = rndn(10,10);

//Perform matrix decomposition
{ u, s, v } = svd1(x);

newx = u*s*v';

//Calculate the largest difference between 'x' and 'newx'
maxdiff = maxc(maxc(abs(newx - x) ));
```

Source

svd.src

See Also

svd, svd2, svdusv

svd2

Purpose

Computes the singular value decomposition of a matrix so that: $x = u * s * v'$ (compact u).

Format

```
{ u, s, v } = svd2(x);
```

Input

x	NxP matrix whose singular values are to be computed.

Output

u	NxN or NxP matrix, the left singular vectors of *x*. If N > P, then *u* will be NxP, containing only the P left singular vectors of *x*.
s	NxP or PxP diagonal matrix, containing the singular values of *x* arranged in descending order on the principal diagonal. If N > P, then *s* will be PxP.
v	PxP matrix, the right singular vectors of *x*.

Global Output

_svderr	scalar, if all of the singular values are correct, *_svderr* is 0. If not all of the singular values can be computed, *_svderr* is

set and the diagonal elements of *s* with indices greater than _*svderr* are correct.

Remarks

Error handling is controlled with the low bit of the trap flag.

trap 0	set _*svderr* and terminate with message
trap 1	set _*svderr* and continue execution

Source

```
svd.src
```

See Also

svd, svd1, svdcusv

svdcusv

Purpose

Computes the singular value decomposition of *x* so that: $x = u * s * v'$ (compact *u*).

Format

{ *u*, *s*, *v* } = **svdcusv**(*x*);

Input

`x`	NxP matrix or K-dimensional array where the last two dimensions are NxP, whose singular values are to be computed.

Output

`u`	NxN or NxP matrix or K-dimensional array where the last two dimensions are NxN or NxP, the left singular vectors of `x`. If N > P, `u` is NxP, containing only the P left singular vectors of `x`.
`s`	NxP or PxP diagonal matrix or K-dimensional array where the last two dimensions describe NxP or PxP diagonal arrays, the singular values of `x` arranged in descending order on the principal diagonal. If N > P, `s` is PxP.
`v`	PxP matrix or K-dimensional array where the last two dimensions are PxP, the right singular vectors of `x`.

Remarks

If `x` is an array, the resulting arrays `u`, `s` and `v` will contain their respective results for each of the corresponding 2-dimensional arrays described by the two trailing dimensions of `x`. In other words, for a 10x4x5 array `x`, `u` will be a 10x4x4 array containing the left singular vectors of each of the 10 corresponding 4x5 arrays contained in `x`. `s` will be a 10x4x5 array and `v` will be a 10x5x5 array both containing their respective results for each of the 10 corresponding 4x5 arrays contained in `x`.

If not all of the singular values can be computed, *s*[1,1] is set to a scalar error code. Use **`scalerr`** to convert this to an integer. The diagonal elements of *s* with indices greater than **`scalerr`**(*s*[1,1]) are correct. If **`scalerr`**(*s*[1,1]) returns a 0, all of the singular values have been computed.

See Also

svd2, svds, svdusv

svds

Purpose

Computes the singular values of a *x*.

Format

s = **`svds`**(*x*);

Input

x	NxP matrix or K-dimensional array where the last two dimensions are NxP, whose singular values are to be computed.

Output

s	min(N,P)x1 vector or K-dimensional array where the last two dimensions are min(N,P)x1, the singular values of *x*

	arranged in descending order.

Remarks

If *x* is an array, the result will be an array containing the singular values of each of the 2-dimensional arrays described by the two trailing dimensions of *x*. In other words, for a 10x4x5 array *x*, *s* will be a 10x4x1 array containing the singular values of each of the 10 4x5 arrays contained in *x*.

If not all of the singular values can be computed, *s*[1] is set to a scalar error code. Use **scalerr** to convert this to an integer. The elements of *s* with indices greater than **scalerr**(*s*[1]) are correct. If **scalerr**(*s*[1]) returns a 0, all of the singular values have been computed.

See Also

svd, svdcusv, svdusv

svdusv

Purpose

Computes the singular value decomposition of *x* so that: $x = u * s * v'$.

Format

{ *u*, *s*, *v* } = **svdusv**(*x*);

Input

x	NxP matrix or K-dimensional array where the last two

	dimensions are NxP, whose singular values are to be computed.

Output

u	NxN matrix or K-dimensional array where the last two dimensions are NxN, the left singular vectors of *x*.
s	NxP diagonal matrix or K-dimensional array where the last two dimensions describe NxP diagonal arrays, the singular values of *x* arranged in descending order on the principal diagonal.
v	PxP matrix or K-dimensional array where the last two dimensions are PxP, the right singular vectors of *x*.

Remarks

If *x* is an array, the resulting arrays *u*, *s* and *v* will contain their respective results for each of the corresponding 2-dimensional arrays described by the two trailing dimensions of *x*. In other words, for a 10x4x5 array *x*, *u* will be a 10x4x4 array containing the left singular vectors of each of the 10 corresponding 4x5 arrays contained in *x*. *s* will be a 10x4x5 array and *v* will be a 10x5x5 array both containing their respective results for each of the 10 corresponding 4x5 arrays contained in *x*.

If not all of the singular values can be computed, *s*[1,1] is set to a scalar error code. Use **scalerr** to convert this to an integer. The diagonal elements of *s* with indices greater than **scalerr**(*s*[1,1]) are correct. If **scalerr**(*s*[1,1]) returns a 0, all of the singular values have been computed.

See Also

svd1, svdcusv, svds

sysstate

Purpose

Gets or sets general system parameters.

Format

{ *rets...* } = **sysstate**(*case*, *y*);

Remarks

The available cases are as follows:

Case 1	**Version Information** Returns the current **GAUSS** version information in an 8-element numeric vector.
Cases 2-7	**GAUSS System Paths** Gets or sets **GAUSS** system path.
Case 8	**Complex Number Toggle** Controls automatic generation of complex numbers in **sqrt**, **ln**, and **log** for negative arguments.
Case 9	**Complex Trailing Character** Gets or sets trailing character for the imaginary part of a complex number.
Case 10	**Printer Width** Gets or sets lprint width.
Case 11	**Auxiliary Output Width** Gets or sets the auxiliary output width.
Case 13	**LU Tolerance** Gets or sets singularity tolerance for LU

	decomposition in current thread.
Case 14	**Cholesky Tolerance** Gets or sets singularity tolerance for Cholesky decomposition in current thread.
Case 15	**Screen State** Gets or sets window state as controlled by `screen` command.
Case 18	**Auxiliary Output** Gets auxiliary output parameters.
Case 19	**Get/Set Format** Gets or sets format parameters.
Case 21	**Imaginary Tolerance** Gets or sets imaginary tolerance in current thread.
Case 22	**Source Path** Gets or sets the path the compiler will search for source files.
Case 24	**Dynamic Library Directory** Gets or sets the path for the default dynamic library directory.
Case 25	**Temporary File Path** Gets or sets the path **GAUSS** will use for temporary files.
Case 26	**Interface Mode** Returns the current interface mode.
Case 28	**Random Number Generator Parameters** Gets or sets parameters used by the random number generation commands.
Case 30	**Base Year Toggle** Specifies whether year value returned by **`date`** is to include base year (1900) or not.
Case 32	**Global LU Tolerance** Gets or sets global singularity tolerance for LU decomposition.
Case 33	**Global Cholesky Tolerance** Gets or sets global singularity tolerance for Cholesky decomposition.
Case 34	**Global Imaginary Tolerance** Gets or sets global imaginary tolerance.

Case 1: Version Information

Purpose

Returns the current **GAUSS** version information in an 8-element numeric vector.

Format

```
vi = sysstate(1,0);
```

Output

vi	8x1 numeric vector containing version information:	
	[1]	Major version number.
	[2]	Minor version number.
	[3]	Revision.
	[4]	Machine type.
	[5]	Operating system.
	[6]	Runtime module.
	[7]	Light version.
	[8]	Always 0.

vi[4] indicates the type of machine on which **GAUSS** is running:

1	Intel x86
2	Sun SPARC

4	HP 9000
7	Mac 32-bit PowerPC

vi[5] indicates the operating system on which **GAUSS** is running:

3	Solaris
5	HP-UX
9	Windows
10	Linux
12	Mac OS

Cases 2-7: GAUSS System Paths

Purpose

Gets or sets **GAUSS** system path.

Format

```
oldpath = sysstate(case, path);
```

Input

case	scalar 2-7, path to set.	
	2	`.exe` file location.
	3	`loadexe` path.

	4	`save` path.
	5	`load`, `loadm` path.
	6	`loadf`, `loadp` path.
	7	**`loads`** path.
`path`	scalar 0 to get path, or string containing the new path.	

Output

`oldpath`	string, original path.

Remarks

If *`path`* is of type matrix, the path will be returned but not modified.

Case 8: Complex Number Toggle

Purpose

Controls automatic generation of complex numbers in **`sqrt`**, **`ln`** and **`log`** for negative arguments.

Format

`oldstate` = **`sysstate`**(8, *`state`*);

Input

`state`	scalar, 1, 0, or -1

Output

oldstate	scalar, the original state.

Remarks

If *state* = 1, **log**, **ln**, and **sqrt** will return complex numbers for negative arguments. If *state* = 0, the program will terminate with an error message when negative numbers are passed to **log**, **ln**, and **sqrt**. If *state* = -1, the current state is returned and left unchanged. The default state is 1.

Case 9: Complex Trailing Character

Purpose

Gets or sets trailing character for the imaginary part of a complex number.

Format

oldtrail = **sysstate**(9, *trail*);

Input

trail	scalar 0 to get character, or string containing the new trailing character.

Output

oldtrail	string, the original trailing character.

Remarks

The default character is "*i*".

Case 10: Printer Width

Purpose

Gets or sets `lprint` width.

Format

```
oldwidth = sysstate(10, width);
```

Input

width	scalar, new printer width.

Output

oldwidth	scalar, the current original width.

Remarks

If *width* is 0, the printer width will not be changed.

Case 11: Auxiliary Output Width

Purpose

Gets or sets the auxiliary output width.

Format

oldwidth = **sysstate**(11, *width*);

Input

width	scalar, new output width.

Output

oldwidth	scalar, the original output width.

Remarks

If *width* is 0 then the output width will not be changed.

This may also be set with the `outwidth` command.

See Also

outwidth

Case 13: LU Tolerance

Purpose

Gets or sets singularity tolerance for LU decomposition in current thread.

Format

oldtol = **sysstate**(13, *tol*);

Input

tol	scalar, new tolerance.

Output

oldtol	scalar, the original tolerance.

Remarks

The tolerance must be ≥0. If *tol* is negative, the tolerance is returned and left unchanged.

This tolerance is thread-safe. It must be set in the same thread in which it is to be referenced. To set the global singularity tolerance for LU decomposition, use case 32.

See Also

croutp, inv

Case 14: Cholesky Tolerance

Purpose

Gets or sets singularity tolerance for Cholesky decomposition in current thread.

Format

```
oldtol = sysstate(14, tol);
```

Input

tol	scalar, new tolerance.

Output

oldtol	scalar, the original tolerance.

Remarks

The tolerance must be ≥ 0. If *tol* is negative, the tolerance is returned and left unchanged.

This tolerance is thread-safe. It must be set in the same thread in which it is to be referenced. To set the global singularity tolerance for Cholesky decomposition, use case 33.

This affects the following functions:

solpd
invpd for matrices ≤ 12x12

See Also

chol, invpd, solpd

Case 15: Screen State

Purpose

Gets or sets window state as controlled by `screen` command.

Format

oldstate = **sysstate**(15, *state*);

Input

state	scalar, new window state.

Output

oldstate	scalar, the original window state.

Remarks

If *state* = 1, window output is turned on. If *state* = 0, window output is turned off. If *state* = -1, the state is returned unchanged.

See Also

screen

Case 18: Auxiliary Output

Purpose

Gets auxiliary output parameters.

Format

{ *state*, *name* } = **sysstate**(18,0);

Output

state	scalar, auxiliary output state, 1 - on, 0 - off.
name	string, auxiliary output filename.

See Also

output

Case 19: Get/Set Format

Purpose

Gets or sets format parameters.

Format

oldfmt = **sysstate**(19, *fmt*);

Input

fmt	scalar or 11x1 column vector containing the new formatparameters. Usually this will have come from a previous **sysstate**(19,0) call. See Output for description of matrix.

Output

oldfmt	11x1 vector containing the current format parameters. The

characters in quotes are components of the format string that gets passed through to the C library **`sprintf`** function:

[1]	format conversion type:	
	0	string format ("*s*")
	1	compact format ("*g*").
	2	auto format ("*#g*").
	3	scientific format ("*e*").
	4	decimal format ("*f*").
	5	compact format, upper case ("*G*").
	6	auto format, upper case ("*#G*").
	7	scientific format, upper case ("*E*").
[2]	justification:	
	0	right justification.
	1	left justification ("*-*").
[3]	sign:	
	0	sign used only for negative numbers.
	1	sign always used ("*+*").
[4]	leading zero:	
	0	no leading zero.
	1	leading zero ("*0*").

[5] trailing character:

0	no trailing character.
1	trailing space ("").
2	trailing comma ("`,`").
3	trailing tab ("`\t`").

[6] row delimiter:

0	no row delimiter.
1	one newline between rows ("`\n`").
2	two newlines between rows ("`\n\n`").
3	print "**Row 1**, **Row 2**, ..." before each row ("`\nRow %u\n`", where "`%u`" is the row number).

[7] carriage line feed position:

0	newline row delimiters positioned before rows.
1	newline row delimiters positioned after rows.

[8] automatic line feed for row vectors.

0	newline row delimiters occur between rows of a matrix only if that matrix has more than one row.

		1	newline row delimiters occur between rows of a matrix, regardless of number of rows.
	[9]	field width.	
	[10]	precision.	
	[11]	formatted flag.	
		0	formatting disabled.
		1	formatting enabled.

Remarks

If *fmt* is scalar 0, then the format parameters will be left unchanged.

See the `format` and `print` commands for more information on the formatting parameters.

See Also

format, print

Case 21: Imaginary Tolerance

Purpose

Gets or sets imaginary tolerance in current thread.

Format

```
oldtol = sysstate(21, tol);
```

Input

tol	scalar, the new tolerance.

Output

oldtol	scalar, the original tolerance.

Remarks

The imaginary tolerance is used to test whether the imaginary part of a complex matrix can be treated as zero or not. Functions that are not defined for complex matrices check the imaginary part to see if it can be ignored. The default tolerance is 2.23e-16, or machine epsilon.

If *tol*<0, the current tolerance is returned.

This tolerance is thread-safe. It must be set in the same thread in which it is to be referenced. To set the global imaginary tolerance, use case 34.

See Also

hasimag

Case 22: Source Path

Purpose

Gets or sets the path the compiler will search for source files.

Format

oldpath = **sysstate**(22, *path*);

Input

path	scalar 0 to get path, or string containing the new path.

Output

oldpath	string, original path.

Remarks

If *path* is a matrix, the current source path is returned.

This resets the `src_path` configuration variable. `src_path` is initially defined in the **GAUSS** configuration file, `gauss.cfg`.

path can list a sequence of directories, separated by semicolons.

Resetting `src_path` affects the path used for subsequent `run` and `compile` statements.

Case 24: Dynamic Library Directory

Purpose

Gets or sets the path for the default dynamic library directory.

Format

oldpath = **sysstate**(24, *path*);

Input

path	scalar 0 to get path, or string containing the new path.

Output

oldpath	string, original path.

Remarks

If *path* is a matrix, the current path is returned.

path should list a single directory, not a sequence of directories.

Changing the dynamic library path does not affect the state of any DLL's currently linked to **GAUSS**. Rather, it determines the directory that will be searched the next time `dlibrary` is called.

UNIX

Changing the path has no effect on **GAUSS**'s default DLL, `libgauss.so`. `libgauss.so` must always be located in the GAUSSHOME directory.

Windows

Changing the path has no effect on **GAUSS**'s default DLL, `gauss.dll`. `gauss.dll` must always be located in the GAUSSHOME directory.

See Also

dlibrary, dllcall

Case 25: Temporary File Path

Purpose

Gets or sets the path **GAUSS** will use for temporary files.

Format

```
oldpath = sysstate(25, path);
```

Input

path	scalar 0 to get path, or string containing the new path.

Output

oldpath	string, original path.

Remarks

If *path* is of type matrix, the path will be returned but not modified.

Case 26: Interface Mode

Purpose

Returns the current interface mode.

Format

```
mode = sysstate(26,0);
```

Output

mode	scalar, interface mode flag	
	0	non-X mode
	1	terminal (-v) mode
	2	X Windows mode

Remarks

A mode of 0 indicates that you're running a non-X version of **GAUSS**; i.e., a version that has no X Windows capabilities. A mode of 1 indicates that you're running an X Windows version of **GAUSS**, but in terminal mode; i.e., you started **GAUSS** with the -v flag. A mode of 2 indicates that you're running **GAUSS** in X Windows mode.

Case 28: Random Number Generator Parameters

Purpose

Gets or sets the random number generator (RNG) parameters.

Format

oldprms = **sysstate**(28, *prms*);

Input

prms	scalar 0 to get parameters, or 3x1 matrix of new parameters.	
	[1] seed,	$0<\text{seed}<2^{32}$

[2]	multiplier,	$0<mult<2^{32}$
[3]	constant,	$0<=const<2^{32}$

Output

oldprms	3x1 vector, the original parameters.

Remarks

If *prms* is a scalar 0, the current parameters will be returned without being changed.

The modulus of the RNG cannot be changed; it is fixed at 2^{32}.

See Also

rndcon, rndmult, rndseed, rndn, rndu

Case 30: Base Year Toggle

Purpose

Specifies whether year value returned by **date** is to include base year (1900) or not.

Format

```
oldstate = sysstate(30, state);
```

Input

state	scalar, 1, 0, or missing value.

Output

oldstate	scalar, the original state.

Remarks

Internally, **date** acquires the number of years since 1900. **sysstate** case 30 specifies whether **date** should add the base year to that value or not. If *state* = 1, **date** adds 1900, returning a fully-qualified 4-digit year.

If *state* = 0, **date** returns the number of years since 1900. If *state* is a missing value, the current state is returned. The default state is 1.

Case 32: Global LU Tolerance

Purpose

Gets or sets global singularity tolerance for LU decomposition.

Format

oldtol = **sysstate**(32, *tol*);

Input

tol	scalar, new tolerance.

Output

oldtol	scalar, the original tolerance.

Remarks

The tolerance must be ≥0. If *tol* is negative, the tolerance is returned and left unchanged.

This is a global tolerance and therefore not thread-safe. To set the singularity tolerance for LU decomposition in the current thread, use case 13.

See Also

croutp, inv

Case 33: Global Cholesky Tolerance

Purpose

Gets or sets global singularity tolerance for Cholesky decomposition.

Format

oldtol = **sysstate**(33, *tol*);

Input

tol	scalar, new tolerance.

Output

oldtol	scalar, the original tolerance.

Remarks

The tolerance must be ≥0. If *tol* is negative, the tolerance is returned and left unchanged.

This is a global tolerance and therefore not thread-safe. To set the singularity tolerance for Cholesky decomposition in the current thread, use case 14.

This affects the following functions:

solpd
invpd for matrices ≤ 12x12

See Also

chol, invpd, solpd

Case 34: Global Imaginary Tolerance

Purpose

Gets or sets the global imaginary tolerance.

Format

oldtol = **sysstate**(34, *tol*);

Input

tol	scalar, the new tolerance.

Output

oldtol	scalar, the original tolerance.

Remarks

The imaginary tolerance is used to test whether the imaginary part of a complex matrix can be treated as zero or not. Functions that are not defined for complex matrices check the imaginary part to see if it can be ignored. The default tolerance is 2.23e-16, or machine epsilon.

If *tol*<0, the current tolerance is returned.

This is a global tolerance and therefore not thread-safe. To set the imaginary tolerance in the current thread, use case 21.

See Also

hasimag

system

Purpose

Quits **GAUSS** and returns to the operating system.

Format

```
    system;
    system c;
```

Input

c	scalar, an optional exit code that can be recovered by the program that invoked **GAUSS**. The default is 0. Valid arguments are 0-255.

Remarks

The `system` command always returns an exit code to the operating system or invoking program. If you don't supply one, it returns 0. This is usually interpreted as indicating success.

See Also

exec

tab

Purpose

Tabs the cursor to a specified text column.

Format

tab(*col*);

print *expr1 expr2* **tab**(*col1*) *expr3* **tab**(*col2*) *expr4* ...;

Input

col	scalar, the column position to tab to.

Remarks

col specifies an absolute column position. If *col* is not an integer, it will be truncated.

tab can be called alone or embedded in a print statement. You cannot embed it within a parenthesized expression in a print statement, though. For example:

```
    print (tab(20) c + d * e);
```

will not give the results you expect. If you have to use parenthesized expressions, write it like this instead:

```
    print tab(20) (c + d * e);
```

tan

Purpose

Returns the tangent of its argument.

Format

y = **tan**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array.

Remarks

For real matrices, *x* should contain angles measured in radians.

To convert degrees to radians, multiply the degrees by $\pi/180$.

Example

```
    //Create an additive sequence 0.1, 0.2, 0.3...0.9
    x = seqa(0.1, 0.1, 9);

    y = tan(x);
```

The above code produces:

```
         0.1003346
         0.2027100
         0.3093362
         0.4227932
    y =  0.5463024
         0.6841368
         0.8422883
         1.0296386
         1.2601582
```

See Also

atan, pi

tanh

Purpose

Computes the hyperbolic tangent.

Format

y = **tanh**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing the hyperbolic tangents of the elements of *x*.

Example

```
//Create a sequence starting at -0.5 and increasing by
//0.25, i.e. -0.5, -0.25, 0, 0.25...1
x = seqa(-0.5, 0.25, 7);
x = x * pi;
y = tanh(x);
```

After the above code, *y* is equal to:

```
-0.46211716
-0.24491866
 0.00000000
 0.24491866
 0.46211716
 0.63514895
 0.76159416
```

Source

`trig.src`

tempname

Purpose

Creates a temporary file with a unique name.

Format

`tname = tempname(path, pre, suf);`

Input

path	string, path where the file will reside.
pre	string, a prefix to begin the file name with.
suf	string, a suffix to end the file name with.

Output

tname	string, unique temporary file name of the form *path*/*pre*XXXXnnnnn*suf*, where XXXX are 4 letters, and nnnnn is the process id of the calling process.

Remarks

Any or all of the inputs may be a null string or 0. If *path* is not specified, the current working directory is used.

If unable to create a unique file name of the form requested, **tempname** returns a null string.

WARNING: **GAUSS** does not remove temporary files created by **`tempname`**. It is left to the user to remove them when they are no longer needed.

ThreadBegin

Purpose

Marks the beginning of a multi-line block of code to be executed as a thread.

Format

```
ThreadBegin;
```

Example

```
ThreadBegin;
   m = n*p;
   n = calcA(m);
ThreadEnd;
```

Notice that the **writer-must-isolate** rule (see **Multi-Threaded Programming in GAUSS**, Chapter 1) does not apply within the bounds of the `ThreadBegin`/`ThreadEnd` pair, as there is no risk of simultaneous access to a symbol. The rule only applies between the threads in a given set (and their children).

See `ThreadJoin` for an example of a fully-defined thread set.

See Also

ThreadEnd, ThreadJoin, ThreadStat

ThreadEnd

Purpose

Marks the end of a multi-line block of code to be executed as a thread.

Format

```
ThreadEnd;
```

Example

```
ThreadBegin;
   m = n*p;
   n = calcA(m);
ThreadEnd;
```

Notice that the **writer-must-isolate** rule (see **Multi-Threaded Programming in GAUSS**, Chapter 1) does not apply within the bounds of the `ThreadBegin`/`ThreadEnd` pair, as there is no risk of simultaneous access to a symbol. The rule only applies between the threads in a given set (and their children).

See `ThreadJoin` for an example of a fully-defined thread set.

See Also

ThreadBegin, ThreadJoin, ThreadStat

ThreadJoin

Purpose

Completes the definition of a set of threads to be executed simultaneously.

Format

```
ThreadJoin;
```

Remarks

Each thread in the set must adhere to the **writer-must-isolate** rule (see **Multi-Threaded Programming in GAUSS**, Chapter 1). Because the threads in a set execute simultaneously, there is no way of knowing in one thread the current "state" of a symbol in another, and thus no way of safely or meaningfully accessing it.

Example

```
ThreadBegin;          //Thread 1--isolates y,z
   y = x'x;
   z = y'y;
ThreadEnd;
ThreadBegin;          //Thread 2--isolates q,r
   q = r'r;
   r = q'q;
ThreadEnd;
ThreadStat n = m'm;  //Thread 3--isolates n
ThreadStat p = o'o;  //Thread 4--isolates p
ThreadJoin;          //Joins threads 1-4
b = z + r + n'p;      //y,z,q,r,n,p available again,
                       // can be read and written
```

Note how threads 1-4 isolate the various symbols they assign to--no other thread references the written symbols at all. Once the threads are joined, however, the symbols are again available for use, and can be both read and assigned to.

See Also

ThreadBegin, ThreadEnd, ThreadStat

ThreadStat

Purpose

Marks a single line of code to be executed as a thread.

Format

```
ThreadStat statement;
```

Example

```
ThreadStat m = n*p;
```

See `ThreadJoin` for an example of a fully-defined thread set.

See Also

ThreadBegin, ThreadEnd, ThreadJoin

time

Purpose

Returns the current system time.

Format

y = **`time`**;

Output

y	4x1 numeric vector, the current time in the order: hours, minutes, seconds, and hundredths of a second.

Example

```
print time;
```

```
     7.000000
     31.000000
     46.000000
     33.000000
```

See Also

date, datestr, datestring, datestrymd, hsec, timestr

timedt

Purpose

Returns system date and time in DT scalar format.

Format

dt = **timedt;**

Output

dt	scalar, system date and time in DT scalar format.

Remarks

The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number:

```
    20100306071511
```

represents:

```
    07:15:11 or 7:15:11 AM on March 6, 2010.
```

Source

time.src

See Also

todaydt, timeutc, dtdate

timestr

Purpose

Formats a time in a vector to a string.

Format

ts = **timestr**(*t*);

Input

t	4x1 vector from the **time** function, or a zero. If the input is

	0, the **`time`** function will be called to return the current system time.

Output

`ts`	8 character string containing current time in the format: hr:mn:sc

Example

```
    t = { 7, 31, 46, 33 };
    ts = timestr(t);
    print ts;
```

produces:

```
    7:31:46
```

Source

`time.src`

See Also

date, datestr, datestring, datestrymd, ethsec, etstr, time

timeutc

Purpose

Returns the number of seconds since January 1, 1970 Greenwich Mean Time.

Format

```
tc = timeutc;
```

Output

tc	scalar, number of seconds since January 1, 1970 Greenwich Mean Time.

Example

```
//Retrieve seconds since January 1, 1970 GMT
tc = timeutc;

//Convert to a date time vector
utv = utctodtv(tc);
```

After the code above, *tc* and *utv* are equal to:

```
tc = 1340080112

utv = 2012 06 18 21 28 32 1 169
```

See Also

dtvnormal, utctodtv

title

Purpose

Sets the title for the graph. NOTE: This function is for the deprecated PQG graphics. Use **`plotSetTitle`** instead.

Library

`pgraph`

Format

`title`(*`str`*);

Input

`str`	string, the title to display above the graph.

Remarks

Up to three lines of title may be produced by embedding a line feed character ("\L") in the title string.

Example

```
title("First title line\LSecond title line\L"\
"Third title line");
```

Fonts may be specified in the title string. For instructions on using fonts, see **Selecting Fonts**, Section 1.0.1.

Source

`pgraph.src`

See Also

xlabel, ylabel, fonts

tkf2eps

Purpose

Converts a `.tkf` file to an Encapsulated PostScript file. NOTE: This function is deprecated and does not work for the new `.plot` graphics files. Use **`plotSave`** to convert `.plot` files to EPS format.

Library

`pgraph`

Format

ret = **tkf2eps**(*tekfile*, *epsfile*);

Input

tekfile	string, name of `.tkf` file.
epsfile	string, name of Encapsulated PostScript file.

Output

ret	scalar, 0 if successful

Remarks

The conversion is done using the global parameters in `peps.dec`. You can modify these globally by editing the `.dec` file, or locally by setting them in your program before calling **`tkf2eps`**.

See the header of the output Encapsulated PostScript file and a PostScript manual if you want to modify these parameters.

tkf2ps

Purpose

Converts a `.tkf` file to a PostScript file. NOTE: This function is deprecated and does not work for the new `.plot` graphics files. Use **`plotSave`** to convert `.plot` files to PS format.

Library

pgraph

Format

```
ret = tkf2ps(tekfile, psfile);
```

Input

`tekfile`	string, name of `.tkf` file.
`psfile`	string, name of PostScript file.

Output

`ret`	scalar, 0 if successful.

Remarks

The conversion is done using the global parameters in `peps.dec`. You can modify these globally by editing the `.dec` file, or locally by setting them in your program before calling **`tkf2ps`**.

See the header of the output PostScript file and a PostScript manual if you want to modify these parameters.

tocart

Purpose

Converts from polar to Cartesian coordinates.

Format

xy = **`tocart`**(*r*, *`theta`*);

Input

r	NxK real matrix, radius.
theta	LxM real matrix, ExE conformable with *r*, angle in radians.

Output

xy	max(N,L) by max(K,M) complex matrix containing the *x* coordinate in the real part and the *y* coordinate in the imaginary part.

Source

`coord.src`

todaydt

Purpose

Returns system date in DT scalar format. The time returned is always midnight (00:00:00), the beginning of the returned day.

Format

dt = **todaydt**();

Output

dt	scalar, system date in DT scalar format.

Remarks

The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number:

```
20130906130525
```

represents 13:05:25 or 1:05:25 PM on September 6, 2013.

Example

todaydt can be used with the function **dttostr** to print the current date as a string:

```
//Get today's date in dt scalar format
dt = todaydt();

//Convert the dtscalar to a desired format
dt_str = dttostr(dt, "MO/DD/YYYY");

//Combining strings with the '$+' operator
print "Today is "$+dt_str;
```

returns:

```
Today is 10/22/2013
```

Source

time.src

See Also

dttostr, timedt, timeutc, dtdate

toeplitz

Purpose

Creates a Toeplitz matrix from a column vector.

Format

t = **toeplitz**(*x*);

Input

x	Kx1 vector.

Output

t	KxK Toeplitz matrix.

Example

```
//Create the sequence 1, 2, 3, 4, 5 and assign it to 'x'
x = seqa(1,1,5);

//Create a diagonal-constant or Toeplitz matrix
y = toeplitz(x);
```

After the code above, *y* is equal to:

```
1   2   3   4   5
2   1   2   3   4
3   2   1   2   3
```

```
    4   3   2   1   2
    5   4   2   2   3
```

Source

`toeplitz.src`

token

Purpose

Extracts the leading token from a string.

Format

{ *token*, *str_left* } = **token**(*str*);

Input

str	string, the string to parse.

Output

token	string, the first token in *str*.
str_left	string, *str* minus *token*.

Remarks

str can be delimited with commas or spaces.

The advantage of **`token`** over **`parse`** is that **`parse`** is limited to tokens of 8 characters or less; **`token`** can extract tokens of any length.

Example

Example 1

```
    title = "Australia exports";
    { country, str_left } = token(title);
    print country;
    print str_left;
```

returns:

```
    Australia
    exports
```

Example 2

Here is a keyword that uses **`token`** to parse its string parameter:

```

    //Create a keyword called 'add' that takes the input
    //'s' and executes all of the code from the 'keyword
    //add(s)' line until the 'endp' statement each time
    //it is called
    keyword add(s);
       local tok,sum;
       sum = 0;

       //Continue loop until 's' equals an empty string
       do until s $== "";

          //Remove the first token from 's' and return
          //it in 'tok'
          { tok, s } = token(s);
          //Convert the string in 'tok' to a floating
          //point number and add it to 'sum'
          sum = sum + stof(tok);
       endo;

       //Set the formatting for print statements to
       //create 1 space between numbers and
       //to print 2 digits after the decimal point
       format /rd 1,2;
       print "Sum is: " sum;
    endp;
```

If you type:

```
    //Since it is a 'keyword' and not a 'proc', 'add'
    //will take everything between 'add' and the
    //semi-colon as a string input and refer to it
    //internally as the 's' variable
    add 1 2 3 4 5 6;
```

add will respond:

```
    Sum is: 15.00
```

Source

token.src

See Also

parse

topolar

Purpose

Converts from Cartesian to polar coordinates.

Format

{ *r*, *theta* } = **topolar**(*xy*);

Input

xy	NxK complex matrix containing the *x* coordinate in the real part and the *y* coordinate in the imaginary part.

Output

r	NxK real matrix, radius.
theta	NxK real matrix, angle in radians.

Source

`coord.src`

trace

Purpose

Allows the user to trace program execution for debugging purposes.

Format

```
    trace new;
    trace new, mask;
```

Input

new	scalar, new value for trace flag.
mask	scalar, optional mask to allow leaving some bits of the trace flag unchanged.

Remarks

The `trace` command has no effect unless you are running your program under **GAUSS**'s source level debugger. Setting the `trace` flag will not generate any debugging output during normal execution of a program.

The argument is converted to a binary integer with the following meanings:

bit	decimal	meaning
ones	1	trace calls/returns
twos	2	trace line numbers
fours	4	unused
eights	8	output to window
sixteens	16	output to print
thirty-twos	32	output to auxiliary output
sixty-fours	64	output to error log

You must set one or more of the output bits to get any output from `trace`. If you set `trace` to 2, you'll be doing a line number trace of your program, but the output will not be displayed anywhere.

The `trace` output as a program executes will be as follows:

(+GRAD)	calling function or procedure **GRAD**
(-GRAD)	returning from **GRAD**
[47]	executing line 47

Note that the line number trace will only produce output if the program was compiled with line number records.

To set a single bit use two arguments:

`trace` 16,16;	turn on output to printer
`trace` 0,16;	turn off output to printer

Example

```
     trace 1+8;     //trace fn/proc calls/returns to standard
                    //output
     trace 2+8;     //trace line numbers to standard output
     trace 1+2+8;   //trace line numbers and fn/proc calls/
                    //returns to standard output
     trace 1+16;    //trace fn/proc calls/returns to printer
     trace 2+16;    //trace line numbers to printer
     trace 1+2+16;  //trace line numbers and fn/proc calls/
                    //returns to printer
```

See Also

lineson

trap

Purpose

Sets the trap flag to enable or disable trapping of numerical errors.

Format

```
trap new;
trap new, mask;
```

Input

new	scalar, new trap value.
mask	scalar, optional mask to allow leaving some bits of the trap flag unchanged.

Remarks

The trap flag is examined by some functions to control error handling. There are 16 bits in the trap flag, but most **GAUSS** functions will examine only the lowest order bit:

trap 1;	turn trapping on
trap 0;	turn trapping off

If we extend the use of the trap flag, we will use the lower order bits of the trap flag. It would be wise for you to use the highest 8 bits of the trap flag if you create some sort of user-defined trap mechanism for use in your programs. (See the function **trapchk** for detailed instructions on testing the state of the trap flag; see **error** for generating user-defined error codes.)

To set only one bit and leave the others unchanged, use two arguments:

trap 1,1;	set the ones bit
trap 0,1;	clear the ones bit

Example

```
proc(0) = printinv(x);
   local oldval,y;
   oldval = trapchk(1);
   trap 1,1;
   y = inv(x);
   trap oldval,1;
   if scalerr(y);
      errorlog "WARNING: x is singular";
   else;
      print "y" y;
   endif;
endp;
```

In this example the result of **inv** is trapped in case *x* is singular. The trap state is reset to the original value after the call to **inv**.

Calling **printinv** as follows:

```
x = eye(3);
printinv(x);
```

produces:

```
y =
    1.0000000 0.0000000 0.0000000
    0.0000000 1.0000000 0.0000000
    0.0000000 0.0000000 1.0000000
```

while

```
    x = ones(3,3);
    printinv(x);
```

produces:

```
        WARNING: x is singular
```

See Also

scalerr, trapchk, error

trapchk

Purpose

Tests the value of the trap flag.

Format

y = **trapchk**(*m*);

Input

m	scalar mask value.

Output

y	scalar which is the result of the bitwise logical AND of the trap flag and the mask value.

Remarks

To check the various bits in the trap flag, add the decimal values for the bits you wish to check according to the chart below and pass the sum in as the argument to the **`trapchk`** function:

bit	decimal value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768

If you want to test if either bit 0 or bit 8 is set, then pass an argument of 1+256 or 257 to **`trapchk`**. The following table demonstrates values that will be returned for:

```
    y=trapchk(257);
```

	0	1	value of bit 0 in trap flag
0	0	1	
1	256	257	
value of bit 8 in trap flag			

GAUSS functions that test the trap flag currently test only bits 0 and 1.

See Also

scalerr, trap, error

trigamma

Purpose

Computes trigamma function.

Format

y = **`trigamma`**(*x*);

Input

x	MxN matrix or N-dimensional array.

Output

y	MxN matrix or N-dimensional array, trigamma.

Remarks

The trigamma function is the second derivative of the log of the gamma function with respect to its argument.

trimr

Purpose

Trims rows from the top and/or bottom of a matrix.

Format

```
y = trimr(x, t, b);
```

Input

x	NxK matrix from which rows are to be trimmed.
t	scalar containing the number of rows which are to be removed from the top of *x*.
b	scalar containing the number of rows which are to be removed from the bottom of *x*.

Output

y	RxK matrix where R=N-(*t* + *b*), containing the rows left after the trim.

Remarks

If either *t* or *b* is zero, then no rows will be trimmed from that end of the matrix.

Example

```
//Create a 5x3 matrix of random uniform numbers
x = rndu(5,3);

//Remove the top 2 rows of x and the bottom row
y = trimr(x,2,1);
```

If *x* is equal to:

```
0.780 0.922 0.864
0.151 0.687 0.947
0.271 0.014 0.060
0.054 0.084 0.526
0.880 0.278 0.199
```

then *y* will equal:

```
0.271 0.014 0.060
0.054 0.084 0.526
```

See Also

submat, rotater, shiftr

trunc

Purpose

Converts numbers to integers by truncating the fractional portion.

Format

y = **trunc**(*x*);

Input

x	NxK matrix or N-dimensional array.

Output

y	NxK matrix or N-dimensional array containing the truncated elements of *x*.

Example

```
    x = 100*rndn(2,2);
    y = trunc(x);
```

If *x* equals:

```
    -153.373  -1.972
     109.412 127.732
```

then, *y* will equal:

```
    -153.000  -1.000
     109.000 127.000
```

See Also

ceil, floor, round

type

Purpose

Returns the symbol table type of its argument.

Format

```
t = type(x);
```

Input

x	local or global symbol, can be an expression.

Output

t scalar, argument type.

6	matrix
13	string
15	string array
17	structure
21	array
23	structure pointer
23	sparse matrix

Remarks

type returns the type of a single symbol. The related function **typecv** will take a character vector of symbol names and return a vector of either their types or the missing value code for any that are undefined. **type** works for the symbol types listed above; **typecv** works for user-defined procedures, keywords and functions as well. **type** works for global or local symbols; **typecv** works only for global symbols.

Example

```
    k = { "CHARS" };
    print k;
    iftype(k) == 6;
        k = "" $+ k; /* force matrix to string */
    endif;

    //The '$' in front of 'k' tells GAUSS to interpret it as
    //character data
    print $k;
```

produces:

```
    CHARS
```

See Also

typecv, typef

typecv

Purpose

Returns the symbol table type of objects whose names are given as a string or as elements of a character vector or string array.

Format

```
y = typecv(x);
```

Input

x	string, or Nx1 character vector or string array which contains the names of variables whose type is to be determined.

Output

y	scalar or Nx1 vector containing the types of the respective symbols in *x*.

Remarks

The values returned by **typecv** for the various variable types are as follows:

5	keyword (`keyword`)
6	matrix (numeric, character, or mixed)
8	procedure (`proc`)
9	function (`fn`)
13	string
15	string array
17	structure
21	array
23	structure pointer

typecv will return the **GAUSS** missing value code if the symbol is not found, so it may be used to determine if a symbol is defined or not.

Example

```
xvar = sqrt(5);
yvar = "betahat";
fn area(r) = pi*r*r;
let names = xvar yvar area;
y = typecv(names);
```

This code assigns the following to *y*:

```
     6  //6 for type matrix
y = 13  //13 for string
     9  //9 for function
```

See Also

type, typef, varput, varget

typef

Purpose

Returns the type of data (the number of bytes per element) in a **GAUSS** data set.

Format

y = **typef**(*fp*);

Input

fp	scalar, file handle of an open file.

Output

y	scalar, type of data in **GAUSS** data set.

Remarks

If *fp* is a valid **GAUSS** file handle, then *y* will be set to the type of the data in the file as follows:

2	2-byte signed integer
4	4-byte IEEE floating point
8	8-byte IEEE floating point

Example

```
//Assign a variable to represent each of our file names
infile = "dat1";
outfile = "dat2";

//Open the file "dat1" for reading.
//Note: The ^ before 'infile' tells GAUSS to use the value
 //of the string variable 'infile' (which is 'dat1' in this
//case) rather than name of the variable.
open fin = ^infile;

//Get the names of the variables that are saved in the
//dataset
```

```
    names = getname(infile);

    //Create a new data set file using the same variable names
    //as 'dat1', with 1 column per data element and using the
    //same size data, i.e. the number of bytes per element, as
    //the data in 'dat1'
    create fout = ^outfile with ^names, 0, typef(fin);
```

In this example, a file `dat2.dat` is created which has the same variables and variable type as the input file, `dat1.dat`. **`typef`** is used to return the type of the input file data for the `create` statement.

See Also

colsf, rowsf

union

Purpose

Returns the union of two vectors with duplicates removed.

Format

```
y = union(v1, v2, flag);
```

Input

v1	Nx1 vector.
v2	Mx1 vector.
flag	scalar, 1 if numeric data, 0 if character.

Output

y	Lx1 vector containing all unique values that are in *v1* and *v2*, sorted in ascending order.

Remarks

The combined elements of *v1* and *v2* must fit into a single vector.

Example

```
//Create two column vectors with character data
let v1 = mary jane linda john;
let v2 = mary sally;

x = union(v1,v2,0);

//The '$' in front of 'x' tells GAUSS to print 'x' as
//character data
print $x;
```

The above code will produce the following results:

```
 JANE
 JOHN
LINDA
 MARY
SALLY
```

unionsa

Purpose

Returns the union of two string vectors with duplicates removed.

Format

y = **unionsa**(*sv1*, *sv2*);

Input

sv1	Nx1 or 1xN string vector.
sv2	Mx1 or 1xM string vector.

Output

y	Lx1 vector containing all unique values that are in *sv1* and *sv2*, sorted in ascending order.

Example

```
    string sv1 = { "mary", "jane", "linda", "john" };
    string sv2 = { "mary", "sally" };
    y = unionsa(sv1,sv2);
    print y;
```

The above code produces the following output:

```
     jane
     john
    linda
     mary
    sally
```

Source

unionsa.src

See Also

union

uniqindx

Purpose

Computes the sorted index of *x*, leaving out duplicate elements.

Format

```
index = uniqindx(x, flag);
```

Input

x	Nx1 or 1xN vector.
flag	scalar, 1 if numeric data, 0 if character.

Output

index	Mx1 vector, indices corresponding to the elements of *x* sorted in ascending order with duplicates removed.

Remarks

Among sets of duplicates it is unpredictable which elements will be indexed.

Example

```
let x = 5 4 4 3 3 2 1;

//Create a sorted index of all the unique elements in 'x'
```

```
    ind = uniqindx(x,1);

    //Use the index 'ind' to return all of the unique elements
    //of 'x' in ascending order
    y = x[ind];
```

After running the above code, *ind* and *y* are equal to:

```
            7.0000000          1.0000000
            6.0000000          2.0000000
    ind =   4.0000000    y =   3.0000000
            3.0000000          4.0000000
            1.0000000          5.0000000
```

See Also

unique, uniqindxsa

uniqindxsa

Purpose

Computes the sorted index of a string vector, omitting duplicate elements.

Format

ind = **uniqindxsa**(*sv*);

Input

sv	Nx1 or 1xN string vector.

Output

ind	Mx1 vector, indices corresponding to the elements of *sv* sorted in ascending order with duplicates removed.

Remarks

Among sets of duplicates it is unpredictable which elements will be indexed.

Example

```
    string sv = {"mary", "linda", "linda", "jane",
                 "jane", "cindy", "betty"};
    ind = uniqindxsa(sv);
    y = sv[ind];
```

The above code assigns the variables *ind* and *y* as follows:

```
          7       betty
          6       cindy
    ind = 4   y =  jane
          2       linda
          1        mary
```

Source

uniquesa.src

See Also

unique, uniquesa, uniqindx

unique

Purpose

Sorts and removes duplicate elements from a vector.

Format

y = **unique**(*x*, *flag*);

Input

x	Nx1 or 1xN vector.
flag	scalar, 1 if numeric data, 0 if character.

Output

y	Mx1 vector, sorted *x* with the duplicates removed.

Example

```
//Create a column vector with duplicate elements
let eventYear = 1632 2012 1709 1812 1709 1989 1830 1875 191
2 1912 1924 1960;

//Sort 'eventYear' as numeric data and remove any duplicate
//elements
years = unique(eventYear,1);
```

After the code above, the variables *eventYear* and *years* are assigned as follows:

```
                1632
                2012          1632
                1709          1709
                1812          1812
                1709          1830
    eventYear = 1989   year = 1875
                1830          1912
                1875          1924
                1912          1960
                1912          1989
                1924          2012
                1960
```

See Also

uniquesa, uniqindx

uniquesa

Purpose

Removes duplicate elements from a string vector.

Format

y = **uniquesa**(*sv*);

Input

sv	Nx1 or 1xN string vector.

Output

y	sorted Mx1 string vector containing all unique elements found in *sv*.

Example

```
//Create a 8x1 string array
string comTrades = { "corn", "gold", "soybeans", "silver",
"coffee",
                     "oil", "silver", "soybeans" };

//Return an alphabetized string array containing the
//unique elements from 'comTrades'
commodity = uniquesa(comTrades);
```

After the code above, the variables *comTrades* and *commodity* will be equal to:

```
                     corn
                     gold                  coffee
                 soybeans                    corn
comTrades =        silver   commodity =      gold
                   coffee                     oil
                      oil                  silver
                   silver                soybeans
                 soybeans
```

Remarks

It is important to note that the return from **uniquesa** will always be a column vector, even if the input string array is a row vector.

Source

`uniquesa.src`

See Also

unique, uniqindxsa, uniqindx

upmat, upmat1

Purpose

Returns the upper portion of a matrix. **`upmat`** returns the main diagonal and every element above. **`upmat1`** is the same except it replaces the main diagonal with ones.

Format

u = **`upmat`**(*x*);
u = **`upmat1`**(*x*);

Input

x	NxK matrix.

Output

u	NxK matrix containing the upper elements of *x*. The lower elements are replaced with zeros. **`upmat`** returns the main diagonal intact. **`upmat1`** replaces the main diagonal with ones.

Example

```
x = { 7  2 -1,
      2  3 -2,
      4 -2  8 };

u = upmat(x);
u1 = upmat1(x);
```

The resulting matrices are:

```
      7  2 -1           1  2 -1
u = 0  3 -2    u1 = 0  1 -2
      0  0  8           0  0  1
```

Source

`diag.src`

See Also

lowmat, lowmat1, diag, diagrv, crout

upper

Purpose

Converts a string, matrix of character data, or string array to uppercase.

Format

y = **upper**(x);

Input

x	string, or NxK matrix, or string array containing the character data to be converted to uppercase.

Output

y	string, or NxK matrix, or string array containing the uppercase equivalent of the data in *x*.

Remarks

If *x* is a numeric matrix, *y* will contain garbage. No error message will be generated since **GAUSS** does not distinguish between numeric and character data in matrices.

Example

```
//Create a lowercase string
x = "uppercase";

//Convert the string to upper case
y = upper(x);

//Adding the '$' tells GAUSS to treat the data as character
//data
print $y;
```

This code produces:

```
UPPERCASE
```

See Also

lower

use

Purpose

Loads a compiled file at the beginning of the compilation of a source program.

Format

```
use fname;
```

Input

fname	literal or ^string, the name of a compiled file created using the `compile` or the `saveall` command.

Remarks

The `use` command can be used ONCE at the TOP of a program to load in a compiled file which the rest of the program will be added to. In other words, if `xy.e` had the following lines:

```
    library pgraph;
    externalproc xy;
    x = seqa(0.1,0.1,100);
```

it could be compiled to `xy.gcg`. Then the following program could be run:

```
    use xy;
    xy(x, sin(x));
```

which would be equivalent to:

```
    new;
    library pgraph;
    x = seqa(0.1,0.1,100);
    xy(x, sin(x));
```

The `use` command can be used at the top of files that are to be compiled with the `compile` command. This can greatly shorten compile time for a set of closely related programs. For example:

```
    library pgraph;
    externalproc xy,logx,logy,loglog,hist;
    saveall pgraph;
```

This would create a file called `pgraph.gcg` containing all the procedures, strings and matrices needed to run PQG programs. Other programs could be compiled very quickly with the following statement at the top of each:

```
    use pgraph;
```

or the same statement could be executed once, for instance from the command prompt, to instantly load all the procedures for PQG.

When the compiled file is loaded with `use`, all previous symbols and procedures are deleted before the program is loaded. It is therefore unnecessary to execute a `new` before `use`'ing a compiled file.

`use` can appear only ONCE at the TOP of a program.

See Also

compile, run, saveall

utctodt

Purpose

Converts UTC scalar format to DT scalar format.

Format

dt = **utctodt**(*utc*);

Input

utc	Nx1 vector, UTC scalar format.

Output

dt	Nx1 vector, DT scalar format.

Remarks

A UTC scalar gives the number of seconds since or before January 1, 1970 Greenwich Mean Time. In DT scalar format, 08:35:52 on June 11, 2005 is 20050611083552.

Example

```
tc = 1346290409;
print "tc = " tc;
dt = utctodt(tc);
print "dt = " dt;
```

produces:

```
    tc = 1346290409
    dt = 20120829183329
```

Source

`time.src`

See Also

dtvnormal, timeutc, utctodtv, dttodtv, dtvtodt, dttoutc, dtvtodt, strtodt, dttostr

utctodtv

Purpose

Converts UTC scalar format to DTV vector format.

Format

dtv = **utctodtv**(*utc*);

Input

utc	Nx1 vector, UTC scalar format.

Output

dtv	Nx8 matrix, DTV vector format.

Remarks

A UTC scalar gives the number of seconds since or before January 1, 1970 Greenwich Mean Time.

Each row of *dtv*, in DTV vector format, contains:

[N,1]	Year, four digit integer.
[N,2]	Month in Year, 1-12.
[N,3]	Day of month, 1-31.
[N,4]	Hours since midnight, 0-23.
[N,5]	Minutes, 0-59.
[N,6]	Seconds, 0-59.
[N,7]	Day of week, 0-6, 0=Sunday.
[N,8]	Days since Jan 1 of current year, 0-365.

Example

```
//Set 'tc' equal to the number of seconds since January 1,
//1970
tc = timeutc;
print "tc = " tc;

dtv = utctodtv(tc);
print "dtv = " dtv;
```

produces:

```
tc = 1340315529
dtv = 2012 6 21 14 52 9 4 172
```

See Also

dtvnormal, timeutc, utctodt, dttodtv, dttoutc, dtvtodt, dtvtoutc, strtodt, dttostr

utrisol

Purpose

Computes the solution of $Ux = b$ where U is an upper triangular matrix.

Format

```
x = utrisol(b, U);
```

Input

b	PxK matrix.
U	PxP upper triangular matrix.

Output

x	PxK matrix, solution of $Ux = b$.

Remarks

utrisol applies a back solve to $Ux = b$ to solve for x. If b has more than one column, each column is solved for separately, i.e., **utrisol** applies a back solve to $U * x[.,i] = b[.,i]$.

V

vals

Purpose

Converts a string into a matrix of its ASCII values.

Format

y = **`vals`**(*s*);

Input

s	string of length N where N > 0.

Output

y	Nx1 matrix containing the ASCII values of the characters in the string *s*.

Remarks

If the string is null, the function will fail and an error message will be given.

Example

```
//Initialize 'k' so it will be 0 for the first iteration of
//the 'do while' loop
k = 0;

//Prompt the user for input
print"Continue Program? [Y/N]";

//Continually check for keyboard input and exit the loop on
//keyboard input
do while (k == 0);
   k = key;
endo;

//Follow a different code branch depending upon which key
//the user entered
if k == vals("Y") or k == vals("y");
   print "You chose to continue";
else;
   print "Exiting program now";
endif;
```

In this example the **key** function is used to read keyboard input. When **key** returns a nonzero value, meaning a key has been pressed, the ASCII value it returns is tested to see if it is an uppercase or lowercase 'Y'. If it is, the program will follow the first branch and print:

```
You chose to continue
```

otherwise, it will follow the second branch and print:

```
Exiting program now
```

See Also

chrs, ftos, stof

varCovM, varCovX

Purpose

Computes the population variance-covariance matrix.

Format

```
vc = varCovM(mm);
vc = varCovX(x);
```

Input

mm	KxK moment ($x'x$) matrix. A constant term MUST have been the first variable when the moment matrix was computed.
x	NxK matrix of data.

Output

vc	KxK variance-covariance matrix.

Example

```
//Set rndseed for repeatable random numbers
rndseed 7234242;

//Create three randomly generated independent variables
x = rndn(500, 3);

//Create the population variance-covariance matrix from dat
a matrix 'x'
var_x = varCovX(x);
```

After the code above, *var_x* will be equal to:

```
   1.0941806   0.0040829  -0.0024871
   0.0040829   1.0606611   0.0493555
  -0.0024871   0.0493555   0.8729622
```

where the diagonal elements in the matrix represent the population variance of the each column, while the off-diagonal elements represent the population covariance between the data columns.

The population variance can also be calculated using the moment matrix, *x'x* and the GAUSS function **varCovM**. A constant term must be included in the data matrix *x* when computing the moment equation. Consider the following data matrix *x1*, consisting of the original data matrix *x* and a column of ones:

```
//Set rndseed so 'rndn' will return the same numbers as abo
ve
rndseed 7234242;

//Note: the ~ operator performs horizontal concatenation
x1 = ones(500,1)~rndn(500,3);
```

```
//Create moment matrix
x2 = x1'x1;

//Calculate variance-covariance matrix using the moment mat
rix
var_xm = varCovM(x2);
```

After the code above, *var_xm* will be equal to:

```
   1.0941806   0.0040829  -0.0024871
   0.0040829   1.0606611   0.0493555
  -0.0024871   0.0493555   0.8729622
```

Remarks

The variance covariance matrix is that of the population data matrix. It is computed as the moment matrix of deviations about the mean divided by the number of observations *N*. For a sample covariance matrix which uses *N - 1* rather than *N* see **varCovMS** or **varCovXS**.

Source

`corrs.src`

See Also

momentd, corrms, corrxs

varCovMS, varCovXS

Purpose

Computes a sample variance-covariance matrix.

Format

vc = **varCovMS**(*mm*);
vc = **varCovXS**(*x*);

Input

mm	KxK moment (*x'x*) matrix. A constant term MUST have been the first variable when the moment matrix was computed.
x	NxK matrix of data.

Output

vc	KxK variance-covariance matrix.

Example

```
//Set rndseed for repeatable random numbers
rndseed 7234242;

//Create three randomly generated independent variables
x = rndn(500, 3);

//Create the sample variance-covariance matrix from data ma
trix 'x'
var_x = varCovXS(x);
```

After the code above, `var_x` will be equal to:

```
        1.0963733    0.0040911   -0.0024921
        0.0040911    1.0627867    0.0494544
       -0.0024921    0.0494544    0.8747116
```

where the diagonal elements in the matrix represent the sample variance of the each column, while the off-diagonal elements represent the sample covariance between the data columns.

The sample variance can also be calculated using the moment matrix, `x'x` and the GAUSS function **`varCovMS`**. A constant term must be included in the data matrix *x* when computing the moment equation. Consider the following data matrix *x1*, consisting of the original data matrix *x* and a column of ones:

```
    //Set rndseed so 'rndn' will return the same numbers as abo
    ve
    rndseed 7234242;

    //Note: the ~ operator performs horizontal concatenation
    x1 = ones(500,1)~rndn(500,3);

    //Create moment matrix
    x2 = x1'x1;

    //Calculate variance-covariance matrix using the moment mat
    rix
    var_xm = varCovMS(x2);
```

After the code above, *var_xm* will be equal to:

```
        1.0963733    0.0040911   -0.0024921
        0.0040911    1.0627867    0.0494544
       -0.0024921    0.0494544    0.8747116
```

Remarks

The variance covariance matrix is that of the sample data matrix. It is computed as the moment matrix of deviations about the mean divided by the number of observations minus one, *N - 1*. For a population covariance matrix which uses *N* rather than *N - 1* see **varCovM** or **varCovX**.

Source

corrs.src

See Also

momentd, corrms, corrxs

varget

Purpose

Accesses a global variable whose name is given as a string argument.

Format

y = **varget**(*s*);

Input

s	string containing the name of the global symbol you wish to access.

Output

y	contents of the variable whose name is in *s*.

Remarks

This function searches the global symbol table for the symbol whose name is in *s* and returns the contents of the variable if it exists. If the symbol does not exist, the function will terminate with an Undefined symbol error message. If you want to check to see if a variable exists before using this function, use **typecv**.

Example

```
alpha = 1;
beta = 2;
letter = "alpha";

//Check to see if a variable named alpha exists
if typecv(letter) == miss(0,0);
   print letter " does NOT exist";
else;
   //Assign the value of the variable named alpha to 'tmp'
   tmp = varget(letter);
   print "the value of " letter " is: " tmp;
endif;
```

The code above produces the following output:

```
the value of alpha is: 1
```

vargetl

Purpose

Accesses a local variable whose name is given as a string argument.

Format

y = **vargetl**(*s*);

Input

s	string containing the name of the local symbol you wish to access.

Output

y	contents of the variable whose name is in *s*.

Remarks

This function searches the local symbol list for the symbol whose name is in *s* and returns the contents of the variable if it exists. If the symbol does not exist, the function will terminate with an Undefined symbol error message.

Example

```
proc rndNormEx( r, c, loc, std, ptVar);
   local rnd1, rnd2, rnd3;
```

```
        //Create random normal numbers with mean 0 and standard
        //deviation 1
        rnd1 = rndn(r, c);

        //Change the mean to 'loc'
        rnd2 = rnd1 + loc;

        //Change the standard deviation to 'std'
        rnd3 = std * rnd2;

        //Set the contents of tmp to be equal to the contents of
        //the local variable with the same name as the string
        //passed in as 'ptVar'
        tmp = vargetl(ptVar);

        print ptVar " is equal to: " tmp;

        retp(rnd3);
    endp;

    //Set the rng seed for repeatable results
    rndseed 54223423;

    //Passing in the final variable as the string rnd1, will
    //cause the proc rndNormEx to print the contents of rnd1
    r = rndNormEx( 2, 2, 0, 3, "rnd1");
```

The code above will produce the following output:

```
    rnd1 is equal to:
     0.5240627925408163  1.4904799236486497
    -1.1716182730350617 -0.0519353312479753
```

See Also

varputl

varmall

Purpose

Computes log-likelihood of a Vector ARMA model.

Format

```
ll = varmall(w, phi, theta, vc);
```

Input

w	NxK matrix, time series.
phi	(K*P)xK matrix, AR coefficient matrices.
theta	(K*Q)xK matrix, MA coefficient matrices.
vc	KxK matrix, covariance matrix.

Output

ll scalar, log-likelihood. If the calculation fails *ll* is set to missing value with error code:

Error Code	Reason for Failure
1	M < 1
2	N < 1
3	P < 0

4	Q < 0
5	P = 0 and Q = 0
7	floating point work space too small
8	integer work space too small
9	vc is not positive definite
10	AR parameters too close to stationarity boundary
11	model not stationary
12	model not invertible
13	I+M'H'HM not positive definite

Remarks

`varmall` is adapted from code developed by Jose Alberto Mauricio of the Universidad Complutense de Madrid. It was published as Algorithm AS311 in Applied Statistics. Also described in "Exact Maximum Likelihood Estimation of Stationary Vector ARMA Models," JASA, 90:282-264.

varmares

Purpose

Computes residuals of a Vector ARMA model.

Format

```
res = varmares(w, phi, theta);
```

Input

`w`	NxK matrix, time series.
`phi`	(K*P)xK matrix, AR coefficient matrices.
`theta`	(K*Q)xK matrix, MA coefficient matrices.

Output

`res` NxK matrix, residuals. If the calculation fails `res` is set to missing value with error code:

Error Code	Reason for Failure
1	M < 1
2	N < 1
3	P < 0
4	Q < 0
5	P = 0 and Q = 0
7	floating point work space too small
8	integer work space too small
10	AR parameters too close to stationarity boundary
11	model not stationary
12	model not invertible
13	I+M'H'HM not positive definite

Remarks

`varmares` is adapted from code developed by Jose Alberto Mauricio of the Universidad Complutense de Madrid. It was published as Algorithm AS311 in Applied Statistics. Also described in "Exact Maximum Likelihood Estimation of Stationary Vector ARMA Models," JASA, 90:282-264.

varput

Purpose

Allows a matrix, array, string, or string array to be assigned to a global symbol whose name is given as a string argument.

Format

y = **`varput`**(*x*, *n*);

Input

x	matrix, array, string, or string array which is to be assigned to the target variable.
n	string containing the name of the global symbol which will be the target variable.

Output

y	scalar, 1 if the operation is successful and 0 if the operation fails.

Remarks

x and *n* may be global or local. The variable, whose name is in *n*, that *x* is assigned to is always a global.

If the function fails, it will be because the global symbol table is full.

This function is useful for returning values generated in local variables within a procedure to the global symbol table.

Example

```
    source = rndn(2,2);
    targname = "target";

    if notvarput(source,targname);
       print "Symbol table full";
       end;
    endif;
```

See Also

varget, typecv

varputl

Purpose

Allows a matrix, array, string, or string array to be assigned to a local symbol given as a string argument.

Format

y = **varputl**(*x*, *n*);

Input

x	matrix, array, string, or string array which is to be assigned to the target variable.
n	string containing the name of the local symbol which will be the target variable.

Output

y	scalar, 1 if the operation is successful and 0 if the operation fails.

Remarks

x and *n* may be global or local. The variable, whose name is in *n*, that *x* is assigned to is always a local.

Example

```
proc myProc(x);
   local a,b,c,d,e,vars,putvar;
   a=1;b=2;c=3;d=5;e=7;
   vars = { a b c d e };
   putvar = 0;

   //Keep looping until the user enters a letter
```

```
        //a-e or A-E
        do while putvar $/= vars;
           //Two semi-colons at the end of a print statement,
           //prevents a 'new line' from being printed
           print "Assign x (" $vars "): ";;
           putvar = upper(cons);
           print;
        endo;

        //Assign the variable whose letter/name was entered by
        //the user to be the value passed into 'myProc'
        call varputl(x,putvar);
        retp(a+b*c-d/e);
     endp;

     //Format printing of numbers to allow 2 spaces between them
     //and 1 digit after the decimal place
     format /rds 2,1;

     z = myProc(17);
     print " z is " z;
```

produces (Note: this program will ask for user input at the GAUSS command prompt):

```
     Assign x ( A   B   C   D   E ): a

      z is 22.3
```

See Also

vargetl

vartypef

Purpose

Returns a vector of ones and zeros that indicate whether variables in a data set are character or numeric.

Format

`y = vartypef(f);`

Input

f	file handle of an open file.

Output

y	Nx1 vector of ones and zeros, 1 if variable is numeric, 0 if character.

Remarks

This function should be used in place of older functions that are based on the case of the variable names. You should also use the *v96* data set format.

vcm, vcx

Purpose

Computes an unbiased estimate a variance-covariance matrix.

NOTE: **vcm** and **vcx** have been replaced with functions **varCovXS** and **varCovMS** whose descriptions use more standard statistical nomenclature. **vcx** and **vcm** will continue to be available for backwards compatibility.

Format

vc = **vcm**(*m*);
vc = **vcx**(*x*);

Input

m	KxK moment (*x*'*x*) matrix. A constant term MUST have been the first variable when the moment matrix was computed.
x	NxK matrix of data.

Output

vc	KxK variance-covariance matrix.

Remarks

The variance-covariance matrix is computed as an unbiased estimator of the population variance-covariance. It is computed as the moment matrix of deviations about the mean divided by the number of observations minus one, *N - 1*. For an observed variance-covariance matrix which uses *N* rather than `N - 1` see **`vcms`** or **`vcxs`**.

Source

```
corr.src
```

See Also

momentd

vcms, vcxs

Purpose

Computes the observed variance-covariance matrix.

NOTE: **`vcms`** and **`vcxs`** have been replaced with functions **`varCovX`** and **`varCovM`** whose descriptions use more standard statistical nomenclature. **`vcxs`** and **`vcms`** will continue to be available for backwards compatibility.

Format

vc = **`vcms`**(*m*);
vc = **`vcxs`**(*x*);

Input

m	KxK moment ($x'x$) matrix. A constant term MUST have been the first variable when the moment matrix was computed.

x	NxK matrix of data.

Output

vc	KxK variance-covariance matrix.

Remarks

The variance covariance matrix is that of the input data matrix. It is computed as the moment matrix of deviations about the mean divided by the number of observations *N*. For an unbiased estimator covariance matrix which uses *N - 1* rather than *N* see **vcm** or **vcx**.

Source

`corrs.src`

See Also

momentd, corrms, corrxs

vec, vecr

Purpose

Creates a column vector by appending the columns/rows of a matrix to each other.

Format

yc = **vec**(*x*);
yr = **vecr**(*x*);

Input

x	NxK matrix.

Output

yc	(N*K)x1 vector, the columns of *x* appended to each other.
yr	(N*K)x1 vector, the rows of *x* appended to each other and the result transposed.

Remarks

vecr is much faster.

Example

```
x = { 1 2,
      3 4 };
yc = vec(x);
yr = vecr(x);
```

The code above assigns the variables *yc* and *yr*:

```
     1        1
yc = 3   yr = 2
```

```
       2         3
       4         4
```

vech

Purpose

Vectorizes a symmetric matrix by retaining only the lower triangular portion of the matrix.

Format

v = **vech**(*x*);

Input

x	NxN symmetric matrix.

Output

v	(N*(N+1)/2)x1 vector, the lower triangular portion of the matrix *x*.

Remarks

As you can see from the example below, **vech** will not check to see if *x* is symmetric. It just packs the lower trangular portion of the matrix into a column vector in row-wise order.

Example

```
//Add a 3x1 column vector containing 10, 20, 30 to a 1x3
//row vector containing 1, 2, 3, to create a 3x3 matrix
x = seqa(10,10,3) + seqa(1,1,3)';

//Turn the lower triangular portion of 'x' into a column
//vector in 'v'
v = vech(x);

//Expand the vector 'v' into a symmetric matrix in 'sx'
sx = xpnd(v);
```

After the code above:

```
                        11
    11  12  13          21          11  21  31
x = 21  22  23      v = 22    sx =  21  22  32
    31  32  33          31          31  32  33
                        32
                        33
```

See Also

xpnd

vector (dataloop)

Purpose

Specifies the creation of a new variable within a data loop.

Format

```
vector # numvar = numeric_expression;
vector $ charvar =  character_expression;
```

Remarks

A `numeric_expression` is any valid expression returning a numeric value. A `character_expression` is any valid expression returning a character value. If neither '$' nor '#' is specified, '#' is assumed.

`vector` is used in place of `make` when the expression returns a scalar rather than a vector. `vector` forces the result of such an expression to a vector of the correct length. `vector` could actually be used anywhere that `make` is used, but would generate slower code for expressions that already return vectors.

Any variables referenced must already exist, either as elements of the source data set, as `extern`'s, or as the result of a previous `make`, `vector`, or `code` statement.

Example

```
        vector const = 1;
```

See Also

make (dataloop)

vget

Purpose

Extracts a matrix or string from a data buffer constructed with **`vput`**.

Format

```
{ x, dbufnew } = vget(dbuf, name);
```

Input

dbuf	Nx1 vector, a data buffer containing various strings and matrices.
name	string, the name of the string or matrix to extract from *dbuf*.

Output

x	LxM matrix or string, the item extracted from *dbuf*.
dbufnew	Kx1 vector, the remainder of *dbuf* after *x* has been extracted.

Source

`pack.src`

See Also

vlist, vput, vread

view

Purpose

Sets the position of the observer in workbox units for 3-D plots. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

`view`(*x*, *y*, *z*);

Input

x	scalar, the X position in workbox units.
y	scalar, the Y position in workbox units.
z	scalar, the Z position in workbox units.

Remarks

The size of the workbox is set with **`volume`**. The viewer MUST be outside of the workbox. The closer the position of the observer, the more perspective distortion there will be. If $x = y =\ \ z$, the projection will be isometric.

If **`view`** is not called, a default position will be calculated.

Use **`viewxyz`** to locate the observer in plot coordinates.

Source

`pgraph.src`

See Also

volume, viewxyz

viewxyz

Purpose

To set the position of the observer in plot coordinates for 3-D plots. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

`viewxyz`(*x*, *y*, *z*);

Input

x	scalar, the X position in plot coordinates.
y	scalar, the Y position in plot coordinates.
z	scalar, the Z position in plot coordinates.

Remarks

The viewer MUST be outside of the workbox. The closer the observer, the more perspective distortion there will be.

If **`viewxyz`** is not called, a default position will be calculated.

Use **`view`** to locate the observer in workbox units.

Source

`pgraph.src`

See Also

volume, view

vlist

Purpose

Lists the contents of a data buffer constructed with **`vput`**.

Format

`vlist`(*`dbuf`*);

Input

`dbuf`	Nx1 vector, a data buffer containing various strings and matrices.

Remarks

`vlist` lists the names of all the strings and matrices stored in *`dbuf`*.

Source

`vpack.src`

See Also

vget, vput, vread

vnamecv

Purpose

Returns the names of the elements of a data buffer constructed with **`vput`**.

Format

```
cv = vnamecv(dbuf);
```

Input

`dbuf`	Nx1 vector, a data buffer containing various strings and matrices.

Output

`cv`	Kx1 character vector containing the names of the elements of *`dbuf`*.

See Also

vget, vput, vread, vtypecv

volume

Purpose

Sets the length, width, and height ratios of the 3-D workbox. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

```
volume(x, y, z);
```

Input

x	scalar, the X length of the 3-D workbox.
y	scalar, the Y length of the 3-D workbox.
z	scalar, the Z length of the 3-D workbox.

Remarks

The ratio between these values is what is important. If **volume** is not called, a default workbox will be calculated.

Source

`pgraph.src`

See Also

view

vput

Purpose

Inserts a matrix or string into a data buffer.

Format

dbufnew = **vput**(*dbuf*, *x*, *xname*);

Input

dbuf	Nx1 vector, a data buffer containing various strings and matrices. If *dbuf* is a scalar 0, a new data buffer will be created.
x	LxM matrix or string, item to be inserted into *dbuf*.
xname	string, the name of *x*, will be inserted with *x* into *dbuf*.

Output

dbufnew	Kx1 vector, the data buffer after *x* and *xname* have been inserted.

Remarks

If *dbuf* already contains *x*, the new value of *x* will replace the old one.

Source

`vpack.src`

See Also

vget, vlist, vread

vread

Purpose

Reads a string or matrix from a data buffer constructed with **vput**.

Format

x = **vread**(*dbuf*, *xname*);

Input

dbuf	Nx1 vector, a data buffer containing various strings and matrices.
xname	string, the name of the matrix or string to read from *dbuf*.

Output

x	LxM matrix or string, the item read from *dbuf*.

Remarks

vread, unlike **vget**, does not change the contents of *dbuf*. Reading *x* from *dbuf* does not remove it from *dbuf*.

Source

`vpack.src`

See Also

vget, vlist, vput

vtypecv

Purpose

Returns the types of the elements of a data buffer constructed with **vput**.

Format

cv = **vtypecv**(*dbuf*);

Input

dbuf	Nx1 vector, a data buffer containing various strings and matrices.

Output

cv	Kx1 character vector containing the types of the elements of *dbuf*.

See Also

vget, vput, vread, vnamecv

w

wait, waitc

Purpose

Waits until any key is pressed.

Format

```
wait;
waitc;
```

Remarks

If you are working in terminal mode, these commands do not "see" any keystrokes until `ENTER` is pressed. **`waitc`** clears any pending keystrokes before waiting until another key is pressed.

Source

`wait.src`, `waitc.src`

See Also

pause

walkindex

Purpose

Walks the index of an array forward or backward through a specified dimension.

Format

```
ni = walkindex(i, o, dim);
```

Input

i	Mx1 vector of indices into an array, where M <= N.
o	Nx1 vector of orders of an N-dimensional array.
dim	scalar [1-to-M], index into the vector of indices *i*, corresponding to the dimension to walk through, positive to walk the index forward, or negative to walk backward.

Output

ni	Mx1 vector of indices, the new index.

Remarks

walkindex will return a scalar error code if the index cannot walk further in the specified dimension and direction.

Example

```
orders =(3,4,5,6,7);

//Create a 3x4x5x6x7 dimensional array with each element
//equal to 1
a = arrayinit(orders,1);

ind = { 2,3,3 };
ind = walkindex(ind,orders,-2);
```

```
        2
ind = 2
        3
```

This example decrements the second value of the index vector *ind*.

```
        ind = walkindex(ind,orders,3);
```

```
        2
ind = 2
        4
```

Using the **orders** from the example above and the *ind* that was returned, this example increments the third value of the index vector *ind*.

See Also

nextindex, previousindex, loopnextindex

window

Purpose

Partitions the window into tiled regions (graphic panels) of equal size.
NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

`window`(*`row`*, *`col`*, *`typ`*);

Input

`row`	scalar, number of rows of graphic panels.
`col`	scalar, number of columns of graphic panels.
`typ`	scalar, graphic panel attribute type. If 1, the graphic panels will be transparent, if 0, the graphic panels will be nontransparent (blanked).

Remarks

The graphic panels will be numbered from 1 to (*`row`*) x (*`col`*) starting from the left topmost graphic panel and moving right.

See **`makewind`** for creating graphic panels of a specific size and position. (For more information, see **Graphic Panels**, Section 1.1.

Source

`pwindow.src`

See Also

endwind, begwind, setwind, nextwind, getwind, makewind

writer

Purpose

Writes a matrix to a **GAUSS** data set.

Format

y = **`writer`**(*fh*, *x*);

Input

fh	handle of the file that data is to be written to.
x	NxK matrix.

Output

y	scalar specifying the number of rows of data actually written to the data set.

Remarks

The file must have been opened with `create`, `open for append`, or `open for update`.

The data in *x* will be written to the data set whose handle is *fh* starting at the current pointer position in the file. The pointer position in the file will be updated, so the next call to **`writer`** will put the next block of data after the first block. (See `open` and `create` for the initial pointer positions in the file for reading and writing.)

x must have the same number of columns as the data set. **`colsf`** returns the number of columns in a data set.

`writer` returns the number of rows actually written to the data set. If *y* does not equal **`rows`**(*x*), the disk is probably full.

If the data set is not double precision, the data will be rounded as it is written out.

If the data contain character elements, the file must be double precision or the character information will be lost.

If the file being written to is the 2-byte integer data type, then missing values will be written out as -32768. These will not automatically be converted to missings on input. They can be converted with the **`miss`** function:

```
x = miss(x,-32768);
```

Trying to write complex data to a data set that was originally created to store real data will cause a program to abort with an error message. (See `create` for details on creating a complex data set.)

Example

```
create fp = data with x,10,8;

if fp == -1;
```

```
        errorlog "Can't create output file";
        end;
     endif;

     c = 0;
     do until c >= 10000;
        y = rndn(100,10);
        k = writer(fp,y);

        if k /= rows(y);
          errorlog "Disk Full";
          fp = close(fp);
          end;
        endif;

        c = c+k;
     endo;

     fp = close(fp);
```

In this example, a 10000x10 data set of Normal random numbers is written to a data set called `data.dat`. The variable names are *X01* - *X10*.

See Also

open, close, create, readr, saved, seekr

xlabel

Purpose

Sets a label for the X axis. NOTE: This function is for use with the deprecated PQG graphics, use **`plotSetXLabel`** for equivalent functionality.

Library

pgraph

Format

```
xlabel(str);
```

Input

str	string, the label for the X axis.

Source

`pgraph.src`

See Also

title, ylabel, zlabel

xlsGetSheetCount

Purpose

Gets the number of sheets in an Excel® spreadsheet.

Format

`nsheets` = **`xlsGetSheetCount`**(*`file`*);

Input

`file`	string, name of `.xls` or `.xlsx` file.

Output

`nsheets`	scalar, sheet count or an error code.

Portability

Windows, **Linux** and **Mac**

Remarks

If **`xlsGetSheetCount`** fails, it will either terminate with an error message or return a scalar error code, which can be decoded with **`scalerr`**, depending on the lowest order bit of the trap flag.

`trap 0`	Print error message and terminate program.
`trap 1`	Return scalar error code 10.

Example

Example 1

If you had an Excel file named `myfile.xlsx` in the directory `C:\mydata`, then you could determine the number of sheets in the file with the following code:

```
    nsheets = xlsGetSheetCount("C:\\mydata\\myfile.xlsx");
```

Example 2

If you do not want your program to terminate in the case of an error in this function, you can set the `trap` state as in the example below.

```
    fname = "C:\\mydata\\myfile.xlsx";

    //Turn on trap
    trap 1;

    nsheets = xlsGetSheetCount(fname);

    //Check to see if xlsGetSheetCount returned an error code
    if scalmiss(nsheets);
       //Code to execute in error case here
    endif;
```

See Also

xlsGetSheetSize, xlsGetSheetTypes, xlsMakeRange

xlsGetSheetSize

Purpose

Gets the size (rows and columns) of a specified sheet in an Excel® spreadsheet.

Format

{ *rows*, *cols* } = **xlsGetSheetSize**(*file*, *sheet*);

Input

file	string, name of `.xls` or `.xlsx` file.
sheet	scalar, sheet index (1-based).

Output

rows	scalar, number of rows.
cols	scalar, number of columns.

Portability

Windows, **Linux** and **Mac**

Remarks

If **xlsGetSheetSize** fails, it will either terminate and print an error message or return a scalar error code, which can be decoded with **scalerr**, depending on the state of the `trap` flag.

`trap 0`	Print error message and terminate program.
`trap 1`	Return scalar error code 10.

If a scalar error code is returned, both return values will be set with the error code.

Example

Example 1

If you had an Excel file named `myfile.xlsx` in the directory `C:\mydata`, then you could determine the number of rows and columns in the first sheet of this file with the following code:

```
sheetNum = 1;
{ r, c } = xlsGetSheetSize("C:\\mydata\\myfile.xlsx",
sheetNum);
```

Example 2

If you do not want your program to terminate in the case of an error in this function, you can set the `trap` state as in the example below.

```
sheetNum = 1;

//Retain the old trap value so it can be reset to its
//previous state
oldtrap = trapchk(1);

//Set trap
trap 1;

{ r, c } = xlsGetSheetSize("C:\\mydata\\myfile.xlsx",
sheetNum);

//Check to see if return value is an error code
if scalmiss(r);
   //User error handling code here
endif;
```

See Also

xlsGetSheetCount, xlsGetSheetTypes, xlsMakeRange

xlsGetSheetTypes

Purpose

Gets the cell format types of a row in an Excel® spreadsheet.

Format

nsheets = **xlsGetSheetTypes**(*file*, *sheet*, *row*);

Input

file	string, name of `.xls` or `.xlsx` file.
sheet	scalar, sheet index (1-based).
row	scalar, the row of cells to be scanned.

Output

types	1xK vector of predefined data types representing the format of each cell in the specified row. The possible types are: *0* Text *1* Numeric *2* Date

Portability

Windows, **Linux** and **Mac**

Example

For example, let us suppose that a file named `myfile.xlsx` exists in the directory `C:\mydata`. Le us further suppose that the 'A1' element is a string and the 'B1:C1' elements are numbers. The first row has no other elements. Then the code:

```
fname = "C:\\mydata\\myfile.xlsx";
sheetNum = 1;
rowNum = 1;
ctypes = xlsGetSheetTypes(fname, sheetNum, rowNum);

//Do not print any values after the decimal point
format /rd 6,0
print ctypes;
```

would produce the following output:

```
     0     1     1
```

Remarks

K is the number of columns found in the spreadsheet.

If **`xlsGetSheetTypes`** fails, it will either terminate and print an error message or return a scalar error code, which can be decoded with **`scalerr`**, depending on the state of the `trap` flag.

trap 0	Print error message and terminate program.
trap 1	Return scalar error code 10.

See Also

xlsGetSheetCount, xlsGetSheetSize, xlsMakeRange

xlsMakeRange

Purpose

Builds an Excel® range string from a row/column pair.

Format

`range` = **`xlsMakeRange`**(*`row`*, *`col`*);

Input

`row`	scalar or 2x1 vector.
`col`	scalar or 2x1 vector.

Output

range	string, an Excel®-formatted range specifier.

Portability

Available on **Windows**, **Linux** and **Mac**.

Remarks

If *row* is a 2x1 vector, it is interpreted as follows

row[1]	starting row
row[2]	ending row

If *col* is a 2x1 vector, it is interpreted as follows:

col[1]	starting column
col[2]	ending column

If **xlsMakeRange** fails, it will either terminate and print an error message or return a scalar error code, which can be decoded with **scalerr**, depending on the state of the trap flag.

trap 0	Print error message and terminate program.
trap 1	Return scalar error code 10.

Example

```
//Scalar inputs
r = 3;
```

```
c = 6;
range = xlsMakeRange(r, c);
print range;
```

produces:

```
    F3
```

```

//2x1 vector inputs
r = { 2, 37 };
c = { 3, 19 };
range = xlsMakeRange(r, c);
print range;
```

produces:

```
    C2:S37
```

See Also

xlsGetSheetCount, xlsGetSheetSize, xlsGetSheetTypes

xlsReadM

Purpose

Reads from an Excel® spreadsheet into a **GAUSS** matrix.

Format

mat = **xlsReadM**(*file*, *range*, *sheet*, *vls*);

Input

file	string, name of `.xls` or `.xlsx` file.
range	string, range to read, e.g. `a2:b20`, or the starting point of the read, e.g. `a2`.
sheet	scalar, sheet number.
vls	null string or 9x1 matrix, specifies the conversion of Excel® empty cells and special types into **GAUSS** (see Remarks). A null string results in all empty cells and special types being converted to **GAUSS** missing values.

Output

mat	matrix or a scalar error code.

Portability

Windows, **Linux** and **Mac**

The *vls* input is currently ignored on Mac and Linux. Missing values will be returned for all cells that are empty or contain errors.

Example

For example, let us suppose that a file named `myfile.xlsx` exists in the directory `C:\mydata`. Let us further suppose that the 'A2:B10' elements contain a 9x2 matrix that we would like to read into **GAUSS**. Then the code:

```
    fname = "C:\\mydata\\myfile.xlsx";
    range = "A2:B10";
    sheetNum = 1;
    rowNum = 1;
    vls = "";
    newMat = xlsReadM(fname, range, sheetNum, vls);
```

will read in the values in the specified range and assign them to `newMat`.

Remarks

If range is a null string, then by default the read will begin at cell `a1`.

The *`vls`* argument lets users control the import of Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell
1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!
7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of #DIV/0! to 9999.99, and all other empty cells and special types to **GAUSS** missing values:

```
    vls = reshape("",9,1);
    vls[4] = "Division by Zero";
```

If **`xlsReadM`** fails, it will either terminate and print an error message or return a scalar error code, which can be decoded with **`scalerr`**, depending on the state of the `trap` flag.

trap 0	Print error message and terminate program.
trap 1	Return scalar error code 10.

See Also

xlsReadSA, xlsWrite, xlsWriteM, xlsWriteSA, xlsGetSheetCount, xlsGetSheetSize, xlsGetSheetTypes, xlsMakeRange

xlsReadSA

Purpose

Reads from an Excel® spreadsheet into a **GAUSS** string array or string.

Format

s = **xlsReadSA**(*file*, *range*, *sheet*, *vls*);

Input

file	string, name of `.xls` or `.xlsx` file.
range	string, range to read, e.g. `a2:b20` or the starting point of the read, e.g. `a2`.
sheet	scalar, sheet number.

vls	null string or 9x1 string array, specifies the conversion of Excel® empty cells and special types into **GAUSS** (see Remarks). A null string results in all empty cells and special types being converted to null strings.

Output

s	string array or string or a scalar error code.

Portability

Windows, **Linux** and **Mac**

The *vls* input is currently ignored on Mac and Linux. Missing values will be returned for all cells that are empty or contain errors.

Remarks

If *range* is a null string, then by default the read will begin at cell `a1`.

The *vls* argument lets users control the import of Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell
1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!

7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of #DIV/0! to "Division by Zero", and all other empty cells and special types to null strings:

```
vls = reshape("",9,1);
vls[4] = "Division by Zero";
```

If **`xlsReadSA`** fails, it will either terminate and print an error message or return a scalar error code, which can be decoded with **`scalerr`**, depending on the state of the `trap` flag.

trap 0	Print error message and terminate program.
trap 1	Return scalar error code 10.

See Also

xlsReadM, xlsWrite, xlsWriteM, xlsWriteSA, xlsGetSheetCount, xlsGetSheetSize, xlsGetSheetTypes, xlsMakeRange

xlsWrite

Purpose

Writes a **GAUSS** matrix, string, or string array to an Excel® spreadsheet.

Format

```
ret = xlsWrite(data, file, range, sheet, vls);
```

Input

data	matrix, string, or string array.
file	string, name of `.xls` or `.xlsx` file.
range	string, the starting point of the write, e.g. `a2`.
sheet	scalar, sheet number.
vls	null string or 9x1 matrix or string array, specifies the conversion of **GAUSS** values or characters into Excel® empty cells and special types (see Remarks). A null string results in all **GAUSS** missing values and null strings being converted to empty cells.

Output

ret	scalar, 0 if success or a scalar error code.

Portability

Windows, **Linux** and **Mac**

The *vls* input is currently ignored on Mac and Linux. Missing values will be returned for all cells that are empty or contain errors.

Remarks

The *vls* argument lets users control the export to Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell

1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!
7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of 9999.99 to #DIV/0! in Excel® and convert all **GAUSS** missing values to empty cells in Excel®:

```
    vls = reshape(error(0),9,1);
    vls[4] = 9999.99;
```

If **`xlsWrite`** fails, it will either terminate and print an error message or return a scalar error code, which can be decoded with **`scalerr`**, depending on the state of the `trap` flag.

trap 0	Print error message and terminate program.
trap 1	Return scalar error code 10.

See Also

xlsReadSA, xlsReadM, xlsWriteM, xlsWriteSA, xlsGetSheetCount, xlsGetSheetSize, xlsGetSheetTypes, xlsMakeRange

xlsWriteM

Purpose

Writes a **GAUSS** matrix to an Excel® spreadsheet.

Format

```
ret = xlsWriteM(data, file, range, sheet, vls);
```

Input

data	matrix.
file	string, name of `.xls` or `.xlsx` file.
range	string, the starting point of the write, e.g. `a2`.
sheet	scalar, sheet number.
vls	null string or 9x1 matrix, specifies the conversion of **GAUSS** values into Excel® empty cells and special types (see Remarks). A null string results in all **GAUSS** missing values being converted to empty cells.

Output

ret	scalar, 0 if success or a scalar error code.

Portability

Windows, **Linux** and **Mac**

The `vls` input is currently ignored on Mac and Linux. Missing values will be returned for all cells that are empty or contain errors.

Remarks

The `vls` argument lets users control the export to Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell
1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!
7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of 9999.99 to #DIV/0! in Excel® and convert all **GAUSS** missing values to empty cells in Excel®:

```
    vls = reshape(error(0),9,1);
    vls[4] = 9999.99;
```

If **`xlsWriteM`** fails, it will either terminate and print an error message or return a scalar error code, which can be decoded with **`scalerr`**, depending on the state of the `trap` flag.

trap 0	Print error message and terminate program.
trap 1	Return scalar error code 10.

See Also

xlsReadSA, xlsReadM, xlsWrite, xlsWriteSA, xlsGetSheetCount, xlsGetSheetSize, xlsGetSheetTypes, xlsMakeRange

xlsWriteSA

Purpose

Writes a **GAUSS** string or string array to an Excel® spreadsheet.

Format

ret = **xlsWriteSA**(*data*, *file*, *range*, *sheet*, *vls*);

Input

data	string or string array.
file	string, name of `.xls` file.
range	string, the starting point of the write, e.g. `a2`.
sheet	scalar, sheet number.
vls	null string or 9x1 string array, specifies the conversion of **GAUSS** characters into Excel® empty cells and special types (see Remarks). A null string results in all null strings being converted to empty cells.

Output

ret	scalar, 0 if success or a scalar error code.

Portability

Windows, **Linux** and **Mac**

The *vls* input is currently ignored on Mac and Linux. Missing values will be returned for all cells that are empty or contain errors.

Remarks

The *vls* argument lets users control the export to Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell
1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!
7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of "Division by Zero" to #DIV/0!, and all null strings to empty cells:

```
    vls = reshape("",9,1);
    vls[4] = "Division by Zero";
```

If **xlsWriteSA** fails, it will either terminate and print an error message or return a scalar error code, which can be decoded with **scalerr**, depending on the state of the `trap` flag.

trap 0	Print error message and terminate program.
trap 1	Return scalar error code 10.

See Also

xlsReadM, xlsWrite, xlsWriteM, xlsReadSA, xlsGetSheetCount, xlsGetSheetSize, xlsGetSheetTypes, xlsMakeRange

xpnd

Purpose

Expands a column vector into a symmetric matrix.

Format

x = **xpnd**(*v*);

Input

v	Kx1 vector, to be expanded into a symmetric matrix.

Output

x	MxM matrix, the results of taking *v* and filling in a symmetric matrix with its elements. `M =( (-1 + sqrt(1 + 8 * K) )/2)`

Remarks

If *v* does not contain the right number of elements, (that is, if **sqrt**(1 + 8 * K) is not integral), then an error message is generated.

This function is particularly useful for hard-coding symmetric matrices, because only about half of the matrix needs to be entered.

Example

```
x = { 1,
      2, 3,
      4, 5, 6,
      7, 8, 9, 10 };
y = xpnd(x);
```

After the code above, the variables *x* and *y* are equal to:

```
    1
    2
    3
    4        1    2    4    7
x = 5    y = 2    3    5    8
    6        4    5    6    9
    7        7    8    9   10
    8
```

```
        9
       10
```

See Also

vech

xtics

Purpose

Sets and fixes scaling, axes numbering and tick marks for the X axis. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

xtics(*min*, *max*, *step*, *minordiv*);

Input

min	scalar, the minimum value.
max	scalar, the maximum value.
step	scalar, the value between major tick marks.
minordiv	scalar, the number of minor subdivisions.

Remarks

This routine fixes the scaling for all subsequent graphs until **graphset** is called.

This gives you direct control over the axes endpoints and tick marks. If **xtics** is called after a call to **scale**, it will override **scale**.

X and Y axes numbering may be reversed for **xy**, **logx**, **logy**, and **loglog** graphs. This may be accomplished by using a negative step value in the **xtics** and **ytics** functions.

Source

pscale.src

See Also

scale, ytics, ztics

xy

Purpose

Graphs X vs. Y using Cartesian coordinates. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

xy(*x*, *y*);

Input

x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

Remarks

Missing values are ignored when plotting symbols. If missing values are encountered while plotting a curve, the curve will end and a new curve will begin plotting at the next non-missing value.

Source

`pxy.src`

See Also

xyz, logx, logy, loglog

xyz

Purpose

Graphs X vs. Y vs. Z using Cartesian coordinates. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

```
xyz(x, y, z);
```

Input

x	Nx1 or NxK matrix. Each column contains the X values for a particular line.
y	Nx1 or NxK matrix. Each column contains the Y values for a particular line.
z	Nx1 or NxK matrix. Each column contains the Z values for a particular line.

Remarks

Missing values are ignored when plotting symbols. If missing values are encountered while plotting a curve, the curve will end and a new curve will begin plotting at the next non-missing value.

Source

`pxyz.src`

y

ylabel

Purpose

Sets a label for the Y axis. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

```
ylabel(str);
```

Input

str	string, the label for the Y axis.

Source

`pgraph.src`

See Also

title, xlabel, zlabel

ytics

Purpose

Sets and fixes scaling, axes numbering and tick marks for the Y axis. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

```
ytics(min, max, step, minordiv);
```

Input

min	scalar, the minimum value.
max	scalar, the maximum value.
step	scalar, the value between major tick marks.
minordiv	scalar, the number of minor subdivisions.

Remarks

This routine fixes the scaling for all subsequent graphs until **graphset** is called.

This gives you direct control over the axes endpoints and tick marks. If **ytics** is called after a call to **scale**, it will override **scale**.

X and Y axes numbering may be reversed for **xy**, **logx**, **logy** and **loglog** graphs. This may be accomplished by using a negative step value in the **xtics** and **ytics** functions.

Source

```
pscale.src
```

See Also

scale, xtics, ztics

Z

zeros

Purpose

Creates a matrix of zeros.

Format

```
y = zeros(r, c);
```

Input

r	scalar, the number of rows.
c	scalar, the number of columns.

Output

y	*r* x *c* matrix of zeros.

Remarks

This is faster than **ones**.

Noninteger arguments will be truncated to an integer.

Example

```
y = zeros(3,2);
print y;
```

The code above produces the following output:

```
0.000     0.000
0.000     0.000
0.000     0.000
```

See Also

ones, eye

zeta

Purpose

Computes the Rieman Zeta function.

Format

f = **zeta**(*z*);

Input

z	NxK matrix; *z* may be complex.

Output

f	NxK matrix.

Remarks

Euler MacLaurin series.

References

1. Jon Breslaw, 2009

zlabel

Purpose

Sets a label for the Z axis. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

`zlabel`(*`str`*);

Input

str	string, the label for the Z axis.

Source

pgraph.src

See Also

title, xlabel, ylabel

ztics

Purpose

Sets and fixes scaling, axes numbering and tick marks for the Z axis. NOTE: This function is for the deprecated PQG graphics.

Library

pgraph

Format

```
ztics(min, max, step, minordiv);
```

Input

min	scalar, the minimum value.
max	scalar, the maximum value.
step	scalar, the value between major tick marks.
minordiv	scalar, the number of minor subdivisions. If this function is used with **contour**, contour labels will be placed every

`minordiv` levels. If 0, there will be no labels.

Remarks

This routine fixes the scaling for all subsequent graphs until **`graphset`** is called.

This gives you direct control over the axes endpoints and tick marks. If **`ztics`** is called after a call to **`scale3d`**, it will override **`scale3d`**.

Source

```
pscale.src
```

See Also

scale3d, xtics, ytics, contour

39 Obsolete Commands

The following commands will no longer be supported and therefore should not be used when creating new programs.

`color`	`eigcg2`
`coreleft`	`eigch`
`csrtype`	`eigch2`
`denseSubmat`	`eigrg`
`dfree`	`eigrg2`
`disable`	`eigrs`
`editm`	`eigrs2`
`eigcg`	`enable`

`export`	`ndpclex`
`exportf`	`ndpcntrl`
`files`	`plot`
`font`	`plotsym`
`FontLoad`	`prcsn`
`FontUnload`	`print on/off`
`FontUnloadAll`	`rndns`
`graph`	`rndus`
`import`	`scroll`
`importf`	`setvmode`
`isSparse`	`sparseCols`
`line`	`sparseEye`
`lpos`	`sparseFD`
`lprint`	`sparseFP`
`lprint on/off`	`sparseHConcat`
`lpwidth`	`sparseNZE`
`lshow`	`sparseOnes`
`medit`	`sparseRows`
`nametype`	`sparseScale`
`ndpchk`	`sparseSet`

sparseSolve

sparseSubmat

sparseTD

sparseTranspose

sparseTrTD

sparseTScalar

sparseVConcat

spline1d

spline2d

vartype

WinClear

WinClearArea

WinClearTTYlog

WinClose

WinCloseAll

WinGetActive

WinGetAttributes

WinGetColorCells

WinGetCursor

WinMove

WinOpenPQG

WinOpenText

WinOpenTTY

WinPan

WinPrint

WinPrintPQG

WinRefresh

WinRefreshArea

WinResize

WinSetActive

WinSetBackground

WinSetColorCells

WinSetColormap

WinSetCursor

WinSetForeground

WinSetRefresh

WinSetTextWrap

WinZoomPQG

Index