

GAUSSTM

Language Reference

Information in this document is subject to change without notice and does not represent a commitment on the part of Aptech Systems, Inc. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of this agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying or recording, for any purpose other than the purchaser's personal use without the written permission of Aptech Systems, Inc.

© 1984–2000 Aptech Systems, Inc. All rights reserved.

GAUSS, GAUSS Light, GAUSSi, GAUSS-386, GAUSS-386i, and GAUSS 3.5 are trademarks of Aptech Systems, Inc.
GEM is a trademark of Digital Research, Inc.
Lotus is a trademark of Lotus Development Corp.
HP Laser Jet and HP-GL are trademarks of Hewlett-Packard Corp.
PostScript is a trademark of Adobe Systems Inc.
IBM is a trademark of International Business Machines Corporation.
Hercules is a trademark of Hercules Computer Technology, Inc.
GraphicC is a trademark of Scientific Endeavors Corporation.
Tektronix is a trademark of Tektronix, Inc.
Windows is a registered trademark of Microsoft Corporation.
Other trademarks are the property of their respective owners.

Part Number: 000564
Version 3.5
Revised June 1, 2000

Contents

Contents	i
Introduction	1-1
Documentation Conventions	1-2
Using This Manual	1-2
Global Control Variables	1-3
Commands by Category	2-1
Mathematical Functions	2-1
Matrix Manipulation	2-9
Data Handling	2-12
Compiler Control	2-15
Program Control	2-15
OS Functions	2-19
Workspace Management	2-20
Error Handling and Debugging	2-20
String Handling	2-21
Time and Date Functions	2-22
Console I/O	2-22
Output Functions	2-23
Graphics	2-24
Command Reference	3-1
Command Components	3-1
Obsolete Commands	A1
Colors Appendix	B1

Introduction 1

This GAUSSTM 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 that 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. A variety of sorting routines that 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.

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 ."
<i>italic text</i>	variables	"...If <i>vnames</i> is a string or has fewer elements than <i>x</i> has columns, it will be..."
monospace	code example	<pre>if scalerr(cm); cm = inv(x); endif;</pre>
monospace bold	Refers to a GAUSS programming element within a narrative paragraph.	"...as explained under create ..."

Using This Manual

Users who are new to GAUSS should make sure they have familiarized themselves with "Language Fundamentals" in the *User's Guide* before proceeding here. That chapter contains the basics of GAUSS programming.

In all, there are over 400 routines described in this GAUSS Language Reference. We suggest that new GAUSS users skim through Chapter 2, and then browse through Chapter 3, the main part of this manual. Here, users can familiarize themselves with the kinds of tasks that GAUSS can handle easily.

Chapter 2 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 3.

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 3 under the discussion of that function.

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.

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 2 to 0, for example, edit the file `gauss.dec` and change the statement

```
declare matrix __output = 2;
```

to read

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

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

```
__output = 2;
```

similarly.

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.

Commands by Category **2**

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.
cos	Computes cosine.
cosh	Computes hyperbolic cosine.
exp	Computes the exponential function of x .
gamma	Computes gamma function value.
ln	Computes the natural log of each element.
lnfact	Computes natural log of factorial function.
log	Computes the $\log_{10}$ of each element.

pi	Returns π .
sin	Computes sine.
sinh	Computes the hyperbolic sine.
sqrt	Computes the square root of each element.
tan	Computes tangent.
tanh	Computes hyperbolic tangent

All trigonometric functions take or return values in radian units.

Differentiation and Integration

gradp	Computes first derivative of a function.
hessp	Computes second derivative of a function.
intgrat2	Integrates a 2-dimensional function over a user-defined region.
intgrat3	Integrates a 3-dimensional function over a user-defined region.
intquad1	Integrates a 1-dimensional function.
intquad2	Integrates a 2-dimensional function over a user-defined rectangular region.
intquad3	Integrates a 3-dimensional function over a 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.
cond	Computes condition number of a 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.
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	Generalized sweep inverse.
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.
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 .
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 .
qrsol	Solves a system of equations $Rx=b$ 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.
rank	Computes rank of a matrix.
rcondl	Returns reciprocal of the condition number of last decomposed matrix.
rref	Computes reduced row echelon form of a matrix.
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 .

The decomposition routines are **chol** for Cholesky decomposition, **crout** and **croutp** for Crout decomposition, **qqr-qyrep** for QR decomposition, and **svd**, **svd1**, and **svd2** 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 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 N^{th} 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.
fftn	Computes 1- or 2-D FFT using prime factor algorithm.
ffti	Computes inverse 1- or 2-D FFT.
fftm	Computes multi-dimensional FFT.
fftmf	Computes inverse multi-dimensional FFT.
rfft	Computes real 1- or 2-D FFT.
rfftn	Computes real 1- or 2-D FFT using prime factor algorithm.
rfftp	Computes real 1- or 2-D FFT, returns packed format FFT.
rfftnp	Computes real 1- or 2-D FFT using prime factor algorithm, returns packed format FFT.
rffti	Computes inverse real 1- or 2-D FFT.
rfftip	Computes inverse real 1- or 2-D FFT from packed format FFT.

Random Numbers

rndbeta	Computes random numbers with beta distribution.
rndgam	Computes random numbers with gamma distribution.
rndn	Computes random numbers with Normal distribution.
rndns	Computes random numbers with Normal distribution using specified seed.
rndnb	Computes random numbers with negative binomial distribution.

rndp	Computes random numbers with Poisson distribution.
rndu	Computes random numbers with uniform distribution.
rndus	Computes random numbers with uniform distribution using specified seed.
rndcon	Changes constant of random number generator.
rndmod	Changes modulus of random number generator.
rndmult	Changes multiplier of random number generator.
rndseed	Changes seed of random number generator.

The random number generator can be seeded. Set the seed using **rndseed**, or generate the random numbers using **rndus** or **rndns**. For example:

```
rndseed 44435667;  
x = rndu(1,1);  
or  
seed = 44435667;  
x = rndus(1,1,seed);
```

Fuzzy Conditional Functions

feq	Fuzzy ==
fge	Fuzzy $\geq$
fgt	Fuzzy >
fle	Fuzzy $\leq$
flt	Fuzzy <
fne	Fuzzy $\neq$
dotfeq	Fuzzy .==
dotfge	Fuzzy . $\geq$
dotfgt	Fuzzy .>
dotfle	Fuzzy . $\leq$
dotflt	Fuzzy .<
dotfne	Fuzzy . $\neq$

The global variable **_fcomptol** controls the tolerance used for comparison. By default, this is 1e-15. The default can be changed by editing the file **fcompare.dec**.

Statistical Functions

dstat	Computes descriptive statistics of a data set or matrix.
conv	Computes convolution of two vectors.
corrmm	Computes correlation matrix of a moment matrix.
corrvc	Computes correlation matrix from a variance-covariance matrix.
corrxx	Computes correlation matrix.
crossprd	Computes cross product.
design	Creates a design matrix of 0's and 1's.
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.
ols	Computes least squares regression of data set or matrix.
olsqr	Computes OLS coefficients using QR decomposition.
olsqr2	Computes OLS coefficients, residuals, and predicted values using QR decomposition.
stdc	Computes standard deviation of the columns of a matrix.
toeplitz	Computes Toeplitz matrix from column vector.
vcm	Computes a variance-covariance matrix from a moment matrix.
vcx	Computes a 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.)

Statistical Distributions

cdfbeta	Computes integral of beta function.
cdfbvn	Computes lower tail of bivariate Normal cdf.
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 .
cdffc	Computes complement of cdf of F .
cdffnc	Computes integral of noncentral F .

cdfgam	Computes integral of incomplete Γ function.
cdfmvn	Computes multivariate Normal cdf.
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).
cdftc	Computes complement of cdf of t -distribution.
cdftnc	Computes integral of noncentral t -distribution.
cdftvn	Computes lower tail of trivariate Normal cdf.
erf	Computes Gaussian error function.
erfc	Computes complement of Gaussian error function.
lncdfbvn	Computes natural log of bivariate Normal cdf.
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.
lnpdfn	Computes Normal log-probabilities.
lnpdfmvn	Computes multivariate Normal log-probabilities.
pdfn	Computes standard Normal probability density function.

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 $+\infty$.
floor	Rounds down towards $-\infty$.
prcsn	Sets computational precision for matrix operations.
round	Rounds to the nearest integer.
trunc	Truncates toward 0.

All calculations in GAUSS are done in double precision, with the exception of some of the intrinsic functions on OS/2 and DOS. These may use extended precision (18-19 digits of accuracy). Use **prcsn** to change the internal accuracy used in these cases.

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.

Matrix Manipulation

Creating Vectors and Matrices

editm	Simple matrix editor.
eye	Creates identity matrix.
let	Creates matrix from list of constants.
medit	Full-screen spreadsheet-like matrix editor.
ones	Creates a matrix of ones.
zeros	Creates a matrix of zeros.

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

medit is a full-screen editor that can be used to create matrices to be stored in memory, or to edit matrices that already exist.

Matrices can also be loaded from an ASCII file, from a GAUSS matrix file, or from a GAUSS data set. (See “Procedures and Keywords” in the *User’s Guide* for more information.)

Loading and Storing Matrices

loadd	Loads matrix from data set.
loadm	Loads matrix from ASCII or matrix file.
save	Saves matrix to matrix 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.
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.
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.
cumprodc	Computes cumulative products of each column of a matrix.
cumsumc	Computes cumulative sums of each column of a matrix.
prodc	Computes the product of each column of a matrix.
sumc	Computes the sum of each column 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.

Sparse Matrix Functions

sparseFD	Converts dense matrix to sparse matrix.
sparseFP	Converts packed matrix to sparse matrix.
sparseOnes	Generates sparse matrix of ones and zeros.
sparseEye	Creates sparse identity matrix.
sparseTD	Multiplies sparse matrix by dense matrix.
sparseTrTD	Multiplies sparse matrix transposed by dense matrix.
sparseSolve	Solves $Ax = B$ for x where A is a sparse matrix.
sparseHConcat	Horizontally concatenates sparse matrices.
sparseVConcat	Vertically concatenates sparse matrices.
sparseSubmat	Returns sparse submatrix of sparse matrix.
denseSubmat	Returns dense submatrix of sparse matrix.

sparseRows	Returns number of rows in sparse matrix.
sparseCols	Returns number of columns in sparse matrix.
sparseNZE	Returns the number of nonzero elements in sparse matrix.
isSparse	Tests whether a matrix is a sparse matrix.
sparseSet	Resets sparse library globals.

Miscellaneous Matrix Manipulation

rev	Reverses the order of rows of a matrix.
rotater	Rotates the rows of a matrix, wrapping elements as necessary.
shiftr	Shifts rows of a matrix, filling in holes with a specified value.
reshape	Reshapes a matrix to new dimensions.
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.
xpnd	Expands a column vector into a symmetric matrix.
delif	Deletes rows from a matrix using a logical expression.
diag	Extracts the diagonal of a matrix.
diagrv	Puts a column vector into the diagonal of a matrix.
exctsmpl	Creates a random subsample of data set, with replacement.
lowmat	Returns the main diagonal and lower triangle.
lowmat1	Returns a main diagonal of 1's and the lower triangle.
upmat	Returns the main diagonal and upper triangle.
upmat1	Returns a main diagonal of 1's and the upper triangle.
selif	Selects rows from a matrix using a logical expression.
submat	Extracts a submatrix from a matrix.
trimr	Trims rows from top or bottom of a matrix.
union	Returns the union of two vectors.
intrsect	Returns the intersection of two vectors.
setdif	Returns elements of one vector that are not in another.
complex	Creates a complex matrix from two real matrices.
imag	Returns the imaginary part of a complex matrix.
real	Returns the real part of a complex 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 that contain missing values from a matrix in memory, see **packr**.

Data Handling

Data Sets

close	Closes an open data set (.dat file).
closeall	Closes all open data sets.
create	Creates and opens a data set.
eof	Tests for end of file.
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.
iscplx	Returns whether a data set is real or complex.
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). (See “File I/O” in the *User’s Guide* for more information.)

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 utility **atog**. (See “Utilities” in the *User’s Guide*.)

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” in the *User’s Guide*.) 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.
indcv	Returns column numbers of variables within 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.
mergevar	Concatenates column vectors to create larger matrix.
makevars	Decomposes matrix to create column vectors.
setvars	Creates globals using the names in a data set.
vartype	Returns column vector of variable types (numeric/character) 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.

getname and **vartype** are supported for backwards compatibility.

Data Coding

code	Codes the data in a vector by applying a logical set of rules to assign each data value to a category.
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 .
ismiss	Returns 1 if matrix has any missing values, 0 otherwise.
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.
packr	Deletes rows with missing values.

recode	Similar to code , but leaves the original data in place if no condition is met.
scalmiss	Tests whether a scalar is the missing value code.
substute	Similar to recode , but operates on matrices.
subscat	Simpler version of recode , but uses ascending bins instead of logical conditions.

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

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-sortsrows 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.
uniqindx	Returns a sorted unique index of a vector.
unique	Removes duplicate elements of a vector.
intrleav	Produces one large sorted data file from two smaller sorted files having the same keys.
mergeby	Produces one large sorted data file from two smaller sorted files having a single key column in common.

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.

Compiler Control

#define	Defines a case-insensitive text-replacement or flag variable.
#definecs	Defines a case-sensitive text-replacement or flag variable.
#undef	Undefines a text-replacement or flag variable.
#ifdef	Compiles code block if a variable has been #define 'd.
#ifndef	Compiles code block if a variable has not been #define 'd.
#iflight	Compiles code block if running GAUSS Light.
#ifdos	Compiles code block if running DOS.
#ifos2win	Compiles code block if running OS/2 or Windows.
#ifunix	Compiles code block if running UNIX.
#else	Alternates clause for #if-#else-#endif code block.
#endif	End of #if-#else-#endif code block.
#include	Includes code from another file in program.
#lineson	Compiles program with line number and file name records.
#linesoff	Compiles program without 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).

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 "Language Fundamentals" in the *User's Guide* for more information.)

Program Control

Execution Control

end	Terminates a program and close 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 leave 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

if..endif Conditional branching.
goto Unconditional branching.
pop Retrieve **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 Jump out the bottom of a **do** or **for** loop.
continue Jump to the top of a **do** or **for** loop.
do while..endo Loop if *TRUE*.
do until..endo Loop if *FALSE*.
for.. endfor Loop 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	Branch to subroutine.
pop	Retrieve gosub arguments.
return	Return 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 Chapter 3, “Command Reference”, for details.

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

Procedures

local	Declares variables local to a procedure.
proc	Begins definition of multi-line procedure.
retp	Returns from a procedure.
endp	Terminates a procedure definition.

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” and “Libraries” in the *User’s Guide* for details.

Libraries

call	Calls function and discard return values.
declare	Initializes variables at compile time.
external	External symbol definitions.
library	Sets up list of active libraries.
lib	Builds or updates a GAUSS library.

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.
saveall	Saves the contents of the current workspace to a file.
use	Loads previously compiled code.
save	Saves the compiled image of a procedure to disk.
loadp	Loads compiled procedure.

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 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.)

OS Functions

cdir	Returns current directory.
chdir	Changes directory interactively.
ChangeDir	Changes directory in program.
dfree	Returns free space on disk.
shell	Shells to OS.
envget	Gets an environment string.
exec	Executes an executable program file.
fileinfo	Takes a file specification, returns names and information of files that match.
files	Takes a file specification, returns names of files that match.
filesa	Takes a file specification, returns names of files that match.

Workspace Management

clear	Sets matrices equal to 0.
clearg	Sets global symbols to 0.
maxvec	Returns maximum allowed vector size.
delete	Deletes specified global symbols.
new	Clears current workspace.
show	Displays global symbol table.
iscplx	Returns whether a matrix is real or complex.
hasimag	Examines matrix for nonzero imaginary part.
type	Returns types of argument (matrix or string).
typecv	Returns types of symbol (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.

coreleft is most commonly used to determine how many rows of a data set may be read into memory at one time.

Error Handling and Debugging

debug	Executes a program under the source level debugger.
disable	Disables invalid operation interrupt of coprocessor.
error	Creates user-defined error code.
errorlog	Sends error message to screen and log file.
enable	Enables invalid operation interrupt of coprocessor.
#lineson	Includes line number and file name records in program.
#linesoff	Omits line number and file name records from program.
ndpchk	Examines status word of coprocessor.
ndpclex	Clears coprocessor exception flags.
ndpcntrl	Sets and gets coprocessor control word.
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**.

String Handling

chrs	Converts ASCII values to a string.
ftocv	Converts an NxK matrix to a character matrix.
ftos	Converts a floating point scalar to string.
getf	Loads ASCII or binary file into string.
loads	Loads a string file (.fst file).
lower	Converts a string to lowercase.
putf	Writes a string to disk file.
stof	Converts a string to floating point numbers.
strindx	Finds starting location of one string in another string.
strlen	Returns length of 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.
upper	Changes a string to uppercase.
vals	Converts a string to ASCII values.
varget	Accesses the global variable named by a string.
varput	Assigns a global variable named by a string.
vargetl	Accesses the local 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**.

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.
etdays	Difference between two times in days.
ethsec	Difference between two times in 100ths of a second.
etstr	Converts elapsed time to string.
hsec	Returns elapsed time since midnight in 100ths of a second.
time	Returns current system time.
timestr	Formats time as “hh:mm:ss”.

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

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

will time the GAUSS multiplication operator.

Console I/O

con	Requests console input, create matrix.
cons	Requests console input, create string.
key	Gets the next key from the keyboard buffer. If buffer is empty, returns a 0.
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.

Output Functions

Text Output

comlog	Controls interactive command logging.
output	Redirects print statements to auxiliary output.
outwidth	Sets line width of auxiliary output.
print	Prints to window.
print [[on off]]	Turns auto window print on and off.
printfm	Prints matrices using a different format for each column.
screen [[on off]]	Directs/suppresses print statements to window.
screen out	Dumps snapshot of window to auxiliary output.
lpos	Returns print head position in printer buffer.
lprint	Prints expression to the printer.
lprint [[on off]]	Switches auto printer mode on and off.
lpwidth	Specifies printer width.
lshow	Prints global symbol table on the printer.
plot	Plots elements of two matrices in text mode.
plotsym	Controls data symbol used by plot .
cls	Clears the window.
color	Sets pixel, text, background colors.
csrcol	Gets column position of cursor on window.
csrlin	Gets row position of cursor on window.
edit	Edits a file with the GAUSS editor.
ed	Accesses an alternate editor.

format	Defines format of matrix printing.
locate	Positions the cursor on the window.
printdos	Prints a string for special handling by the OS.
scroll	Scrolls a section of the 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.

printdos can be used to print in reverse video, or using different colors. It requires that `ansi.sys` be installed.

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

Window Graphics

graph	Sets pixels.
setvmode	Sets video mode.
color	Sets color.
line	Draws lines.

graph allows the user to plot individual pixels.

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 “Publications Quality Graphics” in the *User’s Guide*.

Graph Types

xy	Graphs X,Y using Cartesian coordinate system.
logx	Graphs X,Y using logarithmic X axis.
logy	Graphs X,Y using logarithmic Y axis.
loglog	Graphs X,Y using logarithmic X and Y axes.
bar	Generates bar graph.
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.
box	Graphs data using the box graph percentile method.

xyz	Graphs X, Y, Z using 3-D Cartesian coordinate system.
surface	Graphs a 3-D surface.
contour	Graphs contour data.
draw	Supplies additional graphic elements to graphs.

Axes Control and Scaling

_paxes	Turns axes on or off.
_pcross	Controls where axes intersect.
_pgrid	Controls major and minor grid lines.
_pxpmax	Controls precision of numbers on X axis.
_pypmax	Controls precision of numbers on Y axis.
_pzpmax	Controls precision of numbers on Z axis.
_pxsci	Controls use of scientific notation on X axis.
_pysci	Controls use of scientific notation on Y axis.
_pzsci	Controls use of scientific notation on Z axis.
_pticout	Controls direction of tick marks on axes.
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 control tick marks.
ytics	Scales Y axis and control tick marks.
ztics	Scales Z axis and control tick marks.

Text, Labels, Titles, and Fonts

_plegctl	Sets location and size of plot legend.
_plegstr	Specifies legend text entries.
_pmsgctl	Controls message position.
_pmsgstr	Specifies message text.
_paxht	Controls size of axes labels.
_pnumht	Controls size of axes numeric labels.
_pnum	Axes numeric label control and orientation.
_pdate	Dates string contents and control.
_ptitlht	Controls main title size.
xlabel	X axis label.
ylabel	Y axis label.

zlabel	Z axis label.
title	Specifies main title for graph.
asclabel	Defines character labels for tick marks.
fonts	Loads fonts for labels, titles, messages and legend.

Main Curve Lines and Symbols

_pboxctl	Controls box plotter.
_pboxlim	Outputs percentile matrix from box plotter.
_plctrl	Controls main curve and frequency of data symbols.
_pcolor	Controls line color for main curves.
_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.
_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.
_perrbar	Plots error bars.

Graphic Panel, Page, and Plot Control

_pagesiz	Controls size of graph for printer output.
_pageshf	Shifts the graph for printer output.
_plotsiz	Controls plot area size.
_plotshf	Controls plot area position.
_protate	Rotates the graph 90 degrees.
margin	Controls graph margins.
axmargin	Controls axes margins and plot size.
begwind	Graphic panel initialization procedure.

endwind	End graphic panel manipulation, display graphs.
window	Creates tiled graphic panels of equal size.
makewind	Creates graphic panel with specified size and position.
setwind	Sets to specified graphic panel number.
nextwind	Sets to next available graphic panel number.
getwind	Gets current graphic panel number.
savewind	Saves graphic panel configuration to a file.
loadwind	Loads a graphic panel configuration from a file.

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.
_pzoom	Specifies zoom parameters.
_ptek	Controls creation and name of <code>graphics.tkf</code> file.
graphprt	Generates print, conversion file.

Miscellaneous

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

Command Reference 3

Command Components

The following list describes each of the components used in the GAUSS Language Command Reference.

Purpose Describes the function of the command.

Library Describes where the Run-Time Library functions can be found in the GAUSS program structure.

Format Illustrates use of the command syntax.

Input Describes the input parameters of the command.

Global Input Describes the global variables that are referenced by the command.

Output Describes the parameters that will be returned as a result of the input.

Command Reference

a	Global Output	Describes the global variables that are updated by the command.
b	Portability	Describes how GAUSS behaves under various operating systems.
c	Remarks	Explanatory material pertinent to the command.
d	Example	Sample code strings using the command.
e	Source	The source file in which Run-Time Library function is defined.
f	Globals	Global variables that are accessed by the command.
g	See also	Other commands that have similar characteristics.
h	Technical Notes	Technical discussion and reference source citations.
i	References	Reference material citations.
j		
k		
l		
m		
n		
o		
p		
q		
r		
s		
t		
u		
v		
w		
x y z		

abs

Purpose Returns the absolute value or complex modulus of x .

Format $y = \text{abs}(x);$

Input x NxK matrix.

Output y NxK matrix containing absolute values of x .

Example $x = \text{rndn}(2, 2);$
 $y = \text{abs}(x);$

$$x = \begin{bmatrix} 0.675243 & 1.053485 \\ -0.190746 & -1.229539 \end{bmatrix}$$

$$y = \begin{bmatrix} 0.675243 & 1.053485 \\ 0.190746 & 1.229539 \end{bmatrix}$$

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

arccos

arccos

Purpose	Computes the inverse cosine.
Format	$y = \text{arccos}(x);$
Input	x NxK matrix.
Output	y NxK matrix 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	$x = \{ -1, -0.5, 0, 0.5, 1 \};$ $y = \text{arccos}(x);$ -1.000000 -0.500000 $x =$ 0.000000 0.500000 1.000000 3.141593 2.094395 $y =$ 1.570796 1.047198 0.000000
Source	trig.src

arcsin

Purpose Computes the inverse sine.

Format $y = \text{arcsin}(x);$

Input x NxK matrix.

Output y NxK matrix, 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 $x = \{ -1, -0.5, 0, 0.5, 1 \};$
 $y = \text{arcsin}(x);$

```

          -1.000000
          -0.500000
 $x =$       0.000000
           0.500000
           1.000000

          -1.570796
          -0.523599
 $y =$       0.000000
           0.523599
           1.570796

```

Source trig.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

asclabel

asclabel

Purpose Sets up character labels for the X and Y axes.

Library pgraph

Format `asclabel(xl,yl);`

Input

<i>xl</i>	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.
<i>yl</i>	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 DEC";
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. (See Chapter 13 in *Using GAUSS for Windows95*.)

```
fonts("simplex simgrma");
lab = "\2010.25\2021 \2010.5\2021
\2010.75\2021 l";
asclabel(lab,0);
```

Here, the **2021** produces the “ λ ” symbol from Simgrma.

Source pgraph.src

asclabel

See also `xtics`, `ytics`, `scale`, `scale3d`, `fonts`

[a](#)[b](#)[c](#)[d](#)[e](#)[f](#)[g](#)[h](#)[i](#)[j](#)[k](#)[l](#)[m](#)[n](#)[o](#)[p](#)[q](#)[r](#)[s](#)[t](#)[u](#)[v](#)[w](#)[x y z](#)

atan

atan

Purpose Returns the arctangent of its argument.

Format $y = \text{atan}(x);$

Input x NxK matrix.

Output y NxK matrix containing the arctangent of x in radians.

Remarks y will be a matrix 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 $\frac{-\pi}{2}$ to $\frac{+\pi}{2}$. To convert radians to degrees, multiply by $\frac{180}{\pi}$.

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

Example $x = \{ 2, 4, 6, 8 \};$

$z = x/2;$

$y = \text{atan}(z);$

$y =$
 0.785398
 1.107149
 1.249046
 1.325818

See also atan2, sin, cos, pi, tan

atan2

Purpose Computes an angle from an x,y coordinate.

Format $z = \text{atan2}(y,x);$

Input

y	$N \times K$ matrix, the Y coordinate.
x	$L \times M$ matrix, $E \times E$ conformable with y , the X coordinate.

Output z $\max(N,L)$ by $\max(K,M)$ matrix.

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 $\frac{180}{\pi}$.

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

Example

```
x = 2;
y = { 2, 4, 6, 8 };
z = atan2(y,x);
```

```

0.785398
z = 1.107149
    1.249046
    1.325818
```

See also `atan`, `sin`, `cos`, `pi`, `tan`, `arcsin`, `arccos`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

axmargin

axmargin

Purpose Set absolute margins for the plot axes which control placement and size of plot.

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 9 x 6.855 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`

balance

Purpose Balances a square matrix.

Format `{ b,z } = balance(x)`

Input x KxK matrix.

Output b KxK matrix, balanced matrix.
 z KxK matrix, 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);
```

```
      100.0  100.0  37.5
b =      80.0   50.0  15.0
      56.0   32.0   9.0
```

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

band

band

Purpose	Extracts bands from a symmetric banded matrix.			
Format	$a = \text{band}(y,n);$			
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 (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	$x = \{ \begin{array}{cccc} 1 & 2 & 0 & 0, \\ 2 & 8 & 1 & 0, \\ 0 & 1 & 5 & 2, \\ 0 & 0 & 2 & 3 \end{array} \};$			
	$bx = \text{band}(x,1);$			
		0.0000000		1.0000000
	$bx =$	2.0000000		8.0000000
		1.0000000		5.0000000
	2.0000000		3.0000000	

bandchol

Purpose Computes the Cholesky decomposition of a positive definite banded matrix.

Format $l = \text{bandchol}(a);$

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 $A = L \times L'$. 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);

      0.0000000  1.0000000
bx =  2.0000000  8.0000000
      1.0000000  5.0000000
      2.0000000  3.0000000

cx = bandchol(bx);

      0.0000000  1.0000000
cx =  2.0000000  2.0000000
      0.5000000  2.1794495
      0.91766294 1.4689774
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

bandcholsol

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 $x = \text{bandcholsol}(b, l);$

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 $A = L^*L'$. 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

```
x = { 1 2 0 0,
      2 8 1 0,
      0 1 5 2,
      0 0 2 3 };
bx = band(x,1);
```

```
bx =
0.0000000 1.0000000
2.0000000 8.0000000
1.0000000 5.0000000
2.0000000 3.0000000
```


bandcholsol

```
cx = bandchol(bx);
```

```

      0.0000000    1.0000000
cx =   2.0000000    2.0000000
      0.5000000    2.1794495
      0.91766294   1.4689774
```

```
xi = bandcholsol(eye(4),cx);
```

```

      2.0731707 -0.05365854  0.14634146   0.09756098
xi =  -0.53658537  0.26829268 -0.07317073   0.04878049
      0.14634146 -0.07317073  0.29268293  -0.19512195
      -0.09756098  0.04878049 -0.19512195   0.46341463
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

bandltsol

bandltsol

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

Format $x = \text{bandltsol}(b, A);$

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

	0.0000000	1.0000000
$bx =$	2.0000000	8.0000000
	1.0000000	5.0000000
	2.0000000	3.0000000

$cx = \text{bandchol}(bx);$

	0.0000000	1.0000000
$cx =$	2.0000000	2.0000000
	0.50000000	2.1794495
	0.91766294	1.4689774

bandltsol

```
xci = bandltsol(eye(4),cx);
```

```

      1.0000000    0.0000000    0.0000000    0.0000000
xci =  -1.0000000    0.5000000    0.0000000    0.0000000
      0.22941573  -0.11470787    0.45883147    0.0000000
      -0.14331487    0.07165744  -0.28662975    0.68074565

```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

bandrv

bandrv

Purpose Creates a symmetric banded matrix, given its compact form.

Format $y = \text{bandrv}(a);$

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 (N^{th}) 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

```
0.0000000  1.0000000
bx = 2.0000000  8.0000000
      1.0000000  5.0000000
      2.0000000  3.0000000

x = bandrv(bx);

      1.0000000  2.0000000  0.0000000  0.0000000
x  = 2.0000000  8.0000000  1.0000000  0.0000000
      0.0000000  1.0000000  5.0000000  2.0000000
      0.0000000  0.0000000  2.0000000  3.0000000
```

bandso1pd

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

Format $x = \text{bandso1pd}(b,A);$

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,

$$A * x[:, i] = b[:, i]$$

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

bar

bar

Purpose	Bar graph.	
Library	pgraph	
Format	bar (<i>val</i> , <i>ht</i>) ;	
Input	<i>val</i>	Nx1 numeric vector, bar labels. If scalar 0, a sequence from 1 to rows (<i>ht</i>) will be created.
	<i>ht</i>	NxK numeric vector, bar heights. K overlapping or side-by-side sets of N bars will be graphed. For overlapping bars, the first column should contain the set of bars with the greatest height and the last column should contain the set of bars with the least height. Otherwise, the bars that are drawn first may be obscured by the bars drawn last. This is not a problem if the bars are plotted side-by-side.
Global Input	_pbarwid	global scalar, width and positioning 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. See “Colors Appendix” on page B-1.

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 to 10 (magenta) */
_pnum = 2;
bar(t,x); /* Use t vector to label X axis */
```

Source pbar.src

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

base10

base10

Purpose	Break number into a number of the form <i>#####...</i> and a power of 10.	
Format	{ <i>M,P</i> } = base10 (<i>x</i>);	
Input	<i>x</i>	scalar, number to break down.
Output	<i>M</i>	scalar, in the range -10 < <i>M</i> < 10.
	<i>P</i>	scalar, integer power such that:
	$M * 10^P = x$	
Example	{ <i>b, e</i> } = base10 (4500);	
	<i>b</i> = 4.50000000	
	<i>e</i> = 3.00000000	
Source	base10.src	

begwind

Purpose	Initialize global graphic panel variables.
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, setwind, nextwind, getwind

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

besselj

besselj

Purpose	Computes a Bessel function of the first kind, $J_n(x)$.		
Format	$y = \text{besselj}(n,x);$		
Input	n	NxK matrix, the order of the Bessel function. Nonintegers will be truncated to an integer.	
	x	LxM matrix, ExE conformable with n .	
Output	y	max(N,L) by max(K,M) matrix.	
Example	$n = \{ \ 0, \ 1 \};$		
	$x = \{ \ 0.1 \ 1.2, \ 2.3 \ 3.4 \};$		
	$y = \text{besselj}(n,x);$		
	$y =$	$\begin{matrix} 0.99750156 & 0.67113274 \\ 0.53987253 & 0.17922585 \end{matrix}$	
See also	bessely, mbesseli		

bessely

Purpose	To compute a Bessel function of the second kind (Weber's function), $Y_n(x)$.		
Format	$y = \text{bessely}(n,x);$		
Input	n	NxK matrix, the order of the Bessel function. Nonintegers will be truncated to an integer.	
	x	LxM matrix, ExE conformable with n .	
Output	y	max(N,L) by max(K,M) matrix.	
Example	$n = \{ 0, 1 \};$		
	$x = \{ 0.1 \ 1.2, \ 2.3 \ 3.4 \};$		
	$y = \text{bessely}(n,x);$		
	$y =$	$\begin{array}{cc} -1.5342387 & 0.22808351 \\ 0.05227732 & 0.40101529 \end{array}$	
See also	besselj , mbesseli		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

box**box**

Purpose Graph data using the box graph percentile method.

Library `pgraph`

Format `box(grp,y);`

Input

<i>grp</i>	1xM vector. This contains the group numbers corresponding to each column of <i>y</i> data. If scalar 0, a sequence from 1 to <code>cols(y)</code> will be generated automatically for the X axis.
<i>y</i>	NxM matrix. Each column represents the set of <i>y</i> 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:

$$\begin{aligned} \text{intqrang} &= 75^{\text{th}} - 25^{\text{th}} \\ \text{min} &= 25^{\text{th}} - 1.5\text{intqrang} \\ \text{max} &= 75^{\text{th}} + 1.5\text{intqrang} \end{aligned}$$
 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 that 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. See “Colors Appendix” on page B-1.
Remarks		If missing values are encountered in the <i>y</i> data, they will be ignored during calculations and will not be plotted.
Source		<code>pbox.src</code>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

break

break

Purpose Breaks out of a **do** or **for** loop.

Format `break;`

Example

```
x = rndn(4,4);
r = 0;
do while r < rows(x);
    r = r + 1;
    c = 0;
    do while c < cols(x);
        c = c + 1;
        if c == r;
            x[r,c] = 1;
        elseif c > r;
            break; /* terminate inner do loop */
        else;
            x[r,c] = 0;
        endif;
    endo; /* break jumps to the statement */
        /* after this endo */
    endo;
```

```
1.000 0.326 -2.682 -0.594
x = 0.000 1.000 -0.879 0.056
    0.000 0.000 1.000 -0.688
    0.000 0.000 0.000 1.000
```

Remarks This command works just like in C.

break

See also `continue`, `do`, `for`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

call

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 $y = \text{cdfbeta}(x,a,b);$

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:

$$0 \leq x \leq 1$$

$$a > 0$$

$$b > 0$$

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);
```

```

0.142285
y = 0.206629
0.260575
0.310875
```

See also cdfchic, cdfffc, cdfn, cdfnc, cdftc, gamma

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdfbeta

**Technical
Notes**

cdfbeta has the following approximate accuracy:

$\max(a,b) \leq 500$	the absolute error is approx. $\pm 5e-13$
$500 < \max(a,b) \leq 10,000$	the absolute error is approx. $\pm 5e-11$
$10,000 < \max(a,b) \leq 200,000$	the absolute error is approx. $\pm 1e-9$
$200,000 < \max(a,b)$	Normal approximations are used and the absolute error is approx. $\pm 2e-9$

References

Bol'shev, L.N. "Asymptotically Pearson's Transformations." Teor. Veroyat. Primen. (*Theory of Probability and its Applications*). Vol. 8 No. 2, 1963, 129-55.

Bosten, N. E., and E.L. Battiste. "Remark on Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 17 No. 3, March 1974, 156-57.

Ludwig, O.G. "Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 6 No. 6, June 1963, 314.

Mardia, K.V., and P.J. Zemroch. "Tables of the F- and related distributions with algorithms." Academic Press, NY, 1978. ISBN 0-12-471140-5

Peizer, D.B., and J.W. Pratt. "A Normal Approximation for Binomial, F, Beta, and Other Common, Related Tail Probabilities, I." *Journal of American Statistical Association*. Vol. 63, Dec. 1968, 1416-56.

Pike, M.C., and I.D. Hill. "Remark on Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 10 No. 6, June 1967, 375-76.

cdfbvn

Purpose	Computes the cumulative distribution function of the standardized bivariate Normal density (lower tail).		
Format	$c = \text{cdfbvn}(h,k,r);$		
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 $-\infty$ to h and $-\infty$ to k of the standardized bivariate Normal density $f(x,y,r)$.	

Remarks The function integrated is:

$$f(x, y, r) = \frac{e^{-0.5w}}{2\pi\sqrt{1-r^2}}$$

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:

$$-\infty < h < +\infty$$

$$-\infty < k < +\infty$$

$$-1 \leq r \leq 1$$

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:

$$h = (ht - ux) ./ sx;$$

$$k = (kt - uy) ./ sy;$$

cdfbvn

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 $\pm 5.0\text{e-}9$ for the entire range of arguments.

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

cdfbvn2

Purpose Returns cdfbvn of a bounded rectangle.

Format $y = \text{cdfbvn2}(h, dh, k, dk, r);$

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:

$$\text{cdfbvn}(h+dh, k+dk, r) + \text{cdfbvn}(h, k, r) - \text{cdfbvn}(h, k+dk, r) - \text{cdfbvn}(h+dh, k, r).$$

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 $0.01 * \text{abs}(y)$.

For an estimate of the actual error, see **cdfbvn2e**.

Example Example 1

```
cdfbvn2(1, -1, 1, -1, 0.5);
```

produces:

```
1.4105101488974692e-001
```

Example 2

```
cdfbvn2(1, -1e-15, 1, -1e-15, 0.5);
```

produces:

```
7.3955709864469857e-032
```

cdfbvn2

Example 3

```
cdfbvn2(1, -1e-45, 1, -1e-45, 0.5);
```

produces:

```
0.0000000000000000e+000
```

Example 4

```
trap 2,2;
```

```
cdfbvn2(1, -1e-45, 1, 1e-45, 0.5);
```

produces:

```
WARNING: Dubious accuracy from cdfbvn2:
```

```
0.000e+000 +/- 2.8e-060
```

```
0.0000000000000000e+000
```

Source `lncdfn.src`

See also `cdfbvn2e`, `lncdfbvn2`

cdfbvn2e

Purpose	Returns cdfbvn of a bounded rectangle.
Format	$\{ y, e \} = \text{cdfbvn2e}(h, dh, k, dk, r);$
Input	<i>h</i> Nx1 vector, starting points of integration for variable 1. <i>dh</i> Nx1 vector, increments for variable 1. <i>k</i> Nx1 vector, starting points of integration for variable 2. <i>dk</i> Nx1 vector, increments for variable 2. <i>r</i> Nx1 vector, correlation coefficients between the two variables.
Output	<i>y</i> Nx1 vector, the integral over the rectangle bounded by <i>h</i>, <i>h+dh</i>, <i>k</i>, and <i>k+dk</i> of the standardized bivariate Normal distribution. <i>e</i> Nx1 vector, an error estimate.
Remarks	Scalar input arguments are okay; they will be expanded to Nx1 vectors. cdfbvn2e computes $\text{cdfbvn}(h+dh, k+dk, r) + \text{cdfbvn}(h, k, r) - \text{cdfbvn}(h, k+dk, r) - \text{cdfbvn}(h+dh, k, r)$. The real answer is $y \pm e$. The size of the error depends on the input arguments.
Example	Example 1 $\text{cdfbvn2e}(1, -1, 1, -1, 0.5);$ produces: 1.4105101488974692e-001 1.9927918166193113e-014 Example 2 $\text{cdfbvn2e}(1, -1\text{e-}15, 1, -1\text{e-}15, 0.5);$ produces:

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdfbvn2e

7.3955709864469857e-032

2.8306169312687801e-030

Example 3

`cdfbvn2e(1, -1e-45, 1, -1e-45, 0.5);`

produces:

0.000000000000000000e+000

2.8306169312687770e-060

See also `cdfbvn2`, `lncdfbvn2`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

cdfchic

Purpose	Computes the complement of the cdf of the chi-square distribution.
Format	$y = \text{cdfchic}(x, n)$
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: $x \geq 0$ $n > 0$ A -1 is returned for those elements with invalid inputs. This equals $1-F(x,n)$, where F is the chi-square cdf with n degrees of freedom. Thus, to get the chi-square 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	<pre> x = { .1, .2, .3, .4 }; n = 3; y = cdfchic(x,n); 0.991837 y = 0.977589 0.960028 0.940242 </pre>
See also	cdfbeta, cdfffc, cdfn, cdfnc, cdftc, gamma

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdfchic

**Technical
Notes**

For $n \leq 1000$, the incomplete gamma function is used and the absolute error is approx. $\pm 6e-13$. For $n > 1000$, a Normal approximation is used and the absolute error is $\pm 2e-8$.

For higher accuracy when $n > 1000$, use:

`1-cdfgam(0.5*x, 0.5*n);`

References

Bhattacharjee, G.P. "Algorithm AS 32, The Incomplete Gamma Integral." *Applied Statistics*. Vol. 19, 1970, 285-87.

Mardia, K.V., and P.J. Zemroch. "Tables of the F- and related distributions with algorithms." Academic Press, NY, 1978. ISBN 0-12-471140-5

Peizer, D.B., and J.W. Pratt. "A Normal Approximation for Binomial, F, Beta, and Other Common, Related Tail Probabilities, I." *Journal of American Statistical Association*. Vol. 63, Dec. 1968, 1416-56.

cdfchii

Purpose	Compute chi-square abscissae values given probability and degrees of freedom.	
Format	$c = \text{cdfchii}(p,n);$	
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-square distribution.
Example	The following generates a 3x3 matrix of pseudo-random numbers with a chi-squared distribution with expected value of 4: <pre> rndseed 464578; x = cdfchii(rndu(3,3),4+zeros(3,3)); 2.1096456 1.9354989 1.7549182 x = 4.4971008 9.2643386 4.3639694 4.5737473 1.3706243 2.5653688 </pre>	
Source	cdfchii.src	
See also	gammaii	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdfchinc

cdfchinc

Purpose The integral under noncentral chi-square distribution, from 0 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 $y = \text{cdfchinc}(x, v, d);$

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, <i>The Analysis of Variance</i> , App. IV. 1959.)

Output y Nx1 vector, integrals from 0 to x of noncentral chi-square.

Example

```
x = { .5, 1, 5, 25 };
p = cdfchinc(x, 4, 2);

0.0042086234
0.016608592
0.30954232
0.99441140
```

Source cdfnonc.src

See also cdffnc, cdftnc

Technical Notes Relation to cdfchic:

$$\text{cdfchic}(x, v) = 1 - \text{cdfchinc}(x, v, 0);$$

The formula used is taken from Abramowitz and Stegun, *Handbook of Mathematical Functions*. Formula 26.4.25. 1970, 942.

cdfffc

Purpose Computes the complement of the cdf of the F distribution.

Format $y = \text{cdfffc}(x, n1, n2);$

Input

x	$N \times K$ matrix.
$n1$	$L \times M$ matrix, $E \times E$ conformable with x .
$n2$	$P \times Q$ matrix, $E \times E$ 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.

Allowable ranges for the arguments are:

$$\begin{aligned} x &\geq 0 \\ n1 &> 0 \\ n2 &> 0 \end{aligned}$$

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

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, subtract **cdfffc**($x, n1, n2$) 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 $x = \{ .1, .2, .3, .4 \};$

$n1 = 0.5;$

$n2 = 0.3;$

$y = \text{cdfffc}(x, n1, n2);$

$y =$

0.751772
0.708152
0.680365
0.659816

cdffc

See also **cdfbeta, cdfchic, cdfn, cdfnc, cdftc, gamma**

**Technical
Notes**

For $\max(n1, n2) \leq 1000$, the absolute error is approximately $\pm 5e-13$. For $\max(n1, n2) > 1000$, Normal approximations are used and the absolute error is approximately $\pm 2e-6$.

For higher accuracy when $\max(n1, n2) > 1000$, use:

`cdfbeta(n2/(n2+n1*x), n2/2, n1/2);`

References

Bol'shev, L.N. "Asymptotically Pearson's Transformations." Teor. Veroyat. Primen. (*Theory of Probability and its Applications*). Vol. 8 No. 2, 1963, 129-55.

Bosten, N. E., and E.L. Battiste. "Remark on Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 17 No. 3, March 1974, 156-57.

Kennedy, W.J., Jr., and J.E. Gentle. *Statistical Computing*. Marcel Dekker, Inc., NY, 1980.

Ludwig, O.G. "Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 6 No. 6, June 1963, 314.

Mardia, K.V., and P.J. Zemroch. "Tables of the F- and related distributions with algorithms." Academic Press, NY, 1978. ISBN 0-12-471140-5

Peizer, D.B., and J.W. Pratt. "A Normal Approximation for Binomial, F, Beta, and Other Common, Related Tail Probabilities, I." *Journal of American Statistical Association*. Vol. 63, Dec. 1968, 1416-56.

Pike, M.C., and I.D. Hill. "Remark on Algorithm 179 Incomplete Beta Ratio." *Comm. ACM*. Vol. 10 No. 6, June 1967, 375-76.

cdfnnc

Purpose The integral under noncentral F distribution, from 0 to x .

Format $y = \text{cdfnnc}(x, v1, v2, d);$

Input

x	Nx1 vector, values of upper limits of integrals, $x > 0$.
$v1$	scalar, degrees of freedom of numerator, $v1 > 0$.
$v2$	scalar, degrees of freedom of denominator, $v2 > 0$.
d	scalar, noncentrality parameter, $d > 0$. This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. (See Scheffe, <i>The Analysis of Variance</i> , App. IV. 1959.)

Output y Nx1 vector of integrals from 0 to x of noncentral F .

Source cdfnnc.src

See also cdftnc, cdfchinc

Technical Notes Relation to **cdffc**:

$$\text{cdffc}(x, v1, v2) = 1 - \text{cdfnnc}(x, v1, v2, 0);$$

The formula used is taken from Abramowitz and Stegun, *Handbook of Mathematical Functions*. Formula 26.6.20. 1970, 947.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdfgam

cdfgam

Purpose 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

$$\int_0^{intlim} \frac{e^{-t}t^{(x-1)}}{\text{gamma}(x)}dt$$

The allowable ranges for the arguments are:

$$\begin{aligned}x &> 0 \\intlim &\geq 0\end{aligned}$$

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);`

```
x = 0.500000 1.00000 3.00000 10.00000  
  
                                0.000000  
                                0.200000  
                                0.400000  
intlim = 0.600000  
                                0.800000  
                                1.000000
```


cdfgam

	0.000000	0.000000	0.000000	0.000000
	0.472911	0.181269	0.00114848	2.35307E – 014
$g =$	0.628907	0.329680	0.00792633	2.00981E – 011
	0.726678	1.451188	0.0231153	9.66972E – 010
	0.794097	0.550671	0.0474226	1.43310E – 008
	0.842701	0.632120	0.0803014	1.11425E – 007

This computes the integrals over the range from 0 to 1, in increments of .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. $\pm 6e-13$
$500 \leq x \leq 10,000$	the absolute error is approx. $\pm 3e-11$
$10,000 < x$	a Normal approximation is used and the absolute error is approx. $\pm 3e-10$

References

Bhattacharjee, G.P. “Algorithm AS 32, The Incomplete Gamma Integral.” *Applied Statistics*. Vol. 19, 1970, 285-87.

Peizer, D.B., and J.W. Pratt. “A Normal Approximation for Binomial, F, Beta, and Other Common, Related Tail Probabilities, I.” *Journal of American Statistical Association*. Vol. 63, Dec. 1968, 1416-56.

Pike, M.C., and I.D. Hill. “Remark on Algorithm 179 Incomplete Beta Ratio.” *Comm. ACM*. Vol. 10 No. 6, June 1967, 375-76.

cdfmvn

cdfmvn

Purpose	Computes multivariate Normal cumulative distribution function.		
Format	$y = \text{cdfmvn}(x,r);$		
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, cdftvn, lncdfmvn		

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 $n = \text{cdfn}(x);$
 $nc = \text{cdfnc}(x);$

Input x NxK matrix.

Output n NxK matrix.
 nc NxK matrix.

Remarks n is the integral from $-\infty$ to x of the Normal density function, and nc is the integral from x to $+\infty$.

Note that: $\text{cdfn}(x) + \text{cdfnc}(x) = 1$. 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 $\pm 5\text{e-}15$ for **cdfn**(x) around 1, this does not invalidate the result. What it does mean is that for $\text{abs}(x) > (\text{approx.}) 8.2924$, 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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdfn, cdfnc

$\epsilon > 1$, which is about $2.23\text{e-}16$. This defines machine epsilon as the smallest number such that $1 - \text{machine epsilon} < 1$, or about $1.11\text{e-}16$.

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 =  -2.00000  -1.00000   0.00000   1.00000   2.00000
```

```
n =   0.02275   0.15866   0.50000   0.84134   0.97725
```

```
nc =   0.97725   0.84134   0.50000   0.15866   0.02275
```

See also

erf, erfc, cdfbeta, cdfchic, cdftc, cdffc, gamma

Technical Notes

For the integral from $-\infty$ to x :

$x \leq -37$,	cdfn underflows and 0.0 is returned
$-36 < x < -10$,	cdfn has a relative error of approx. $\pm 5\text{e-}12$
$-10 < x < 0$,	cdfn has a relative error of approx. $\pm 1\text{e-}13$
$0 < x$,	cdfn has a relative error of approx. $\pm 5\text{e-}15$

For **cdfnc**, i.e., the integral from x to $+\infty$, use the above accuracies but change x to $-x$.

References

Adams, A.G. "Remark on Algorithm 304 Normal Curve Integral." *Comm. ACM*. Vol. 12 No. 10, Oct. 1969, 565-66.

Hill, I.D., and S.A. Joyce. "Algorithm 304 Normal Curve Integral." *Comm. ACM*. Vol. 10 No. 6, June 1967, 374-75.

Holmgren, B. "Remark on Algorithm 304 Normal Curve Integral." *Comm. ACM*. Vol. 13 No. 10, Oct. 1970.

Mardia, K.V., and P.J. Zemroch. "Tables of the F- and related distributions with algorithms." Academic Press, NY, 1978. ISBN 0-12-471140-5

cdfn2

Purpose Computes interval of Normal cumulative distribution function.

Format `y = cdfn2(x,dx);`

Input

<i>x</i>	MxN matrix, abscissae.
<i>dx</i>	KxL matrix, EXE conformable to <i>x</i> , intervals.

Output *y* max(M,K) by max(N,L) matrix, the integral from *x* to *x+dx* of the Normal distribution, i.e., $Pr(x \leq X \leq x + dx)$.

Remarks The relative error is:

$ x \leq 1$ and $dx \leq 1$	$\pm 1e-14$
$1 < x < 37$ and $ dx < 1/ x $	$\pm 1e-13$
$\min(x, x+dx) > -37$ and $y > 1e-300$	$\pm 1e-11$ or better

A relative error of $\pm 1e-14$ 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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdfni

cdfni

Purpose	Computes the inverse of the cdf of the Normal distribution.	
Format	$x = \text{cdfni}(p);$	
Input	p	NxK real matrix, Normal probability levels, $0 \leq p \leq 1$.
Output	x	NxK real matrix, Normal deviates, such that $\text{cdfn}(x) = p$
Remarks	$\text{cdfn}(\text{cdfni}(p)) = p$ to within the errors given below:	
	$p \leq 4.6\text{e-}308$	-37.5 is returned
	$4.6\text{e-}308 < p < 5\text{e-}24$	accurate to ± 5 in 12th digit
	$5\text{e-}24 < p < 0.5$	accurate to ± 1 in 13th digit
	$0.5 < p < 1 - 2.22045\text{e-}16$	accurate to ± 5 in 15th digit
	$p \geq 1 - 2.22045\text{e-}16$	8.12589... is returned

cdftc

Purpose Computes the complement of the cdf of the Student's t distribution.

Format $y = \text{cdftc}(x,n);$

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:

$$-\infty < x < +\infty$$

$$n > 0$$

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 $x = \{ .1, .2, .3, .4 \};$

$n = 0.5;$

$y = \text{cdftc}(x,n);$

0.473165
 $y =$ 0.447100
 0.422428
 0.399555

See also **cdfbeta**, **cdfchic**, **cdfffc**, **cdfn**, **cdfnc**, **gamma**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdftc**Technical
Notes**

For results greater than $0.5e-30$, the absolute error is approximately $\pm 1e-14$ and the relative error is approximately $\pm 1e-12$. 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 approximately $\pm 1e-14$ 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 $\pm 0.5e-42$.

References

Abramowitz, M., and I. A. Stegun, eds. *Handbook of Mathematical Functions*. 7th ed. Dover, NY, 1970. ISBN 0-486-61272-4

Hill, G.W. "Algorithm 395 Student's t-Distribution." *Comm. ACM*. Vol. 13 No. 10, Oct. 1970.

Hill, G.W. "Student's t-Distribution Quantiles to 20D." *Division of Mathematical Statistics Technical Paper No. 35*. Commonwealth Scientific and Industrial Research Organization, Australia, 1972.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdftci

Purpose	Computes the inverse of the complement of the Student's t cdf.		
Format	$x = \text{cdftci}(p, n);$		
Input	p	NxK real matrix, complementary Student's t probability levels, $0 \leq p \leq 1$.	
	n	LxM real matrix, degrees of freedom, $n \geq 1$, 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 $\text{cdftc}(x, n) = p$.	
Remarks	$\text{cdftc}(\text{cdftci}(p, n)) = p$ to within the errors given below:		
	$0.5\text{e-}30 < p < 0.01$	accurate to ± 1 in 12th digit	
	$0.01 < p$	accurate to $\pm 1\text{e-}14$	
	Extreme values of arguments can give rise to underflows, but no overflows are generated.		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdfnc

cdfnc

Purpose The integral under noncentral Student's t distribution, from $-\infty$ 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 $y = \text{cdfnc}(x, v, d);$

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, <i>The Analysis of Variance</i> , App. IV. 1959.)

Output y Nx1 vector, integrals from $-\infty$ to x of noncentral t .

Source cdfnonc.src

See also cdffnc, cdfchinc

Technical Notes Relation to **cdftc**:

$$\text{cdftc}(x, v) = 1 - \text{cdfnc}(x, v, 0);$$

The formula used is based on the formula in *SUGI Supplemental Library User's Guide*. SAS Institute. 1983, 232 (which is attributed to Johnson and Kotz, 1970).

The formula used here is a modification of that formula. It has been tested against direct numerical integration, and against simulation experiments in which noncentral **t** random variates were generated and the cdf found directly.

cdf tvn

Purpose	Computes the cumulative distribution function of the standardized trivariate Normal density (lower tail).		
Format	$c = \text{cdf tvn}(x1,x2,x3,\text{rho12},\text{rho23},\text{rho31});$		
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 $x3$.	
	rho31	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 $-\infty$ to $x1$, $-\infty$ to $x2$, and $-\infty$ to $x3$ of the standardized trivariate Normal density: $f(x1,x2,x3,\text{rho12},\text{rho23},\text{rho31})$	
Remarks	Allowable ranges for the arguments are: $-\infty < x1 < +\infty$ $-\infty < x2 < +\infty$ $-\infty < x3 < +\infty$ $-1 < \text{rho12} < 1$ $-1 < \text{rho23} < 1$ $-1 < \text{rho31} < 1$ In addition, rho12 , rho23 , and rho31 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 ($\text{rho12},\text{rho23},\text{rho31}$) must also be the same length, and this		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cdftvn

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);
```

See also `cdfn`, `cdfbvn`

Technical Notes

The absolute error for **cdftvn** is approximately $\pm 2.5e-8$ for the entire range of arguments.

References

Daley, D.J. "Computation of Bi- and Tri-variate Normal Integral." *Appl. Statist.* Vol. 23 No. 3, 1974, 435-38.

Steck, G.P. "A Table for Computing Trivariate Normal Probabilities." *Ann. Math. Statist.* Vol. 29, 780-800.

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

```
x = cdir(0);  
y = cdir("d:");  
print x;  
print y;
```

```
C:\GAUSS  
D:\
```

See also `files`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ceil

ceil

Purpose Round up toward $+\infty$.

Format $y = \text{ceil}(x);$

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 $+\infty$.

Example $x = 100 * \text{rndn}(2, 2);$
 $y = \text{ceil}(x);$

$x = \begin{matrix} 77.68 & -14.10 \\ 4.73 & -158.88 \end{matrix}$

$y = \begin{matrix} 78.00 & -14.00 \\ 5.00 & -158.00 \end{matrix}$

See also floor, trunc

ChangeDir

Purpose	Changes the working directory.
Format	<i>d</i> = ChangeDir (<i>s</i>);
Input	<i>s</i> string, directory to change to.
Output	<i>d</i> string, new working directory, or null string if change failed.
See also	chdir

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

chdir

chdir

Purpose	Changes working directory.
Format	chdir <i>dirstr</i> ;
Input	<i>dirstr</i> 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 the UNIX version.

chol

Purpose Computes the Cholesky decomposition of a symmetric, positive definite matrix.

Format $y = \text{chol}(x);$

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

```
x = moment(rndn(100,4),0);
y = chol(x);
ypy = y'y;
```

```

          90.746566  -6.467195  -1.927489  -15.696056
x =      -6.467195  87.806557   6.319043  -2.435953
        -1.927489   6.319043  101.973276   4.355520
        -15.696056  -2.435953   4.355520  99.042850
```

```

          9.526099 -0.678892 -0.202338 -1.647690
y =      0.000000  9.345890  0.661433 -0.380334
          0.000000  0.000000  10.074465  0.424211
          0.000000  0.000000  0.000000  9.798130
```

chol

$$ypy = \begin{pmatrix} 90.746566 & -6.467195 & -1.927489 & -15.696056 \\ -6.467195 & 87.806557 & 6.319043 & -2.435953 \\ -1.927489 & 6.319043 & 101.973276 & 4.355520 \\ -15.696056 & -2.435953 & 4.355520 & 99.042850 \end{pmatrix}$$

See also **crout, solpd**

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

cho1dn

Purpose Performs a Cholesky downdate of one or more rows on an upper triangular matrix.

Format `r = cho1dn(C,x);`

Input *C* KxK upper triangular matrix.
x NxK matrix, the rows to downdate *C* with.

Output *r* KxK upper triangular matrix, the downdated matrix.

Remarks *C* should be a Cholesky factorization.
cho1dn(C,x) is equivalent to **chol(C'C - x'x)**, but **cho1dn** is numerically much more stable.
 Warning: it is possible to render a Cholesky factorization non-positive definite with **cho1dn**. 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 = cho1dn(C,x);

      18.87055964 15.32294435 8.04947012
r =    0.00000000 9.30682813 -2.12009339
      0.00000000 0.00000000 7.62878355
```

See also cholup

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cholsol**cholsol**

Purpose Solves a system of linear equations given the Cholesky factorization of the system.

Format $x = \text{cholsol}(b, C);$

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., $A*x[:,i] = b[:,i]$.

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

```
let b[3,1] = 0.03177513 0.41823100 1.70129375;
let C[3,3] = 1.73351215 1.53201723 1.78102499
              0          1.09926365 0.63230050
              0          0          0.67015361;
x = cholsol(b,C);

      -1.94396905
x =   -1.52686768
      3.21579513

      3.00506436 2.65577048 3.08742844
A0 =   2.65577048 3.55545737 3.42362593
      3.08742844 3.42362593 4.02095978
```

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 15.32294435 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);

      20.16210005 16.50544413 9.86676135
r =    0.00000000 11.16601462 2.97761666
      0.00000000 0.00000000 11.65496052
```

See also `chol` `ldn`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

chrs

chrs

Purpose Converts a matrix of ASCII values into a string containing the appropriate characters.

Format $y = \text{chrs}(x);$

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 $n = 5;$
 $\text{print chrs}(\text{ones}(n,1)*42);$

Since the ASCII value of the asterisk character is 42, the program above will print a string of n asterisks.

$y = \text{chrs}(67\sim 65\sim 84);$

$\text{print } y;$

CAT

See also $\text{vals}, \text{ftos}, \text{stof}$

clear

Purpose Clears space in memory by setting matrices equal to scalar zero.

Format `clear x,y;`

Remarks `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 `clear x;`

See also `clearg, new, show, delete`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

clearg

clearg

Purpose This command 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 `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 procedures to clear global matrices. It will ignore any locals by the same name.

Example `x = 45;`
`clearg x;`

x = 0.0000000

See also `clear`, `delete`, `new`, `show`, `local`

close

Purpose Close 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 *fl* is a file handle and it contains the value 7, then after:

```
call close(fl);
```

the file will be closed but *fl* will still have the value 7. The best procedure is to do the following:

```
fl = close(fl);
```

This will set *fl* 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 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 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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

close

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 Close 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” 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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

closeall

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cls

Purpose Clear the window.

Format `cls;`

Portability UNIX

cls clears the active graphic panel. For Text graphic panels, this means the graphic panel buffer is cleared to the background color. For TTY graphic panels, the current output line is panned to the top of the graphic panel, effectively clearing the display. The output log is still intact. To clear the output log of a TTY graphic panel, use **WinClearTTYLog**. For PQG graphic panels, the graphic panel is cleared to the background color, and the related graphics file is truncated to zero length.

Windows

cls clears the Command window if you're in Cmnd I/O mode, the Output window if you're in Split I/O mode.

OS/2

cls clears the Main window.

Remarks This command will cause the window to clear and will locate the cursor at the upper left hand corner of the window.

See also `locate`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

code

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	$y = \text{code}(e,v);$		
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	<pre>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” */</pre>		

code

```

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);

           0
           5
x1[5, 1] = 10      (column vector of original values)
           15
           20

v[3, 1] = 1 2 3      (Note: v is a column vector)

           0 0
           1 0
e[5, 2] = 0 1
           0 1
           0 1

           3
           1
y[5, 1] = 2
           2
           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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

code

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`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

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

<i>var</i>	literal, the new variable name.
<i>defval</i>	scalar, the default value if none of the expressions are <i>TRUE</i> .
<i>val</i>	scalar, value to be used if corresponding expression is <i>TRUE</i> .
<i>expression</i>	logical scalar-returning expression that returns nonzero <i>TRUE</i> or zero <i>FALSE</i> .

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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

code (dataloop)

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

color

Purpose Set pixel, text, background color, or VGA palette color registers.

Format `y = color(cv)`

Input *cv* scalar, 2x1 or 3x1 vector of color values or Nx4 matrix of palette color values. See Portability, below, for platform specifics.

If the input vector is smaller than 3x1 or the corresponding element in the input vector is -1, the corresponding color will be left unchanged.

If the input is an Nx4 matrix, it will initialize the VGA palette (DOS) or active graphic panel's colormap (UNIX) with user-defined RGB colors interpreted as follows:

[N,1] palette register index 0-255

[N,2] red value 0-63

[N,3] green value 0-63

[N,4] blue value 0-63

Output *y* vector, or Nx4 matrix the same size as the input which contains the original color values or palette values.

Portability DOS

[1] pixel color

[2] text color

[3] ignored

UNIX

color affects the active graphic panel. X supports foreground and background colors. The **color** command makes no distinction between text and pixel colors; both affect the foreground color of the active graphic panel. If both a pixel color and text color are specified, the pixel color will be ignored, and the text color will be used to set the foreground color. Thus:

[1] foreground

or

[1] ignored

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

color

[2] foreground

or

[1] ignored

[2] foreground

[3] background

OS/2, Windows

This function is not supported under OS/2 or Windows.

Remarks This changes the window colors for your program’s output. The editor and interactive mode will not be affected.

See “Colors Appendix” on page B-1 for a color value table.

Under DOS, the VGA color palette registers may be set only if the display adapter has been already been initialized to VGA graphics mode 19 (320x200, 256 colors) with the **setvmode** command. The registers will retain the new values until the adapter is reset to text mode, which resets the palette to the default VGA colors.

This function is useful for obtaining 64 shades of a single color and/or mixing colors to user-specification.

See also **graph, line, setvmode**

cols, colsf

Purpose	cols returns the number of columns in a matrix. colsf returns the number of columns in a GAUSS data (.dat) file or GAUSS matrix (.fmt) file.	
Format	$y = \text{cols}(x);$ $yf = \text{colsf}(fh);$	
Input	x any valid expression that returns a matrix. fh file handle of an open file.	
Output	y number of columns in x . yf number of columns in the file that has the handle fh .	
Remarks	If x is an empty matrix, rows (x) and cols (x) return 0. For colsf , the file must be open.	
Example	<pre> x = rndn(100,3); y = cols(x); y = 3.0000000 create fp = myfile with x,10,4; b = colsf(fp); b = 10.0000000 </pre>	
See also	rows, rowsf, show, lshow	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

comlog

comlog

Purpose Controls logging of interactive mode commands to a disk file.

Format `comlog [[file=filename]] [[on|off|reset]];`

Input *filename* literal or ^string.

The **file=*filename*** subcommand selects the file to log interactive mode statements to. This can be any legal file name.

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.

There is no default file name.

**on,
off,
reset** literal, mode command:

on turns on command logging to the current log file. If the file already exists, subsequent commands will be appended.

off closes the log file and turns off command logging.

reset similar to the **on** subcommand, except that it resets the log file by deleting any previous commands.

Remarks 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.

In interactive mode under DOS, **F10** will load the current log file into the editor if logging is **on**. If logging is **off**, the default log file listed in the **log_file** configuration variable will be loaded into the editor.

compile

Purpose Compiles a source file to a compiled code file. See also “Compiler” in the *User’s Guide*.

Format `compile source fname;`

Input

<i>source</i>	literal or ^string, the name of the file to be compiled.
<i>fname</i>	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 <i>source</i> . 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 DOS 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 matter saved in the compiled image, put a **new** at the top of the *source file* or execute a **new** from interactive level 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.

Do not try to include compiled files with **#include**.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

compile

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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

complex

Purpose	Converts a pair of real matrices to a complex matrix.	
Format	$z = \text{complex}(x_r, x_i);$	
Input	x_r	NxK real matrix, the real elements of z .
	x_i	NxK real matrix or scalar, the imaginary elements of z .
Output	z	NxK complex matrix.
Example	$x = \begin{Bmatrix} 4 & 6 \\ 9 & 8 \end{Bmatrix};$	
	$y = \begin{Bmatrix} 3 & 5 \\ 1 & 7 \end{Bmatrix};$	
	$t = \text{complex}(x, y);$	
	$t = \begin{matrix} 4.0000000 + 3.0000000i & 6.0000000 + 5.0000000i \\ 9.0000000 + 1.0000000i & 8.0000000 + 7.0000000i \end{matrix}$	
See also	imag, real	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

con

con

Purpose Requests input from the keyboard, and returns it in a matrix.

Format $x = \text{con}(r,c);$

Input
 r scalar, row dimension of matrix.
 c scalar, column dimension of matrix.

Output x RxC matrix.

Portability UNIX, OS/2, Windows
con gets input from the active graphic panel. If you are working in terminal mode GAUSS will not “see” any input until you press ENTER, so follow each entry with an ENTER. Arrow keys are not supported in terminal mode.

Remarks r and c may be any scalar-valued expressions. Nonintegers will be truncated to an integer.

A **con** function in a program will cause a question mark to appear in the window. You can then type in numbers, and they will be placed into a matrix. GAUSS will wait until $r*c$ numbers have been typed in (e.g., if $r = 3$ and $c = 2$, then GAUSS will wait for 6 numbers). GAUSS will then arrange them in a matrix from the dimensions specified. The numbers can be separated by spaces, commas, or carriage returns.

When $r*c$ numbers have been entered, the program will resume with the next command.

A matrix editor is invoked when the **con** function is called. This editor allows you to move around the matrix you are entering and make changes. (See **editm** for additional information.)

The matrix editor allows you to use the arrow keys to move along rows or up and down columns in the matrix. When you come to the end of a row or column, movement is left and up, or right and down. If, for instance, you are moving left to right along a row using the right arrow key, you will move down to the beginning of the next row when you come to the end of the row you are in. GAUSS will beep at you if you try to move outside the matrix.

The matrix editor will show you what is in each element, and will prompt you for a new number. If you do not want to change anything, you can

move to a different element by pressing the appropriate arrow key. If you start to type a number and then press an arrow key, the number will not be saved. The BACKSPACE key will delete characters to the left of the cursor.

If you type in a space, comma, or carriage return before you type a number, nothing will happen. GAUSS will wait until you type a number or use an arrow key to move to a new element.

Numbers can be entered in scientific notation. The syntax is: $dE+n$ or $dE-n$, where d is a number and n is an integer (denoting the power of 10). Thus, $1E+10$, $1.1e-4$, $1100E+1$ are all legal.

Complex numbers can be entered by joining the real and imaginary parts with a sign (+ or -); there can be no spaces between the numbers and the sign. Numbers with no real part can be entered by appending an “i” to the number. For example, $1.2+23$, $8.56i$, $3-2.1i$, $-4.2e+6i$, and $1.2e-4-4.5e+3i$ are all legal.

If using DOS, π and e can be entered with ALT+P and ALT+E.

A **con** function can appear anywhere in an expression that any other function can appear.

Example

```
n = con(1,1);
print rndn(n,n);

? 2

-0.148030  0.861562
1.791516 -0.663392
```

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, editm, let, load, medit

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cond

cond

Purpose This procedure will compute the condition number of a matrix using the singular value decomposition.

Format $c = \text{cond}(x);$

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 $x = \begin{Bmatrix} 4 & 2 & 6, \\ 8 & 5 & 7, \\ 3 & 8 & 9 \end{Bmatrix};$

$y = \text{cond}(x);$

$y = 9.8436943$

Source svd.src

conj

Purpose Returns the complex conjugate of a matrix.

Format $y = \text{conj}(x);$

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);
```

	$1.0000000 + 9.0000000i$	2.0000000
$x =$	$4.0000000 + 4.0000000i$	$0.0000000 + 5.0000000i$
	$0.0000000 + 7.0000000i$	$8.0000000 - 2.0000000i$
	$1.0000000 - 9.0000000i$	2.0000000
$y =$	$4.0000000 - 4.0000000i$	$0.0000000 - 5.0000000i$
	$0.0000000 - 7.0000000i$	$8.0000000 + 2.0000000i$

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cons

cons

Purpose Retrieves a character string from the keyboard.

Format `x = cons;`

Output The characters entered from the keyboard. The output will be of type string.

Portability UNIX
cons gets input from the active graphic panel. If you are working in terminal mode GAUSS will not “see” any input until you press ENTER.

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. The standard DOS editing keys will be in effect.

Example `x = cons;`
At the cursor enter:
 probability
`x = “probability”`

See also **con**

continue

Purpose Jumps to the top of a **do** or **for** loop.

Format `continue;`

Remarks This command works just like in C.

Example

```
x = rndn(4,4);
r = 0;
do while r < rows(x);
    r = r + 1;
    c = 0;
    do while c < cols(x); /* continue jumps here */
        c = c + 1;
        if c == r;
            continue;
        endif;
        x[r,c] = 0;
    endo;
endo;
```

```

      -1.032195  0000000 0.000000  0.000000
x =  0.000000 -1.033763 0.000000  0.000000
      0.000000  0.000000 0.061205  0.000000
      0.000000  0.000000 0.000000 -0.225936
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

contour

contour

Purpose To graph a matrix of contour data.

Library pgraph

Format `contour(x,y,z);`

Input

<i>x</i>	1xK vector, the X axis data. K must be odd.
<i>y</i>	Nx1 vector, the Y axis data. N must be odd.
<i>z</i>	NxK matrix, the matrix of height data to be plotted.
<code>_plev</code>	Kx1 vector, user-defined contour levels for contour . Default 0.
<code>_pzc1r</code>	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 $c = \text{conv}(b, x, f, l);$

Input

b	$N \times 1$ vector.
x	$L \times 1$ vector.
f	scalar, the first convolution to compute.
l	scalar, the last convolution to compute.

Output

c	$Q \times 1$ 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.

Example

```
x = { 1, 2, 3, 4 };
y = { 5, 6, 7, 8 };
z1 = conv(x, y, 0, 0);
z2 = conv(x, y, 2, 5);
```

```

      5
      16
      34
z1 = 60
      61
      52
      32
```

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

conv

$$\begin{array}{rcl} & & 16 \\ z2 = & 34 & \\ & 60 & \\ & 61 & \end{array}$$

See also `polymult`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

coreleft

Purpose	Returns the amount, in bytes, of free workspace memory.
Format	$y = \text{coreleft};$
Output	y scalar, number of bytes free.
Portability	DOS only. All others will return the coreleft value specified in the GUASS configuration (<code>.cfg</code>) file.
Remarks	The amount of free memory is dynamic and can change rapidly as expressions and procedures are being executed. coreleft returns the amount of workspace memory free at the time it is called. Workspace memory is used for storing matrices, strings, procedures, and for manipulating matrices and strings. This function can be used to write programs that automatically adjust their use of memory so they do not crash with the “Insufficient memory” error if they are used on machines with less free memory than the one used for development, or if the size of the data used becomes larger. A common use is to adjust the number of rows that are read per iteration of a read loop in programs that access data from a disk.
Example	<pre>open fp = myfile; k = colsf(fp); /* columns in file */ fac = 4; /* check amount of memory available */ nr = coreleft/(fac*k*8);</pre> In this example, nr, the number of rows to read, is computed by taking the number of bytes free (coreleft) divided by fac*k*8. fac is a guesstimate of the number of copies of the data read each iteration that the algorithm we are using will require, plus a little. k*8 is the number of columns times the number of bytes per element.
See also	dfree , new

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

corrmm, corrvc, corrx

corrmm, corrvc, corrx

Purpose	Computes a correlation matrix.	
Format	$cx = \text{corrmm}(m);$ $cx = \text{corrvc}(vc);$ $cx = \text{corrx}(x);$	
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 corrmm , $P = K-1$. For corrvc and corrx , $P = K$.
Source	corr.src	
See also	momentd	

COS

Purpose Returns the cosine of its argument.

Format $y = \cos(x);$

Input x NxK matrix.

Output y NxK matrix.

Remarks For real matrices, x should contain angles measured in radians.

To convert degrees to radians, multiply the degrees by $\frac{\pi}{180}$.

Example $x = \{ 0, .5, 1, 1.5 \};$
 $y = \cos(x);$

$y =$
1.00000000
0.87758256
0.54030231
0.07073720

See also `atan`, `atan2`, `pi`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cosh

cosh

Purpose Computes the hyperbolic cosine.

Format $y = \cosh(x);$

Input x N x K matrix.

Output y N x K matrix containing the hyperbolic cosines of the elements of x .

Example $x = \{ -0.5, -0.25, 0, 0.25, 0.5, 1 \};$
 $x = x * \pi;$
 $y = \cosh(x);$

$x =$
-1.570796
-0.785398
0.000000
0.785398
1.570796
3.141593

 $y =$
2.509178
1.324609
1.000000
1.324609
2.509178
11.591953

Source trig.src

counts

Purpose	Count the numbers of elements of a vector that fall into specified ranges.		
Format	$c = \text{counts}(x, v);$		
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: $\begin{array}{l} x \leq v[1], \\ v[1] < x \leq v[2], \\ \cdot \\ \cdot \\ \cdot \\ v[p-1] < x \leq v[p]. \end{array}$	
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 $x = \begin{array}{l} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \text{ and } v = \begin{array}{l} 4 \\ 5 \\ 8 \end{array}$		

counts

then

$$c = \begin{matrix} 4 \\ 1 \\ 3 \end{matrix}$$

The first category can be a missing value if you need to count missings directly. Also $+\infty$ or $-\infty$ 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. $-\infty$ 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 $+\infty$ is included, it must be the last element of the breakpoint vector.

Example

```
x = { 1, 3, 2,
      4, 1, 3 };
v = { 0, 1, 2, 3, 4 };

c = counts(x,v);

0.0000000
2.0000000
c = 1.0000000
2.0000000
1.0000000
```


countwts

Purpose	Returns a weighted count of the numbers of elements of a vector that fall into specified ranges.		
Format	$c = \text{countwts}(x, v, w);$		
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, the counts of the elements of x that fall into the regions:	
		$\begin{aligned} &x \leq v[1], \\ &v[1] < x \leq v[2], \\ &\quad \cdot \\ &\quad \cdot \\ &\quad \cdot \\ &v[p-1] < x \leq v[p]. \end{aligned}$	
		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 .		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

countwts

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);

0.000000
0.500000
c = 0.333000
2.00000
0.100000
```

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

create

Purpose Creates and opens a GAUSS data set for subsequent writing.

Format `create` *[[vflag]]* **[[complex]]** *fh = filename with*
vnames, col, dtyp, vtyp;
`create` *[[vflag]]* **[[complex]]** *fh = filename using comfile;*

Input *vflag* version flag.
 -**v89** supported on DOS, OS/2, and Windows NT
 -**v92** supported on UNIX and Windows
 -**v96** supported on all platforms

For details on the various versions, see “File I/O” in the *User’s Guide*. 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**.

filename literal or ^string.

filename is the name to be given 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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

create

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 ≤ 32767
single	6-7	$8.43 \times 10^{-37} \leq X \leq 3.37 \times 10^{+38}$
double	15-16	$4.19 \times 10^{-307} \leq X \leq 1.67 \times 10^{+308}$

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. *vtyp* is ignored for **v89** files.

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;
```

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.

create

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.

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;`
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**.
Following are examples of the variable names that will result when using a character vector of names in the argument to the **create** function.

create

```

vnames = { AGE PAY SEX JOB };
typ = { 1, 1, 0, 0 };
create fp = mydata with ^vnames,0,2,typ;

```

The names in 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 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;

```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

create

For platforms using the backslash as a path separator, remember that two backslashes (\\) are required to enter one backslash inside double quotes. This is because a backslash is the escape character used to embed special characters in strings.

See also `open`, `readr`, `writer`, `eof`, `close`, `output`, `iscplxf`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

crossprd

Purpose Computes the cross-products (vector products) of sets of 3x1 vectors.

Format $z = \text{crossprd}(x, y);$

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);

      7.0000000 -39.000000
z =  -28.000000 107.00000
      17.000000 -95.000000

```

Source crossprd.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

crout

crout

Purpose Computes the Crout decomposition of a square matrix without row pivoting, such that: $X = LU$.

Format $y = \text{crout}(x);$

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 $X = \begin{Bmatrix} 1 & 2 & -1, \\ 2 & 3 & -2, \\ 1 & -2 & 1 \end{Bmatrix};$

$y = \text{crout}(x);$

$L = \text{lowmat}(y);$

$U = \text{upmat1}(y);$

$y = \begin{Bmatrix} 1 & 2 & -1 \\ 2 & -1 & 0 \\ 1 & -4 & 2 \end{Bmatrix}$

$L = \begin{Bmatrix} 1 & 0 & 0 \\ 2 & -1 & 0 \\ 1 & -4 & 2 \end{Bmatrix}$

crout

$$U = \begin{pmatrix} 1 & 2 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

See also `croutp`, `chol`, `lowmat`, `lowmat1`, `lu`, `upmat`, `upmat1`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

croutp

croutp

Purpose Computes the Crout decomposition of a square matrix with partial (row) pivoting.

Format $y = \text{croutp}(x);$

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);
r = rows(y);    /* the number of rows of y */
indx = y[r,.]'; /* get the index vector */
z = y[indx,.];  /* z is indexed RxR matrix y */
L = lowmat(z); /* obtain L and U of permuted */
               /* matrix X */
U = upmat1(z);
q = sortc(indx~(L*U),1); /* sort L*U against */
                        /* index */
x2 = q[:,2:cols(q)]; /* remove index column */
```

$$X = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 3 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

$$y = \begin{bmatrix} 1 & 0.5 & 0.2857 \\ 2 & 1.5 & -1 \\ 1 & -3.5 & -0.5714 \\ 2 & 3 & 1 \end{bmatrix}$$

$$r = 4$$

$$\text{indx} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$$

$$z = \begin{bmatrix} 2 & 1.5 & -1 \\ 1 & -3.5 & -0.5714 \\ 1 & 0.5 & 0.2857 \end{bmatrix}$$

$$L = \begin{bmatrix} 2 & 0 & 0 \\ 1 & -3.5 & 0 \\ 1 & 0.5 & 0.2857 \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & 1.5 & -1 \\ 0 & 1 & -0.5714 \\ 0 & 0 & 1 \end{bmatrix}$$

$$q = \begin{bmatrix} 1 & 1 & 2 & -1 \\ 2 & 2 & 3 & -2 \\ 3 & 1 & -2 & 1 \end{bmatrix}$$

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

croutp

$$x2 = \begin{pmatrix} 1 & 2 & -1 \\ 2 & 3 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

See also `crout`, `chol`, `lowmat`, `lowmat1`, `lu`, `upmat`, `upmat1`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

csrcol, csrln

Purpose Returns the position of the cursor.

Format `y = csrcol;`
`y = csrln;`

Portability UNIX

csrcol returns the cursor column for the active graphic panel. For Text graphic panels, this value is the cursor column with respect to the text buffer. For TTY graphic panels, this value is the cursor column with respect to the current output line, i.e., it will be the same whether the text is wrapped or not. For PQG graphic panels, this value is meaningless.

csrln returns the cursor line for the active graphic panel. For Text graphic panels, this value is the cursor row with respect to the text buffer. For TTY graphic panels, this value is the current output line number (i.e., the number of lines logged + 1). For PQG graphic panels, this value is meaningless.

OS/2, Windows

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.

csrln returns the cursor line with respect to the top line in the graphic panel.

DOS

Under DOS, columns are usually numbered 1-80, rows are usually numbered 1-25. **setvmode** will return the current window dimensions.

Remarks `y` will contain the current column or row position of the cursor in the graphic panel. The upper left corner is (1,1).

csrcol returns the column position of the cursor. **csrln** returns the row position.

The **locate** statement allows the cursor to be positioned at a specific row and column.

Example `r = csrln;`
`c = csrcol;`
`cls;`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

csrcol, csrln

```
locate r,c;
```

In this example the window is cleared without affecting the cursor position.

See also **cls, locate, lpos**

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

csrtype

Purpose To set the cursor shape.

Format *old* = **csrtype**(*mode*);

Portability UNIX

This function is not supported in terminal mode.

OS/2, Windows

This function is not supported under OS/2 or Windows.

Input *mode* scalar, cursor type to set.

DOS	
0	cursor off
1	normal cursor
2	large cursor
UNIX	
0	cursor off
1	normal cursor
2	large cursor
3	triangular cursor

Output *old* scalar, original cursor type.

Remarks Under DOS, this function will set the same cursor shape that GAUSS is already using for its three modes.

Example `x = csrtype(2);`

See also `csrcol`, `csrln`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

cumprodc

cumprodc

Purpose	Computes the cumulative products of the columns of a matrix.
Format	$y = \text{cumprodc}(x);$
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: recserp(x, zeros(1, cols(x))) to accomplish the same thing.
Example	$x = \begin{Bmatrix} 1 & -3, \\ 2 & 2, \\ 3 & -1 \end{Bmatrix};$ $y = \text{cumprodc}(x);$ $y = \begin{Bmatrix} 1.00 & -3.00 \\ 2.00 & -6.00 \\ 6.00 & 6.00 \end{Bmatrix}$
Source	cumprodc.src
See also	cumsumc, recsercp, recserar

cumsumc

Purpose	Computes the cumulative sums of the columns of a matrix.
Format	$y = \text{cumsumc}(x);$
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: <pre>recserar(x, x[1, .], ones(1, cols(x)))</pre> to accomplish the same thing.
Example	<pre>x = { 1 -3, 2 2, 3 -1 };</pre> <pre>y = cumsumc(x);</pre> <pre> 1 -3 y = 3 -1 6 -2</pre>
Source	cumsumc.src
See also	cumprodc, recsercp, recserar

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

curve

curve

Purpose Computes a one-dimensional smoothing curve.

Format `{ u, v } = curve(x, y, d, s, sigma, G);`

Input

<i>x</i>	Kx1 vector, x-abcissae (x-axis values).
<i>y</i>	Kx1 vector, y-ordinates (y-axis values).
<i>d</i>	Kx1 vector or scalar, observation weights.
<i>s</i>	scalar, smoothing parameter. If $s = 0$, curve performs an interpolation. If <i>d</i> contains standard deviation estimates, a reasonable value for <i>s</i> is K.
<i>sigma</i>	scalar, tension factor.
<i>G</i>	scalar, grid size factor.

Output

<i>u</i>	K*Gx1 vector, x-abcissae, regularly spaced.
<i>v</i>	K*Gx1 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);`

`s = "Now is the time for all good men"`

See also `stocv`, `vget`, `vlist`, `vput`, `vread`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

datalist

datalist

Purpose	List selected variables from a data set.		
Format	datalist <i>dataset</i> <code>[[var1 [[var2...]]]] ;</code>		
Input	<i>dataset</i>	literal, name of the dataset.	
	<i>var#</i>	literal, the names of the variables to list.	
	__range	global scalar, the range of rows to list. The default is all rows.	
	__miss	global 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	global 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 in the window 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		

dataloop (dataloop)

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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

date

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;`

1998.0000
6.0000000
15.000000
4011252.7

See also `time, timestr, ethsec, hsec, etstr`

datestr

Purpose	Returns a date in a string.
Format	<i>str</i> = datestr (<i>d</i>);
Input	<i>d</i> 4x1 vector, like the date function returns. If this is 0, the date function will be called for the current system date.
Output	<i>str</i> 8 character string containing current date in the form: mo/dy/yr
Example	<pre>d = { 1998, 6, 15, 0 }; y = datestr(d); print y;</pre> 6/15/98
Source	time.src
See also	date , datestring , datestrymd , time , timestr , ethsec

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

datestring

datestring

Purpose Returns a date in a year-2000-compliant string.

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

```
y = datestring(0);  
print y;
```

6/15/1998

Source time.src

See also **date**, **datestr**, **datestrymd**, **time**, **timestr**, **ethsec**

datestrymd

Purpose	Returns a date in a string.
Format	<i>str</i> = datestrymd (<i>d</i>);
Input	<i>d</i> 4x1 vector, like the date function returns. If this is 0, the date function will be called for the current system date.
Output	<i>str</i> 8 character string containing current date in the form: yyyymmdd
Example	<pre> d = { 1998, 6, 15, 0 }; y = datestrymd(d); print y; 19980615 </pre>
Source	time.src
See also	date , datestr , datestring , time , timestr , ethsec

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dayinyr

dayinyr

Purpose	Returns day number in the year of a given date.
Format	<i>daynum</i> = dayinyr (<i>dt</i>);
Input	<i>dt</i> 3x1 or 4x1 vector, date to check. The date should be in the form returned by date .
Output	<i>daynum</i> scalar, the day number of that date in that year.
Example	<pre>x = { 1998, 6, 15, 0 }; y = dayinyr(x); print y; 166.00000</pre>
Source	time.src
Globals	<code>_isleap</code>

debug

Purpose Runs a program under the source level debugger.

Format `debug filename;`

Input *filename* Literal or ^string, name of file to debug.

Remarks See “Debugging” in the *User’s Guide*.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

declare

declare

Purpose	To initialize matrices and strings at compile time.		
Format	declare <i>[[type] symbol [aop clist]</i> ;		
Input	<i>type</i>	optional literal, specifying the type of the symbol. matrix string if <i>type</i> is not specified, matrix is assumed.	
	<i>symbol</i>	the name of the symbol being declared.	
	<i>aop</i>	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 <i>aop</i> is specified, <i>clist</i> must be also.	
	<i>clist</i>	a list of constants to assign to <i>symbol</i> . If <i>aop clist</i> is not specified, <i>symbol</i> 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 matrices and strings 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. 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		

declare

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 listed in the symbol table when the file containing the **declare** statement is compiled. The symbol will be reset.

This allows mixing **declare** statements with the procedure definitions that reference the global matrices and strings, or placing them in your main file.

:= Redefinition is treated as an error. This will not allow you to assign one symbol with another value already in your program. Rename one of them.

Place **declare** statements in a separate file from the rest of your program and procedure definitions.

?= Interactive programming where some global defaults were set when you started and you do not want them reset for each successive run even if the file containing the **declare**'s gets recompiled. Be careful when using.

CTRL+W controls the **declare** statement warning level. If **declare** warnings are on, you will be warned whenever a **declare** statement encounters a symbol that is already initialized. This happens when you declare a symbol that is already initialized when **declare** warnings are turned on:

declare != Reinitialize and warn.

declare := Crash 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;`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

declare

```
x = 0
```

```
y = 0
```

```
z = 0
```

```
declare string x = "This string.";
```

```
x = "This string."
```

```
declare matrix x;
```

```
x = 0
```

```
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

```
declare matrix x = 1 2 3 4 5 6 7 8 9;
```

```
1
```

```
2
```

```
3
```

```
4
```

```
x = 5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
declare matrix x = dog cat;
```

```
x = DOG  
    CAT
```

```
declare matrix x = "dog" "cat";
```

```
x = dog  
    cat
```

See also **let, external**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

delete

delete

Purpose	Deletes global symbols from the symbol table.		
Format	delete <i>[-flags]</i> <i>[[symbol]]</i> <i>[[symbol2]]</i> <i>[[symbol3]]</i> ;		
Input	<i>flags</i>	specify the type(s) of symbols to be deleted	
	p	procedures	
	k	keywords	
	f	fn functions	
	m	matrices	
	s	strings	
	g	only symbols with global references	
	l	only symbols with all local references	
	n	no pause for confirmation	
	<i>symbol</i>	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 symbols from GAUSS's memory and workspace.		
	Flags must be preceded by a slash (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 will be deleted as well.		
Example	print x;		
	96.000000		
	6.0000000		
	14.000000		
	3502965.9		

delete

```
delete -m x;
```

At the Delete? [Yes No Previous Quit] prompt, enter y.

```
show x;
```

x no longer exists.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

delete (dataloop)

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 externs, 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., $\leq$) are already interpreted in terms of their dot equivalents ($\cdot \leq$), 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`

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

<i>x</i>	NxK data matrix.
<i>e</i>	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:

```
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);
```

Here is the resulting matrix **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.

Source `datatran.src`

See also `selif`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

denseSubmat

denseSubmat

Purpose Returns dense submatrix of sparse matrix.

Format $e = \text{denseSubmat}(x, r, c);$

Input

x	MxN sparse matrix.
r	Kx1 vector, row indices.
c	Lx1 vector, column indices.

Output e KxL dense matrix.

Remarks If r or c are scalar zeros, all rows or columns will be returned.

Source `sparse.src`

See also `sparseFd`, `sparseFp`

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	$y = \text{design}(x);$
Input	x Nx1 vector.
Output	y NxK matrix, where $K = \max c(x)$; 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	<pre> x = { 1, 1.2, 2, 3, 4.4 }; y = design(x); 1 0 0 0 1 0 0 0 y = 0 1 0 0 0 0 1 0 0 0 0 1 </pre>
Source	design.src
See also	cumprodc, cumsumc, recserrc

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

det

det

Purpose Returns the determinant of a square matrix.

Format $y = \text{det}(x);$

Input x NxN square matrix.

Output y determinant of x .

Remarks x may be any valid expression that returns a square matrix (number of rows equals number of columns).

det computes a LU decomposition.

detl can be much faster in many applications.

Example $x = \begin{Bmatrix} 3 & 2 & 1, \\ 0 & 1 & -2, \\ 1 & 3 & 4 \end{Bmatrix};$

$y = \text{det}(x);$

$x = \begin{Bmatrix} 3 & 2 & 1 \\ 0 & 1 & -2 \\ 1 & 3 & 4 \end{Bmatrix}$

$y = 25$

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 following functions 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*

y/x determinant of *x* when neither argument is a scalar
or

determinant of $x'x$ if *x* is not square

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:

```
call chol(x);
```

```
xd = detl;
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

det l

The Cholesky decomposition is computed and the result from that is discarded. The determinant saved during that instruction is retrieved using **det l**. This can execute up to 2.5 times faster than **det (x)** for large positive definite matrices.

See also **det**

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dffti

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**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dfree (DOS only)

Purpose	Returns the amount of room left on a diskette or hard disk.
Format	<i>y</i> = dfree (<i>drive</i>);
Input	<i>drive</i> scalar, valid disk drive number.
Output	<i>y</i> number of bytes free.
Portability	All others return -.1
Remarks	Valid disk drive numbers are 0 = default, 1 = A, 2 = B, etc. If an error is encountered, dfree will return -.1.
See also	coreleft

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

diag

diag

Purpose	Creates a column vector from the diagonal of a matrix.
Format	$y = \text{diag}(x);$
Input	x NxK matrix.
Output	y min(N,K)x1 vector.
Remarks	The matrix need not be square. diagrv reverses the procedure and puts a vector into the diagonal of a matrix.
Example	$x = \text{rndu}(3, 3);$ $y = \text{diag}(x);$ $x = \begin{bmatrix} 0.660818 & 0.367424 & 0.302208 \\ 0.204800 & 0.077357 & 0.145755 \\ 0.712284 & 0.353760 & 0.642567 \end{bmatrix}$ $y = \begin{bmatrix} 0.660818 \\ 0.077357 \\ 0.642567 \end{bmatrix}$
See also	diagrv

diagrv

Purpose Inserts a vector into the diagonal of a matrix.

Format `y = diagrv(x, v);`

Input

<i>x</i>	NxK matrix.
<i>v</i>	min(N,K) 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 = randu(3, 3);
v = ones(3, 1);
y = diagrv(x, v);
```

	0.660818	0.367424	0.302208
<i>x</i> =	0.204800	0.077357	0.145755
	0.712284	0.353760	0.642567
	1.000000		
<i>v</i> =	1.000000		
	1.000000		
	1.000000	0.367424	0.302208
<i>y</i> =	0.204800	1.000000	0.145755
	0.712284	0.353760	1.000000

See also **diag**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

disable

disable

This command is obsolete. See “**disable**” on page A-2.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dlibrary

Purpose Dynamically links and unlinks shared libraries.

Format **dlibrary** *lib1* [*lib2...*];
dlibrary -a *lib1* [*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 **.dll**. 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 DLL’s listed are added to the current set of DLL’s rather than replacing them. For search purposes, the new DLL’s follow the already active ones. Without the **-a** flag, any previously linked libraries are dumped.

-d dump flag, ALL DLL’s 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 DLL’s listed are linked into GAUSS and any previously linked libraries are dumped. When you call **dllcall**, the DLL’s 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 DLL’s. 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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dlibrary

directories. A new case (case 24) has also been added to **sysstate** for getting and setting this default.

GAUSS maintains its own DLL, `gauss.dll`. `gauss.dll` is listed when you execute **dlibrary** with no arguments, and searched when you call **dllcall**. By default, `gauss.dll` is searched last, after all other DLL's but you can force it to be searched earlier by listing it explicitly in a **dlibrary** statement. `gauss.dll` is always active. It is not unlinked when you execute **dlibrary -d**. `gauss.dll` is located in the `gauss.exe` directory.

See also `dllcall`, `sysstate-case 24`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dllcall

Purpose Calls functions located in dynamic libraries.

Format `dllcall [-r] [-v] func[(arg1[,arg2...])];`

dllcall works in conjunction with **dlibrary**. **dlibrary** is used to link dynamic-link libraries (DLL's) into GAUSS; **dllcall** is used to access the functions contained in those DLL's. **dllcall** searches the DLL's (see **dlibrary** for an explanation of the search order) for a function named *func*, and calls the first instance it finds. The default DLL, `gauss.dll`, is searched last.

Input

func the name of a function contained in a DLL (linked into GAUSS with **dlibrary**). If *func* is not specified or cannot be located in a DLL, **dllcall** will fail.

arg# arguments to be passed to *func*; optional. These must be elementary variable; 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[,double *arg2...]);`
3. `int func (double *argv[]);`

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

dllcall

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.

See also **dlibrary, sysstate-case 24**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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, following.

do loops may be nested.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

do while, do until

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;

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 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;
```

do while, do until

```
i = 1;  
do while i <= 10;  
  print i;;  
  i = i+1;  
endo;
```

produces

```
1 2 3 4 5 6 7 8 9 10
```

```
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**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dos

dos

Purpose Provides access to the operating system from within GAUSS.

Format `dos  $\llbracket s \rrbracket$ ;`

Input *s* literal or ^string, the OS command to be executed.

Portability **UNIX**

Control and output go to the controlling terminal, if there is one.

This function may be used in terminal mode.

OS/2, Windows

The **dos** function opens a new terminal.

Running programs in the background is allowed in all three 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, 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 **DOS** 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 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 the word "DOS".

Example `comstr = "basic myprog";`
 `dos ^comstr;`

This will cause the BASIC program **myprog** to be run. When that program is finished, control will automatically return to GAUSS.

`>dir *.prg;`

This will use the DOS **dir** command to print a directory listing of all files with a `.prg` extension. When the listing is finished, control will be returned to GAUSS.

`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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

doswin

doswin

Purpose Opens the DOS compatibility window with default settings.

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.
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	<pre>let attr = 50 50 7 0 7; if not DOSWinOpen("Legacy Window", attr); errorlog "Failed to open DOS window, aborting"; stop; endif; . . DOSWinCloseall;</pre>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

DOSWinOpen

DOSWinOpen

Purpose Opens the DOS compatibility window and gives it the specified title and attributes.

Format `ret = DOSWinOpen(title, attr);`

Portability Windows only

Input

<i>title</i>	string, window title.
<i>attr</i>	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

<i>ret</i>	scalar, success flag, 1 if successful, 0 if not.

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

[1]	<i>varies</i>	use x position of previous DOS window
[2]	<i>varies</i>	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:

DOSWinOpen

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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dotfeq-dotfne

dotfeq-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` global 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;`

dotfreq-dotfne

Example `x = rndu(2,2);`
 `y = rndu(2,2);`
 `t = dotfge(x,y);`

`x =    0.85115559 0.98914218`
 `0.12703276 0.43365175`

`y =    0.41907226 0.49648058`
 `0.58039125 0.98200340`

`t =    1.00000000 1.00000000`
 `0.00000000 0.00000000`

Source `fcompare.src`

Globals `_fcmptol`

See also `freq-fne`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

draw

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.

Library **pgraph**

Format **draw;**

Remarks **draw** is especially useful when used in conjunction with transparent graphic panels.

Example `library pgraph;
graphset;
begwind;
/* make full size window for plot */
makewind(9,6.855,0,0,0);
/* make small overlapping window for text*/
makewind(3,1,3,3,0);
 setwind(1);
 x = seqa(.1,.1,100);
 y = sin(x);
 /* plot data in first window*/
 xy(x,y);`

draw

```
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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

drop (dataloop)

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`

dstat

Purpose Compute descriptive statistics.

Format { *vnam,mean,var,std,min,max,valid,mis* } = **dstat**(*dataset,vars*);

Input *dataset* string, name of data set.

If *dataset* contains the name of a GAUSS data set, *vars* will be interpreted as:

vars the variables.

If *dataset* is null or 0, *vars* will be assumed to be a matrix containing the data.

Kx1 character vector names of variables.

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.

If *dataset* is null or 0, *vars* will be interpreted as:

NxK matrix the data on which to compute the descriptive statistics.

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** global matrix, default 0.

___**miss** This can be a Kx1 character vector of alternate variable names for the output.

global 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** global scalar, the number of rows to read per iteration of the read loop.

if 0, (default) the number of rows will be calculated internally.

dstat

__output global 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.

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

Globals __output, _dstatd, _dstatx

dtvnormal

Purpose	Normalizes a date and time vector.
Format	<code>d = dtvnormal(t);</code>
Input	<i>t</i> 1x8 date and time vector that has one or more elements outside the normal range.
Output	<i>d</i> 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. The last two elements are ignored on input.
Example	<pre>format /rd 10,2; x = { 1996 14 21 6 21 37 0 0 }; d = dtvnormal(x); d = 97.00 2.00 21.00 6.00 21.00 37.00 2.00 51.00</pre>
See also	<code>date</code> , <code>ethsec</code> , <code>etstr</code> , <code>time</code> , <code>timestr</code> , <code>timeutc</code> , <code>utctodtv</code>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dummy

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

<i>x</i>	Nx1 vector of data that is to be broken up into dummy variables.
<i>v</i>	(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

<i>y</i>	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

```
x = { 0, 2, 4, 6 };
v = { 1, 5, 7 };
y = dummy(x, v);
```

The result *y* looks like this:

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

The vector *v* will produce 4 dummies satisfying the following conditions:

$$x \leq 1$$

$$1 < x \leq 5$$

$$5 < x \leq 7$$

$$7 < x$$

Source `datatran.src`

See also `dummybr`, `dummydn`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dummybr

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

`x = { 0,
2,
4,
6};`

`v = { 1,
5,
7};`

`y = dummybr(x, v);`

The resulting matrix y looks like this:


```
1 0 0
```

```
0 1 0
```

```
0 1 0
```

```
0 0 1
```

The vector $v = 1\ 5\ 7$ will produce 3 dummies satisfying the following conditions:

$$x \leq 1$$

$$1 < x \leq 5$$

$$5 < x \leq 7$$

Source `datatran.src`

See also `dummydn`, `dummy`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

dummydn

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 $y = \text{dummydn}(x, v, p);$

Input

x Nx1 vector of data to be broken up into dummy variables.

v Kx1 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

```
x = { 0, 2, 4, 6 };
v = { 1, 5, 7 };
p = 2;
y = dummydn(x, v, p);
```

The resulting matrix y looks like this:

```
1 0 0
```

```
0 0 0
```

```
0 0 0
```

```
0 1 0
```

The vector $v = 1\ 5\ 7$ will produce 4 dummies satisfying the following conditions:

$$x \leq 1$$
$$1 < x \leq 5$$
$$5 < x \leq 7$$
$$7 < x$$

Source `datatran.src`

See also `dummy`, `dummybr`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ed

Purpose Accesses an alternate editor.

Format `ed filename;`

Input *filename* The name of the file to be edited.

Remarks The default name of the editor is `b.exe`. To change the name of the editor used type:

`ed = editor_name flags;`

or

`ed = "editor_name flags";`

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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

edit

edit

- Purpose** Edits a disk file.
- Format** `edit filename;`
- Example** `edit b:test1;`
- See also** `run`, `medit`, `editm`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

editm

This command is obsolete. See “**editm**” on page A-3.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

eig

eig

Purpose Computes the eigenvalues of a general matrix.

Format `va = eig(x);`

Input `x` NxN matrix.

Output `va` Nx1 vector, the eigenvalues of `x`.

Remarks 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

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);
```

```
      -4.4979246
va =   14.475702
      5.0222223
```

See also **eigh**, **eighv**, **eigv**

eigcg

Purpose	Computes the eigenvalues of a complex, general matrix. (Included for backwards compatibility — use eig instead.)		
Format	{ <i>var</i> , <i>vai</i> } = eigcg (<i>xr</i> , <i>xi</i>);		
Input	<i>xr</i>	NxN matrix, real part.	
	<i>xi</i>	NxN matrix, imaginary part.	
Output	<i>var</i>	Nx1 vector, real part of eigenvalues.	
	<i>vai</i>	Nx1 vector, imaginary part of eigenvalues.	
	_eigerr	global scalar, if all the eigenvalues can be determined _eigerr = 0, otherwise _eigerr is set to the index of the eigenvalue that failed. The eigenvalues for indices _eigerr +1 to N should be correct.	
Remarks	Error handling is controlled with the low bit of the trap flag.		
	trap 0	set _eigerr and terminate with message	
	trap 1	set _eigerr and continue execution	
	The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.		
Source	eigcg.src		
Globals	_eigerr		
See also	eigcg2 , eigch , eigr , eigrs		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

eigcg2

eigcg2

Purpose Computes eigenvalues and eigenvectors of a complex, general matrix. (Included for backwards compatibility — use **eigv** instead.)

Format `{ var, vai, ver, vei } = eigcg2(xr, xi);`

Input

<i>xr</i>	NxN matrix, real part.
<i>xi</i>	NxN matrix, imaginary part.

Output

<i>var</i>	Nx1 vector, real part of eigenvalues.
<i>vai</i>	Nx1 vector, imaginary part of eigenvalues.
<i>ver</i>	NxN matrix, real part of eigenvectors.
<i>vei</i>	NxN matrix, imaginary part of eigenvectors.
_eigerr	global scalar, if all the eigenvalues can be determined _eigerr = 0, otherwise _eigerr is set to the index of the eigenvalue that failed. The eigenvalues for indices _eigerr +1 to N should be correct. The eigenvectors are not computed.

Remarks Error handling is controlled with the low bit of the trap flag.

trap 0 set **_eigerr** and terminate with message

trap 1 set **_eigerr** and continue execution

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 *ver* and *vei* contain the real and imaginary eigenvectors of *x* in the same order as the eigenvalues. The eigenvectors are not normalized.

Source eigcg.src

Globals **_eigerr**

See also **eigcg**, **eigch**, **eigr**, **eigrs**

eigch

Purpose	Computes the eigenvalues of a complex, hermitian matrix. (Included for backwards compatibility — use eigh instead.)
Format	<code>va = eigch(xr,xi);</code>
Input	<i>xr</i> NxN matrix, real part. <i>xi</i> NxN matrix, imaginary part.
Output	<i>va</i> Nx1 vector, real part of eigenvalues. _eigerr global scalar, if all the eigenvalues can be determined _eigerr = 0, otherwise _eigerr is set to the index of the eigenvalue that failed. The eigenvalues for indices 1 to _eigerr-1 should be correct.
Remarks	Error handling is controlled with the low bit of the trap flag. trap 0 set _eigerr and terminate with message trap 1 set _eigerr and continue execution The eigenvalues are in ascending order. The eigenvalues for a complex hermitian matrix are always real so this procedure returns only one vector.
Source	<code>eigch.src</code>
Globals	_eigerr
See also	eigch2, eigcg, eigrg, eigrs

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

eigch2

eigch2

Purpose Computes eigenvalues and eigenvectors of a complex, hermitian matrix. (Included for backwards compatibility — use **eighv** instead.)

Format `{ var, vai, ver, vei } = eigch2(xr, xi);`

Input

<i>xr</i>	NxN matrix, real part.
<i>xi</i>	NxN matrix, imaginary part.

Output

<i>var</i>	Nx1 vector, real part of eigenvalues.
<i>vai</i>	Nx1 vector, imaginary part of eigenvalues.
<i>ver</i>	NxN matrix, real part of eigenvectors.
<i>vei</i>	NxN matrix, imaginary part of eigenvectors.
_eigerr	global scalar, if all the eigenvalues can be determined _eigerr = 0, otherwise _eigerr is set to the index of the eigenvalue that failed. The eigenvalues for indices 1 to _eigerr -1 should be correct. The eigenvectors are not computed.

Remarks Error handling is controlled with the low bit of the trap flag.

trap 0 set **_eigerr** and terminate with message

trap 1 set **_eigerr** and continue execution

The eigenvalues are in ascending order. The eigenvalues of a complex hermitian matrix are always real. This procedure returns a vector of zeros for the imaginary part of the eigenvalues so the syntax is consistent with other **eigxx** procedure calls. The columns of *ver* and *vei* contain the real and imaginary eigenvectors of *x* in the same order as the eigenvalues. The eigenvectors are orthonormal.

Source eigch.src

Globals **_eigerr**

See also **eigch**, **eigcg**, **eigr**, **eigrs**

eigh

Purpose	Computes the eigenvalues of a complex hermitian or real symmetric matrix.
Format	$va = \text{eigh}(x);$
Input	x NxN matrix.
Output	va Nx1 vector, the eigenvalues of x .
Remarks	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 The eigenvalues are in ascending order. The eigenvalues of a complex hermitian or real symmetric matrix are always real.
See also	eig, eighv, eigv

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

eighv

eighv

Purpose Computes eigenvalues and eigenvectors of a complex hermitian or real symmetric matrix.

Format `{ va, ve } = eighv(x);`

Input *x* NxN matrix.

Output *va* Nx1 vector, the eigenvalues of *x*.
ve NxN matrix, the eigenvectors of *x*.

Remarks 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

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**

eigr

Purpose	Computes the eigenvalues of a real, general matrix. (Included for backwards compatibility — use eig instead.)
Format	{ var, vai } = eigr(x);
Input	<i>x</i> NxN matrix.
Output	<i>var</i> Nx1 vector, real part of eigenvalues. <i>vai</i> Nx1 vector, imaginary part of eigenvalues. _eigerr global scalar, if all the eigenvalues can be determined _eigerr = 0, otherwise _eigerr is set to the index of the eigenvalue that failed. The eigenvalues for indices _eigerr +1 to N should be correct.
Remarks	Error handling is controlled with the low bit of the trap flag. trap 0 set _eigerr and terminate with message trap 1 set _eigerr and continue execution The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.
Example	<pre> x = { 1 2i 3, 4i 5+3i 6, 7 8 9i }; {y,n} = eigr(x); -6.3836054 y = 2.0816489 10.301956 7.2292503 n = -1.4598755 6.2306252 </pre>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

eigr**g**

Source eigr_g.src

Globals _eigerr

See also eigr_g2, eigcg, eigch, eigrs

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

eigrg2

Purpose	Computes eigenvalues and eigenvectors of a real, general matrix. (Included for backwards compatibility — use eigv instead.)		
Format	{ <i>var</i> , <i>vai</i> , <i>ver</i> , <i>vei</i> } = eigrg2 (<i>x</i>);		
Input	<i>x</i>	NxN matrix.	
Output	<i>var</i>	Nx1 vector, real part of eigenvalues.	
	<i>vai</i>	Nx1 vector, imaginary part of eigenvalues.	
	<i>ver</i>	NxN matrix, real part of eigenvectors.	
	<i>vei</i>	NxN matrix, imaginary part of eigenvectors.	
	_eigerr	global scalar, if all the eigenvalues can be determined _eigerr = 0, otherwise _eigerr is set to the index of the eigenvalue that failed. The eigenvalues for indices _eigerr +1 to N should be correct. The eigenvectors are not computed.	
Remarks	Error handling is controlled with the low bit of the trap flag.		
	trap 0 set _eigerr and terminate with message		
	trap 1 set _eigerr and continue execution		
	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 <i>ver</i> and <i>vei</i> contain the real and imaginary eigenvectors of <i>x</i> in the same order as the eigenvalues. The eigenvectors are not normalized.		
Source	eigrg.src		
Globals	_eigerr		
See also	eigrg , eigcg , eigch , eigrs		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

eigrs

eigrs

Purpose	Computes the eigenvalues of a real, symmetric matrix. (Included for backwards compatibility — use eigh instead.)
Format	<code>va = eigrs(x);</code>
Input	<i>x</i> NXN matrix.
Output	<i>va</i> NX1 vector, eigenvalues of <i>x</i>. _eigerr global scalar, if all the eigenvalues can be determined _eigerr = 0, otherwise _eigerr is set to the index of the eigenvalue that failed. The eigenvalues for indices 1 to _eigerr-1 should be correct.
Remarks	Error handling is controlled with the low bit of the trap flag. trap 0 set _eigerr and terminate with message trap 1 set _eigerr and continue execution The eigenvalues are in ascending order. The eigenvalues for a real symmetric matrix are always real so this procedure returns only one vector.
Source	<code>eigrs.src</code>
Globals	_eigerr
See also	eigrs2, eigcg, eigch, eigrg

eigrs2

Purpose	Computes eigenvalues and eigenvectors of a real, symmetric matrix. (Included for backwards compatibility — use eighv instead.)
Format	{ va, ve } = eigrs2(x);
Input	<i>x</i> NxN matrix.
Output	<i>va</i> Nx1 vector, eigenvalues of <i>x</i> . <i>ve</i> NxN matrix, eigenvectors of <i>x</i> . _eigerr global scalar, if all the eigenvalues can be determined _eigerr = 0, otherwise _eigerr is set to the index of the eigenvalue that failed. The eigenvalues and eigenvectors for indices 1 to _eigerr -1 should be correct.
Remarks	Error handling is controlled with the low bit of the trap flag. trap 0 set _eigerr and terminate with message trap 1 set _eigerr and continue execution The eigenvalues are in ascending order. The columns of <i>ve</i> contain the eigenvectors of <i>x</i> in the same order as the eigenvalues. The eigenvectors are orthonormal. The eigenvalues and eigenvectors for a real symmetric matrix are always real so this procedure returns only the real parts.
Source	eigrs.src
Globals	_eigerr
See also	eigrs, eigcg, eigch, eigrg

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

eigv

eigv

Purpose Computes eigenvalues and eigenvectors of a general matrix.

Format `{ va, ve } = eigv(x);`

Input *x* NxN matrix.

Output *va* Nx1 vector, the eigenvalues of *x*.
ve NxN matrix, the eigenvectors of *x*.

Remarks 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

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);`

```

          -4.4979246
y =      14.475702
          5.0222223

          -0.66930459  -0.64076622  -0.40145623
n =      0.71335708   -0.72488533   -0.26047487
          -0.01915672  -0.91339349    1.6734214

```

See also `eig`, `eigh`, `eighv`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

enable

enable

This command is obsolete. See “**enable**” on page A-6.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

end

Purpose Terminates a program.

Format `end;`

Portability UNIX

end closes all graphic panels except graphic panel 1. If you want to leave a program's graphic panels open, use **stop**.

OS/2, Windows

This function closes all user-created graphic panels.

DOS

The window must be in text mode to enter commands, so it will be reset if it is in graphics mode. The window will revert to text mode whenever a program terminates.

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 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 will be turned back on.

See also `new`, `stop`, `system`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

endp

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” in the <i>User’s Guide</i> .)
Example	<pre>proc regress(y,x); retp(inv(x’x)*x’y); endp;</pre> <pre>x = { 1 3 2, 7 4 9, 1 1 6, 3 3 2 }; y = { 3, 5, 2, 7 }; b = regress(y,x); 1.00000000 3.00000000 2.00000000 7.00000000 4.00000000 9.00000000 1.00000000 1.00000000 6.00000000 3.00000000 3.00000000 2.00000000 3.00000000 5.00000000 2.00000000 7.00000000 0.15456890 1.50276345 -0.12840825</pre>

See also `proc`, `keyword`, `retp`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

endwind

endwind

Purpose	Ends graphic panel manipulation; display graphs with rerun .
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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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 or a null string if it is not found.

Example

```
proc dopen(file);
    local fname,fp;
    fname = envget("DPATH");
    if fname $== "";
        fname = file;
    else;
        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 which 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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

eof

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 the **readr** and **fgetsxxx** 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 (**x'x**) matrix 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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

eqSolve

__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.

Output

<i>x</i>	Kx1 vector, solution.
<i>retcode</i>	scalar, the return code: 1 Norm of the scaled function value is less than __Tol. <i>x</i> given is an approximate root of $F(x)$ (unless __Tol is too large). 2 The scaled distance between the last two steps is less than the step-tolerance (__eqs_StepTol). <i>x</i> may be an approximate root of $F(x)$, 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 $\text{norm2}(F(x))$ sufficiently; either <i>x</i> is close to a root of $F(x)$ 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 $\text{norm2}(F(x))$ asymptotes from above to a finite value in some direction or the maximum step length is too small. 6 <i>x</i> seems to be an approximate local minimizer of $\text{norm2}(F(x))$ that is not a root of $F(x)$. To find a root of $F(x)$, restart eqSolve from a different region.

Remarks

The equation procedure should return a column vector containing the result for each equation. For example:

Equation 1: $x1^2 + x2^2 - 2 = 0$

Equation 2: $\exp(x1-1) + x2^3 - 2 = 0$

```

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;

```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x

y

z

eqSolve

```
start = { -1, 12, -1 };
```

```
_eqs_JacobianProc = &fjc;
```

```
{ x,tcode } = eqSolve(&f,start);
```

Produces

```
=====
EqSolve Version 3.2.22          2/24/97   9:54 am
=====
||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-06
X2         12.00000   1.4085912  -6.6263102e-06
X3         -1.00000   1.1111111  4.4175402e-06
-----
```

Source eqsolve.src

eqSolveset

Purpose Sets global input used by **eqSolve** to default values.

Format `eqSolveset;`

Global Output	<code>__eqs_TypicalX</code>	<code>=0</code>
	<code>__eqs_TypicalF</code>	<code>=0</code>
	<code>__eqs_IterInfo</code>	<code>=0</code>
	<code>__eqs_JacobianProc</code>	<code>=0</code>
	<code>__eqs_MaxIters</code>	<code>=100</code>
	<code>__eqs_StepTol</code>	<code>__macheps^(2/3)</code>

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

erf, erfc

erf, erfc

Purpose Computes the Gaussian error function (**erf**) and its complement (**erfc**).

Format $y = \text{erf}(x);$
 $y = \text{erfc}(x);$

Input x NxK matrix.

Output y NxK matrix.

Remarks The allowable range for x is:

$$x \geq 0$$

The **erf** and **erfc** functions are closely related to the Normal distribution:

$$cdfn(x) = \begin{cases} \frac{1}{2} \left(1 + \text{erf} \left(\frac{x}{\sqrt{2}} \right) \right) & x \geq 0 \\ \frac{1}{2} \text{erfc} \left(\frac{-x}{\sqrt{2}} \right) & x < 0 \end{cases}$$

Example $x = \begin{Bmatrix} .5 & .4 & .3 \\ .6 & .8 & .3 \end{Bmatrix};$
 $y = \text{erf}(x);$

$y = \begin{matrix} 0.52049988 & 0.42839236 & 0.32862676 \\ 0.60385609 & 0.74210096 & 0.32862676 \end{matrix}$

$x = \begin{Bmatrix} .5 & .4 & .3 \\ .6 & .8 & .3 \end{Bmatrix};$
 $y = \text{erfc}(x);$

$y = \begin{matrix} 0.47950012 & 0.57160764 & 0.67137324 \\ 0.39614391 & 0.25789904 & 0.67137324 \end{matrix}$

erf, erfc

See also **cdfn, cdfnc**

Technical Notes **erf** and **erfc** are computed by summing the appropriate series and continued fractions. They are accurate to about 10 digits.

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

error**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;
```

error

```
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**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

errorlog

errorlog

Purpose Prints an error message to the window and error log file.

Format `errorlog str;`

Input *str* string, the error message to print.

Portability **UNIX**

This function prints to graphic panel 1 and the error log file.

OS/2, Windows

This function prints to the main GAUSS graphic panel and the error log file.

etdays

Purpose	Computes the difference between two times, as generated by the date command, in days.
Format	<i>days</i> = etdays (<i>tstart</i> , <i>tend</i>);
Input	<i>tstart</i> 3x1 or 4x1 vector, starting date, in the order: yr, mo, day. (Only the first 3 elements are used.) <i>tend</i> 3x1 or 4x1 vector, ending date, in the order: yr, mo, day. (Only the first 3 elements are used.) MUST be later than <i>tstart</i>.
Output	<i>days</i> 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 400.
Example	<pre>let date1 = 1986 1 2; let date2 = 1987 10 25; d = etdays(date1, date2); d = 661</pre>
Source	time.src
See also	dayinyr

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ethsec

ethsec

Purpose	Computes the difference between two times, as generated by the date command, in hundredths of a second.
Format	<i>hs</i> = ethsec (<i>tstart</i> , <i>tend</i>);
Input	<i>tstart</i> 4x1 vector, starting date, in the order: yr, mo, day, hundredths of a second. <i>tend</i> 4x1 vector, ending date, in the order: yr, mo, day, hundredths of a second. MUST be later than <i>tstart</i> .
Output	<i>hs</i> 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 400.
Example	<pre>let date1 = 1986 1 2 0; let date2 = 1987 10 25 0; t = ethsec(date1,date2); t = 5711040000</pre>
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 = { 86, 1, 2, 0 };
d2 = { 86, 2, 5, 815642 };
t = ethsec(d1,d2);
str = etstr(t);
t = 2.9457564e + 08
str = 34 days 2 hours 15 minutes 56.42 seconds
```

Source time.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

exctsmpl

exctsmpl

Purpose Computes a random subsample of a data set.

Format `n = exctsmpl(infile, outfile, percent);`

Input

<i>infile</i>	string, the name of the original data set.
<i>outfile</i>	string, the name of the data set to be created.
<i>percent</i>	scalar, the percentage random sample to take. This must be in the range 0-100.

Output

<i>n</i>	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("freq.dat", "rout", 30);`

`n = 30`

`freq.dat` is an example data set provided with GAUSS. Switching to the GAUSS examples directory will make it possible to do the above example as shown. Otherwise substitute data set names will need to be used.

Source `exctsmpl.src`

exec

Purpose Executes an executable program and returns the exit code to GAUSS.

Format `y = exec(program, comline);`

Input

<i>program</i>	string, the name of the program to be executed.
<i>comline</i>	string, the arguments to be placed on the command line of the program being executed.

Output *y* the exit code returned by *program*.
 If **exec** 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

Portability UNIX

All I/O goes to GAUSS's controlling terminal.

Windows

The file can be given an extension or a "." meaning no extension. Otherwise the file will be searched for with the following extensions in their given order: .com, .exe, .bat, .cmd.

OS/2

exec works the same as in Windows, except that the only default extension is .exe.

DOS

You must use the full name of the executable, including the extension.

This function uses MS-DOS functions 4B and 4D. The child process can return an exit code when it returns if it terminates with the MS-DOS function 4C. (For detailed information on these functions, see the *DOS Technical Reference* manual.)

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

exec**Remarks**

On some platforms, the exit code is truncated to a 2-byte signed integer and only the lower 8 bits will be returned. Therefore a -1 will be returned as 255. The operating system may also set the lower 2 bits of the high byte of the returned word.

Example

```
y = exec("atog", "comd1.cmd");  
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 “Utilities” in the *User’s Guide*.

exp

Purpose Calculates the exponential function.

Format $y = \exp(x);$

Input x NxK matrix.

Output y NxK matrix containing e , the base of natural logs, raised to the powers given by the elements of x .

Example $x = \text{eye}(3);$
 $y = \exp(x);$

```

           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

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

export

export

Purpose Exports a GAUSS matrix to a spreadsheet, database or ASCII file.

Format `ret = export(x, fname, name1);`

Input

<i>x</i>	MxK matrix, data to export.
<i>fname</i>	string, name of file to export with optional path.
<i>name1</i>	Kx1 vector, column names, or scalar 0 for default column names.

Output

<i>ret</i>	scalar, success flag, 1 if successful, 0 if not.
<i>fname</i>	file in the selected format.

Portability Windows only

Remarks Setting *name1* to 0 results in default column names of Var:1 to Var:K.

Example

```
fname = "c:\\temp\\testdata.xls";
matx = rndn(7,4);
let names = gnp invest pop export;
call export(matx, fname, names);
```

This will export the matrix **matx** to the file **c:\\temp\\testdata.xls** in Excel format.

If the first line had been

```
fname = "c:\\temp\\testdata.dbf";
```

the file would have been exported in dBase format.

Globals `_dxaschdr, _dxftype, _dxmiss, _dxprcn, _dxprint,`
`_dxtxdlim, _dxtype, _dxwidth, _dxwkshd`

See also `exportf, import, importf`

exportf

Purpose Exports a GAUSS data set to a spreadsheet, database, or ASCII file.

Format `ret = exportf(dname, fname, name1);`

Input

dname string, name of GAUSS data set with optional path
fname string, name of file to export with optional path.
name1 Kx1 vector, column names, or 0 for all columns.

Output

ret scalar, success flag, 1 if successful, 0 if not.
fname file in the specified format.

Portability Windows only

Remarks The columns of the data set listed in *name1* will be exported. If *name1* is 0, the entire data set will be exported.

Example

```
dname = "c:\\gauss\\data\\mydata.dat";
fname = "c:\\temp\\testdata2.wk1";
call exportf(dname, fname, 0);
```

This will export the data set **dname** to the file **c:\\temp\\testdata2.wk1** in Lotus format.

Globals `_dxaschdr, _dxftype, _dxmiss, _dxprcn, _dxprint, _dxtxdlim, _dxtype, _dxwidth, _dxwkshdr`

See also `export, import, importf`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

extern (dataloop)**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 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 olddata newdata;
    extern base, exvec;
    make ndvar = exvec[seqa(base,1,rows(x_x))];
    /* execute command literally */
    # base = base + rows(x_x);
enddata;
```

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;`

Remarks See “Procedures and Keywords” in the *User’s Guide*.

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 if the symbol is a matrix, string, or procedure. If the symbol is listed and strongly typed in an active library, no **external** statement is needed.

If a matrix or string appears in an **external** statement it needs to appear once in a **declare** statement. If no declaration is found, an **Undefined symbol** error will result.

Example The general eigenvalue procedure, **eigr**, sets a global variable **_eigerr** if it cannot compute all of the eigenvalues.

```
external matrix _eigerr;

x = rndn(4,4);
xi = inv(x);
xeval = eigr(x);
if _eigerr;
    print "Eigenvalues not computed";
end;
endif;
```

Without the **external** statement, the compiler would assume that **_eigerr** was a procedure and incorrectly compile this program. The file

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

external

containing the **eigr** procedure also contains an external statement that defines **_eigerr** as a matrix, but this would not be encountered until the **if** statement containing the reference to **_eigerr** in the main program file had already been incorrectly compiled.

See also **declare**

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

eye

Purpose Creates an identity matrix.

Format `y = eye(n);`

Input *n* scalar, size of identity matrix to be created.

Output *y* NxN 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);`

```

      1.000000 0.000000 0.000000
x =  0.000000 1.000000 0.000000
      0.000000 0.000000 1.000000

```

See also `zeros`, `ones`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fcheckerr

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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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 is 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, and 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**.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

feq-fne

feq-fne

Purpose Fuzzy comparison functions. These functions use `_fcmtol` to fuzz the comparison operations to allow for roundoff error.

Format

```

y = feq(a, b);
y = fge(a, b);
y = fgt(a, b);
y = fle(a, b);
y = flt(a, b);
y = fne(a, b);

```

Input

a NxK matrix, first matrix.

b LXM matrix, second matrix, EXE compatible with *a*.

`_fcmtol` global 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;
```

The calling program can reset `_fcmtol` before calling these procedures.

```
_fcmtol = 1e-12;
```

Example

```

x = rndu(2, 2);
y = rndu(2, 2);
t = fge(x, y);

```

```

x =    0.0382895    0.07253527
      0.01471395    0.96863611

```

freq-fne

y = 0.25622293 0.70636474
0.00361912 0.35913385

t = 0.0000000

- Source fcompare.src
- Globals _fcmtol
- See also dotfreq-dotfne

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

fflush

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.

Example

```
x = { 22 24 ,
      23 25 };
y = fft(x);

y = 23.500000 -1.000000
    -0.500000 0.00000000
```

See also `ffti`, `rfft`, `rffti`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ffti**ffti**

Purpose Computes an inverse 1- or 2-D Fast Fourier transform.

Format $y = \text{ffti}(x);$

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.

Example

$$x = \begin{Bmatrix} 22 & 24 \\ 23 & 25 \end{Bmatrix};$$
$$y = \text{fft}(x);$$
$$y = \begin{bmatrix} 23.500000 & -1.000000 \\ -0.500000 & 0.000000 \end{bmatrix}$$
$$fi = \text{ffti}(y);$$
$$fi = \begin{bmatrix} 22.000000 & 24.000000 \\ 23.000000 & 25.000000 \end{bmatrix}$$

See also **fft, rfft, rffti**

fftm

Purpose Computes a multi-dimensional FFT.

Format $y = \text{fftm}(x, \text{dim});$

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:

$$\begin{aligned} X_{\text{hyper}} &= X_{\text{cube1}} \mid X_{\text{cube2}} \\ X_{\text{cube1}} &= X_{\text{mat1}} \mid X_{\text{mat2}} \\ X_{\text{mat1}} &= X_{\text{row1}} \mid X_{\text{row2}} \\ X_{\text{row1}} &= X_{\text{col1}} \mid X_{\text{col2}} \end{aligned}$$

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]
```

fftm

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 `fftm_i, fft, ffti, fftn`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ffftmi

Purpose Computes a multi-dimensional inverse FFT.

Format $y = \text{ffftmi}(x, \text{dim});$

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:

$$\begin{aligned}
 X_{\text{hyper}} &= X_{\text{cube1}} \mid X_{\text{cube2}} \\
 X_{\text{cube1}} &= X_{\text{mat1}} \mid X_{\text{mat2}} \\
 X_{\text{mat1}} &= X_{\text{row1}} \mid X_{\text{row2}} \\
 X_{\text{row1}} &= X_{\text{col1}} \mid X_{\text{col2}}
 \end{aligned}$$

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 */
xhfftmi = 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)**.

fftm

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 `fftm`, `fft`, `ffti`, `fftn`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fftn

Purpose Computes a complex 1- or 2-D FFT.

Format $y = \text{fftn}(x);$

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

$$2^p \times 3^q \times 5^r \times 7^s, \quad p, q, r \text{ nonnegative integers} \\ s = 0 \text{ or } 1$$

If a dimension of x does not meet this requirement, it will be padded with zeros to the next allowable size before the 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, $2^6 \times 3 \times 5^2 \times 7$, 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 $1/(L \times M)$.

See also **fft**, **ffti**, **fftm**, **fftmf**, **rfft**, **rffti**, **rfftip**, **rfftn**, **rfftnp**, **rfftp**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fgets

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.

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, $N \leq \text{numl}$.

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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fgetsat

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, $N \leq numl$.

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.

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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fileinfo

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 the filename.

Output *fnames* Nx1 string array of all filenames that match, null string if none are found.

finfo Nx13 matrix, information about matching files.

UNIX

- [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

OS/2, 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

fileinfo

DOS	[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
	[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]	n/a, 0
	[N, 10]	last data modification time
	[N, 11]	n/a, 0
	[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]-[N,11]) are expressed as the number of seconds since midnight, Jan. 1, 1970, Coordinated Universal Time (UTC).

See also `files`, `filesa`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

files

files

This command is obsolete. See “**files**” on page A-7.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

filesa

Purpose	Returns a string array of file names.	
Format	<code>y = filesa(<i>n</i>);</code>	
Input	<i>n</i>	string, file specification to search for. Can include path. Wildcards are allowed in the filename.
Output	<i>y</i>	Nx1 string array of all filenames that match, or null string if none are found.
Remarks	<i>y</i> will contain file <i>names</i> only; any path information that was passed is dropped.	
Example	<pre>y = filesa("ch*");</pre> In this example all files listed in the current directory that begin with “ch” will be returned. <pre>proc exist(filename); retp(not filesa(filename) \$== ""); endp;</pre> This procedure will return 1 if the file exists or 0 if not.	
See also	fileinfo, files, shell	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

floor

floor

Purpose	Rounds down toward $-\infty$.
Format	$y = \text{floor}(x);$
Input	x NxK matrix.
Output	y NxK matrix containing the elements of x rounded down.
Remarks	This rounds every element in the matrix x down to the nearest integer.
Example	$x = 100 * \text{rndn}(2, 2);$ $x = \begin{bmatrix} 77.68 & -14.10 \\ 4.73 & -158.88 \end{bmatrix}$ $f = \text{floor}(x);$ $f = \begin{bmatrix} 77.00 & -15.00 \\ 4.00 & -159.00 \end{bmatrix}$
See also	ceil, round, trunc

fmod

Purpose Computes the floating-point remainder of x/y .

Format $r = \text{fmod}(x, y);$

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” in the *User’s Guide*.)

Example

```
x = seqa(1.7, 2.3, 5)';
y = 2;
r = fmod(x, y);

x =  1.7  4  6.3  8.6 10.9
r =  1.7  0  0.3  0.6 0.9
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fn

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);  
a = 50.265482
```

fonts

Purpose	Loads fonts to be used in the graph.		
Library	pgraph		
Format	fonts (<i>str</i>);		
Input	<i>str</i>	string or character vector containing the names of fonts to be used in the plot.	
	Simplex	standard sans serif font.	
	Simgrma	Simplex Greek, math.	
	Microb	bold and boxy.	
	Complex	standard font with serif.	
	The first font specified will be used for the axes numbers.		
	If <i>str</i> is a null string, or fonts is not called, Simplex is loaded by default.		
Remarks	For information on how to select fonts within a text string, see “Publication Quality Graphics in the <i>User’s Guide</i> .”		
Source	pgraph.src		
See also	title, xlabel, ylabel, zlabel		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

font

font

Purpose Set the font for the active window.

Format *oldfonth* = **font** (*fonthandle*);

Input *fonthandle* scalar, font handle or -1.

Output *oldfonth* scalar, handle of previous font.

Portability UNIX only

Remarks **font** sets the font for the active window to *fonthandle* and returns the handle of the window's previous font in *oldfonth*. If *fonthandle* = -1, **font** returns the current font handle without changing it.
font is not supported for PQG windows.

FontLoad

Purpose	Load a font.	
Format	<i>fonthandle</i> = FontLoad (<i>fontspec</i>) ;	
Input	<i>fontspec</i>	string, font specification.
Output	<i>fonthandle</i>	scalar, font handle.
Portability	UNIX only	
Remarks	FontLoad loads the X11 font specified in <i>fontspec</i>, where <i>fontspec</i> is a standard X11 font name, and may contain the wildcard characters “*” and “?”. If the font is successfully loaded, FontLoad returns a GAUSS font handle which is used in other routines (e.g., font, WinGetAttributes) to refer to the font. If the font requested cannot be loaded, FontLoad returns a -1. Font handles 1 through 4 are reserved by GAUSS; font handles returned by FontLoad are ≥ 5.	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

FontUnload

FontUnload

Purpose Unload a font.

Format *ret* = **FontUnload**(*fonthandle*);

Input *fonthandle* scalar, font handle.

Output *ret* scalar, 0 if call is successful, -1 if it fails.

Portability **UNIX only**

Remarks **FontUnload** unloads the font specified by *fonthandle*, freeing the memory associated with using it. If the font is still in use by a window, **FontUnload** will run successfully, but the font will be maintained internally until the window relinquishes it. Then it will be unloaded.

When a font is unloaded, you should set its handle to zero, to let your program know that it no longer refers to a valid font. The best way to do this is:

`fonthandle = FontUnload(fonthandle);`

Font handles 1 through 4 are reserved by GAUSS, and cannot be unloaded.

FontUnloadAll

Purpose Unload all currently loaded fonts.

Format `FontUnloadAll;`

Portability UNIX only

Remarks **FontUnloadAll** unloads all the currently loaded fonts, freeing the memory associated with using them. If any of the fonts are still in use by any windows, **FontUnloadAll** will run successfully, but the fonts in use will be maintained internally until relinquished. Then they will be unloaded.

When fonts are unloaded, you should set their handles to zero, to let your program know that they no longer refer to valid fonts. The easiest way to do this is:

```
clear f1, f2[,f3...];
```

Font handles 1 through 4 are reserved by GAUSS, and cannot be unloaded.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fopen

fopen

Purpose Opens a file.

Format *f* = **fopen**(*filename*, *omode*);

Input

<i>filename</i>	string, name of file to open.
<i>omode</i>	string, file I/O mode. (See Remarks, below.)

Output

<i>f</i>	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 be 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

fopen

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**.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

for**for**

Purpose Begins a **for** loop.

Format **for** *i* (*start*, *stop*, *step*);

·
·
·

endfor;

Input

<i>i</i>	literal, the name of the counter variable.
<i>start</i>	scalar expression, the initial value of the counter.
<i>stop</i>	scalar expression, the final value of the counter.
<i>step</i>	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. They are converted to integers and stored local to the loop.

The **for** loop is optimized for speed and is 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;
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

format

format

Purpose Controls the format of matrices and numbers printed out with **print** or **lprint** statements.

Format **format** *[/typ]* *[/fnted]* *[/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 (**/mat**), string arrays (**/sa**), and 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, format assumes **/mat**.

/fnted literal, enable formatting flag.

/on, /off Enable/disable formatting. When formatting is disabled, the contents of a variable are dumped to the window 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.

format

/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.
<u>Right-Justified</u>	
/rd	Signed decimal number in the form <code>[[-]] #####.#####</code> , 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 <code>[[-]] #####.#####</code> , 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 <code>[[-]] #E±###</code> with no decimal point printed.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

format

/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 suppressed 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 $[[-]] \text{#####}$, 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 $[[-]] \text{#.##E}\pm\text{###}$, 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 $[[-]] \text{#E}\pm\text{###}$ with no decimal point printed. If the number is positive, a space character will replace the leading minus sign.

format

/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 suppressed 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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

format**Portability** **UNIX**

format sets the **print** statement format for the active graphic panel. Each graphic panel “remembers” its own format, even when it is no longer the active graphic panel.

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 $p \geq 8$ 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:

```

-1.63533465    1.61350700    -1.06295179
 0.26171282    0.27972294    -1.38937242
 0.58891114    0.46812202     1.08805960
```

This code:

```
format /m1 /rzs 1,10;
```

```
print x;
```

produces:

```

-1.6353346    1.613507    -1.0629518
 0.26171282   0.27972294   -1.3893724
 0.58891114   0.46812202    1.0880596
```

This code:

```
format /m3 /rdn 16,4;
```

```
print x;
```

produces:

```

Row 1
      -1.6353    1.6135    -1.0630
Row 2
      0.2617    0.2797    -1.3894
Row 3
      0.5889    0.4681     1.0881
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

format

This code:

```
format /m1 /ldn 16,4;  
print x;
```

produces:

-1.6353	1.6135	-1.0630
0.2617	0.2797	-1.3894
0.5889	0.4681	1.0881

This code:

```
format /m1 /res 12,4;  
print x;
```

produces:

-1.6353E+000	1.6135E+000	-1.0630E+000
2.6171E-001	2.7972E-001	-1.3894E+000
5.8891E-001	4.6812E-001	1.0881E+000

See also `formatcv`, `formatnv`, `print`, `lprint`, `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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

formatnv

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  
B 2  
C 3
```

Source gauss.src

Globals __fmtnv

See also formatcv, printfm, printfmt

fputs

Purpose	Writes strings to a file.
Format	<i>numl</i> = fputs (<i>f</i> , <i>sa</i>);
Input	<i>f</i> scalar, file handle of a file opened with fopen . <i>sa</i> string or string array.
Output	<i>numl</i> 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 <i>sa</i> , 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 <i>numl</i> is not equal to the number of elements in <i>sa</i> , 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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fputst

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.

fseek

Purpose Positions the file pointer in a file.

Format `ret = fseek(f, offs, base);`

Input

<i>f</i>	scalar, file handle of a file opened with fopen .
<i>offs</i>	scalar, offset (in bytes).
<i>base</i>	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 **fgets_{xx}** or **fputs_{xx}** commands may not produce the expected result. For example:

```
p = ftell(f);
s = fgets(a(f, 7);
call fseek(f, p, 0);
```

is not reliable. The best results are obtained by **fseek**'ing to the beginning of the file and then **fseek**'ing to the desired location, as in

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fseek

```
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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

fstrerror

Purpose	Returns an error message explaining the cause of the most recent file I/O error.
Format	<code>s = fstrerror;</code>
Output	<code>s</code> 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 fclearrerr, 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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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.

ftocv

Purpose	Converts a matrix containing floating point numbers into a matrix containing the decimal character representation of each element.	
Format	$y = \text{ftocv}(x, \text{field}, \text{prec});$	
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 $\text{prec} = 0$, the decimal point will be suppressed.	
Example	$y = \text{seqa}(6, 1, 5);$	
	$x = 0 \ \$+ \text{"cat"} \ \$+ \text{ftocv}(y, 2, 0);$	
	<pre> cat06 cat07 x = cat08 cat09 cat10 </pre>	
	Notice that the (0 \$+) above was necessary to force the type of the result to matrix because the string constant cat would be of type string. The left operand in an expression containing a \$+ operator controls the type of the result.	
See also	ftos	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ftos

ftos

Purpose Converts a scalar into a string containing the decimal character representation of that number.

Format $y = \text{ftos}(x, \text{fmat}, \text{field}, \text{prec});$

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;
```

ftos

```
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+02"
s1 = " 929.86 seconds"
s2 = "   5.46 seconds"
om = "The maximum resistance is 929.86 ohms."
```

ftos

See also `ftocv`, `stof`, `format`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

gamma

gamma

Purpose Returns the value of the gamma function.

Format $y = \text{gamma}(x);$

Input x NxK matrix.

Output y NxK matrix.

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**. **lnfact** can be used to compute log gamma.

Example $y = \text{gamma}(2.5);$

$y = 1.32934$

See also **cdfchic**, **cdfbeta**, **cdfffc**, **cdfn**, **cdfnc**, **cdftc**, **erf**, **erfc**

gammaii

Purpose	Computes the inverse incomplete gamma function.		
Format	$x = \text{gammaii}(a, p);$		
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	<code>_ginvinc</code> , <code>__macheps</code>		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

gausset

gausset

Purpose	Resets the global control variables declared in <code>gauss .dec</code> .
Format	<code>gausset;</code>
Source	<code>gauss.src</code>
Globals	<code>__altnam, __con, __ff, __fmtcv, __fmtnv,</code> <code>__header, __miss, __output, __row, __rowfac,</code> <code>__sort, __title, __tol, __vpad, __vtype, __weight</code>

getf

Purpose Loads an ASCII or binary file into a string.

Format `y = getf(filename, mode);`

Input

<i>filename</i>	string, any valid file name.
<i>mode</i>	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, up to 65519 bytes long.

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 Create a file `examp.e` containing the following program.

```
library pgraph;
graphset;
x = seqa(0, 0.1, 100);
y = sin(x);
xy(x, y);
```

Then execute the following.

```
y = getf("examp.e", 0);
```

```
print y;
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

getf

This produces:

```
library pgraph;  
graphset;  
x = seqa(0,0.1,100);  
y = sin(x);  
xy(x,y);
```

See also **load, save, let, con**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

getname

Purpose	Returns a column vector containing the names of the variables in a GAUSS data set.
Format	<code>y = getname(<i>dset</i>);</code>
Input	<i>dset</i> string specifying the name of the data set from which the function will obtain the variable names.
Output	<i>y</i> Nx1 vector containing the names of all of the variables in the specified data set.
Remarks	The output, <i>y</i> , will have as many rows as there are variables in the data set.
Example	<pre> y = getname("olsdat"); format 8,8; print \$y; produces: TIME DIST TEMP FRICT The above example assumes the data set olsdat contained the variables <i>TIME</i>, <i>DIST</i>, <i>TEMP</i>, <i>FRICT</i>. Note that the extension is not included in the filename passed to the getname function. </pre>
See also	<code>getnamef</code> , <code>indcv</code>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

getnamef

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 f = olsdat for read;
y = getnamef(f);
t = vartypef(f);
print y;
```

produces:

time

dist

temp

frict

The above example assumes the data set **olsdat** contained the variables *time*, *dist*, *temp*, *frict*.

Note the use of **vartypef** to determine the types of these variables.

See also `getname`, `indcv`, `vartypef`

getnr

Purpose	Computes number of rows to read per iteration for a program that reads data from a disk file in a loop.	
Format	<i>nr</i> = getnr (<i>nsets</i> , <i>ncols</i>);	
Input	<i>nsets</i>	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.
	<i>ncols</i>	scalar, columns in the data file.
Output	<i>nr</i>	scalar, number of rows readr should read per iteration of the read loop.
Remarks	If __row is greater than 0, <i>nr</i> 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	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

getpath

getpath

Purpose	Returns an expanded filename including the drive and path.
Format	<i>fname</i> = getpath (<i>pfname</i>);
Input	<i>pfname</i> string, partial filename with only partial or missing path information.
Output	<i>fname</i> string, filename with full drive and path.
Remarks	This function handles relative path references.
Example	<pre>y = getpath("temp.e"); print y; produces: /gauss/temp.e</pre>
Source	getpath.src

getwind

Purpose	Retrieves the current graphic panel number.
Library	pgraph
Format	<i>n</i> = getwind ;
Output	<i>n</i> 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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

gosub

gosub

Purpose Causes a branch to a subroutine.

Format gosub label;

.
. .
. .

label:

.
. .
. .

return;

Remarks For multi-line recursive user-defined functions, see “Procedures and Keywords” in the *User’s Guide*.

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 and 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.

gosub

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” in the *User’s Guide*.)

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:
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

gosub

```

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 previous example, when the **gosub** statement is executed, the following sequence of events results:

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 popped 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 popped per **pop** statement.

See also **goto, proc, pop, return**

goto

Purpose Causes a branch to a label.

Format **goto** *label*;

·
·
·

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:

1.00000000 2.00000000 3.00000000

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

goto

See also **gosub, if**

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

gradp

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.		
Format	$g = \text{gradp}(\&f, x0);$		
Input	$\&f$	a pointer to a vector-valued function ($f: K \times 1 \rightarrow N \times 1$) 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 $N \times 1$ vector: <pre>proc f(x); retp(exp(x.*b)); endp;</pre>	
	$x0$	$K \times 1$ vector of points at which to compute gradient.	
Output	g	$N \times K$ 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 $1 \times K$ 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	<pre>proc myfunc(x); retp(x.*2 .* exp(x.*x./3)); endp; x0 = 2.5 3.0 3.5; y = gradp(&myfunc,x0); y = 82.98901842 0.00000000 0.00000000 0.00000000 281.19752975 0.00000000 0.00000000 0.00000000 1087.95414117</pre>		

gradp

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`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

graph

Purpose Graphs a set of points, taking the horizontal coordinates of the points from one matrix and the vertical coordinates from the other.

Format `graph x,y;`

Input

<i>x</i>	NxK matrix of horizontal coordinates.
<i>y</i>	LxM matrix, vertical coordinates, ExE conformable with <i>x</i> .

Portability DOS only

Remarks This command matches up the *x* and *y* values in the standard element-by-element fashion and sets the appropriate pixels.

The origin 0,0 is at the lower left-hand corner of the window.

Note that the window must be in graphics mode for this command to operate.

Example

```
x = seqa(0,1,640);
y = 50~100~150;
call setvmode(17);
graph x,y;
wait;
```

The program above will draw three horizontal lines across the window, one at *y* = 50, one at *y* = 100, and one at *y* = 150.

When GAUSS returns to interactive mode, the window will be reset automatically to text mode because the editor requires the window be in text mode. Your program should contain a pause to allow viewing the graph or printing with the DOS graphics window dump.

See also `color`, `plot`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

graphprt

graphprt

Purpose Controls automatic printer hardcopy and conversion file output.

Library pgraph

Format graphprt (*str*) ;

Input *str* string, control string.

Portability UNIX
Not supported, use WinPrintPQG instead.

Windows
-ps=q, -pm, -pr=1, -cs=q, -cm, -w options not supported
OS/2
-pf, -ps, -pr, -cs, -w flags 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.
- PF=*name* set output print file name.
- PO=*c* set print orientation.
 - L landscape.
 - P portrait.
- PS=*c* set print page size.
 - Q quarter page.
 - H half page.
 - F full page.
- PM=*l,r,t,b* set print margins to *left,right,top,bottom* in inch units.
- PR=*n* set print resolution.
 - 1 low.
 - 2 medium.
 - 3 high.
- C=*n* convert to another file format.

graphprt

	1	Encapsulated PostScript file.
	2	Lotus .PIC file.
	3	HPGL Plotter file.
	4	PCX bitmap format. (DOS, OS/2 only)
	5	BMP (Windows only)
	6	CGM (Windows only)
	7	GEM (Windows only)
	8	WMF (Windows only)
	9	SCODL (Windows only)
	10	TIFF (Windows only)
-CF=name		set converted output file name.
-CO=n		set conversion file orientation.
	L	landscape.
	P	portrait.
-CS=c		set conversion file page size.
	Q	quarter page.
	H	half page.
	F	full page.
-CM=l,r,t,b		set conversion file margins to <i>left,right,top,bottom</i> in inch units.
-W=n		display graph, wait <i>n</i> 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 graphics configuration file. Any parameters passed through **graphprt** will override the default values. (See “Publication Quality Graphics” in the *User’s Guide*.)

Under DOS, this uses a utility called **pqgrun**.

Example Automatic print using a single graphics call.

```
library pgraph;
graphset;
load x,y;
```

graphprt

```
graphprt("-p"); /* tell "xy" to print */  
xy(x,y);        /* create graph and print */
```

Automatic print using multiple graphics graphic panels. Note **graphprt** is called once just before the **endwind** call.

```
library pgraph;  
graphset;  
load x,y;  
begwind;  
window(1,2); /* 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;  
prtnam = "myprint.prn"; /* name of output print  
                          /* file */  
prtres = "1";           /* low-res print */  
prtmarg= ".5,.5,.5,.5" /* page margins */
```

graphprt

```
/* concatenate options into one string */  
cmdstr = "-p" $+ " -pf=" $+ prtnam $+ " -pr=" $+ prtres;  
cmdstr = cmdstr $+ " -pm=" $+ prtmarg;  
  
graphprt(cmdstr); /* tell "xy" to print */  
xy(x,y);          /* create graph and print */
```

The above string *cmdstr* will look like this:

“-p -pf=myprint.prn -pr=1 -pm=.5,.5,.5,.5”

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

graphprt

Source pgraph.src

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

graphset

Purpose	Resets graphics globals to default values.
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 <code>pgraph.dec</code> and to the procedure graphset.
Source	<code>pgraph.src</code>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

hasimag

hasimag

Purpose Tests whether the imaginary part of a complex matrix is negligible.

Format `y = hasimag(x);`

Input *x* N x K matrix.

Output *y* scalar, 1 if the imaginary part of *x* has any nonzero elements, 0 if it consists entirely of 0's.

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 };
y = hasimag(x);
y = 1.0000000
```

See also **iscplx**

header

Purpose Prints a header for a report.

Format `header(prcnm, dataset, ver) ;`

Input

<i>prcnm</i>	string, name of procedure that calls header .
<i>dataset</i>	string, name of data set.
<i>ver</i>	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 the 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 printed
f	file name being analyzed is printed
__title	string, title for header.

Source `gauss.src`

Globals `__header, __title`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

hess

hess

Purpose Computes the Hessenberg form of a square matrix.

Format $\{ h, z \} = \text{hess}(x);$

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, No. Holland, Amsterdam, 1982, 102-11).

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 similarity 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 $\text{let } x[3,3] = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9; \end{matrix}$
 $\{ h, z \} = \text{hess}(x);$

$h = \begin{matrix} 1.00000000 & -3.59700730 & -0.24806947 \\ -8.06225775 & 14.04615385 & 2.83076923 \\ 0.00000000 & 0.83076923 & -0.04615385 \end{matrix}$

hess

```

      1.000000000  0.000000000  0.000000000
z =  0.000000000 -0.49613894 -0.86824314
      0.000000000 -0.86824314  0.49613894

```

See also `schur`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

hessp

hessp

Purpose Computes the matrix of second partial derivatives (Hessian matrix) of a function defined as a procedure.

Format $h = \text{hessp}(\&f, x0);$

Input $\&f$ pointer to a single-valued function $f(x)$, defined as a procedure, taking a single $K \times 1$ vector argument ($f: K \times 1 \rightarrow 1 \times 1$); $f(x)$ may be defined in terms of global arguments in addition to x .
 $x0$ $K \times 1$ vector specifying the point at which the Hessian of $f(x)$ is to be computed.

Output h $K \times K$ 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;
```

$b0 = \{ 3, 2, 1 \};$
 $h = \text{hessp}(\&g, b0);$

hessp

The resulting matrix of second partial derivatives of **g(b)** evaluated at **b=b0** is:

22027.12898372	44054.87238165	66083.36762901
44054.87238165	88111.11102645	132168.66742899
66083.36762901	132168.66742899	198256.04087836

Source hessp.src

See also gradp

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

hist

hist

Purpose Computes and graphs a frequency histogram for a vector. The actual frequencies are plotted for each category.

Library pgraph

Format { *b*, *m*, *freq* } = **hist**(*x*, *v*);

Input

<i>x</i>	Mx1 vector of data.
<i>v</i>	Nx1 vector, the breakpoints to be used to compute the frequencies,
	or
	scalar, the number of categories.

Output

<i>b</i>	Px1 vector, the breakpoints used for each category.
<i>m</i>	Px1 vector, the midpoints of each category.
<i>freq</i>	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:

$$\begin{array}{rclclcl} freq[1] & = & & x & \leq & b[1] \\ freq[2] & = & b[1] & < & x & \leq & b[2] \\ freq[3] & = & b[2] & < & x & \leq & b[3] \\ \\ freq[P] & = & b[P-1] & < & x & \leq & b[P] \end{array}$$

hist

Example `library pgraph;`
 `x = rndn(5000,1);`
 `{ b,m,f } = hist(x,20);`

Source `phist.src`

See also `histp, histf, bar`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

histf

histf

Purpose Graphs a histogram given a vector of frequency counts.

Library **pgraph**

Format **histf**(*f*, *c*);

Input

<i>f</i>	Nx1 vector, frequencies to be graphed.
<i>c</i>	Nx1 vector, numeric labels for categories. If this is a scalar 0, a sequence from 1 to rows (<i>f</i>) 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 $\{ b, m, freq \} = \text{histp}(x, v);$

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:

$$\begin{array}{rclclcl} freq[1] & = & & x & \leq & b[1] \\ freq[2] & = & b[1] & < & x & \leq & b[2] \\ freq[3] & = & b[2] & < & x & \leq & b[3] \end{array}$$

$$freq[P] = b[P-1] < x \leq b[P]$$

Source phist.src

See also hist, histf, bar

hsec

hsec

Purpose Returns the number of hundredths of a second since midnight.

Format `y = hsec;`

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(100,100);`

`ts = hsec;`

`y = x*x;`

`et = hsec-ts;`

In this example, **hsec** is used to time a 100x100 multiplication in GAUSS. A 100x100 matrix, **x**, is created, and the current time, in hundredths of a second since midnight, is stored in the variable **ts**. Then the multiplication is carried out. Finally, **ts** is subtracted from **hsec** to give the time difference which is assigned to **et**.

See also `date, time, timestr, ethsec, etstr`

if

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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

if

Example

```
if x < 0;  
    y = -1;  
elseif x > 0;  
    y = 1;  
else;  
    y = 0;  
endif;
```

See also **do**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

imag

Purpose Returns the imaginary part of a matrix.

Format $zi = \text{imag}(x);$

Input x NxK matrix.

Output zi NxK matrix, the imaginary part of x .

Remarks If x is real, zi will be an NxK matrix of zeros.

Example

```
x = { 4i 9 3,
      2 5-6i 7i };
y = imag(x);

y =  4.00000000  0.00000000  0.00000000
     0.00000000 -6.00000000  7.00000000
```

See also `complex`, `real`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

import

import

Purpose Imports a spreadsheet, database, or ASCII file into a matrix.

Format { *x*, *namel* } = **import**(*fname*, *range*, *sheet*);

Input

fname string, file name with path.

range string, range of cells (spreadsheet only), field descriptor (ASCII files), or 0.

sheet scalar, sheet number of multi-sheet spreadsheet or 0.

Output

x NxK matrix, imported data.

namel Kx1 vector, column headers.

Portability Windows only

Remarks

If *sheet* is 0, **import** will import the first (possibly only) sheet of a spreadsheet.

For spreadsheets, *range* indicates the columns to be imported, and can be expressed in the form B7:K243 or B7 . . K243. A *range* of 0 imports all data.

For packed ASCII files, *range* is a descriptor that defines field name, type, width, and optionally precision. It is a single string of the form

name [[*type*]] *fldspec* [[*name* [[*type*]] *fldspec* . . .]]

where

name is the column name for the field.

type is \$ for a character field, blank for a numeric field.

fldspec is either a column range (e.g., 5-8), a start column and field width (e.g., 5,4 or 5,4.2), or a field width only (e.g., 4 or 4.2). If only a field width is given, the start column is input from the previous field. If the field width is specified with a decimal (e.g., 4.2), then a decimal point will be inserted that many places in from the right edge of the field.

A shorthand for a set of columns is available (e.g., x01-x09); the subsequent *type* and *fldspec* are applied to all columns in the set.

import

Example `fname = "c:\\temp\\testdata.xls";`
 `let names = gnp invest pop export;`
 `{mmx, names} = import(fname, "A3:D12", 3);`

This will import the range of cells **A3** to **D12** of the third sheet of `testdata.xls` into the GAUSS matrix **mmx** with the column headings defined by **names**. Substituting a **0** or **1** for the **3** would have imported the first (possibly only) sheet.

```
fname = "c:\\temp\\tstdta.asc";
range = "var1 $ 6,2 var2 8-15 var3 18,4.1 gnp 7
cons 6.2 xs1-xs20 2";
{x,names} = import(fname,range,0);
```

Here, a GAUSS matrix **x** is created from a packed ASCII data set, using the descriptor shown.

Globals `_dxaschdr, _dxftype, _dxmiss, _dxprcn, _dxprint,`
 `_dxtxdlim, _dxtype, _dxwidth, _dxwkshdr`

See also `export, exportf, importf`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

importf

importf

Purpose Imports a spreadsheet, database, or ASCII file to a GAUSS data set.

Format *ret* = **importf**(*fname*, *dset*, *range*, *sheet*);

Input

fname string, file name with optional path.

dset string, name of data set with optional path.

range string, range of cells (spreadsheet only), field descriptor (ASCII files), or 0.

sheet scalar, sheet number of multi-sheet spreadsheet or 0.

Output

ret scalar, success flag, 1 if successful, 0 if not.

dset GAUSS data set.

Portability Windows only

Remarks See **import** for details on the *range* and *sheet* parameters.

Example

```
fname = "c:\\temp\\testdata2.dbf";
dname = "c:\\gauss\\data\\mydata.dat";
call importf(dname, fname, 0, 0);
```

This will create a GAUSS data set `c:\gauss\data\mydata.dat` from the data in the dBase III file `c:\temp\testdata2.dbf`.

Globals `_dxaschdr`, `_dxftype`, `_dxmiss`, `_dxprcn`, `_dxprint`,
`_dxtxdlim`, `_dxtype`, `_dxwidth`, `_dxwkshdr`

See also `export`, `exportf`, `import`

#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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

#include

Example `#include "/gauss/inc/cond.inc";`

The command will cause the code in the file `cond.inc` to be merged into the current program at the point at which this statement appears.

See also `run, #lineson`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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 what = AGE PAY SEX;
let where = AGE SEX JOB "date" PAY;
z = indcv(what,where);
```

```

      AGE
what =  PAY
      SEX
```

```

      AGE
      SEX
where =  JOB
        date
        PAY
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

indcv

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

$$Z = \begin{matrix} 1 \\ 5 \\ 2 \end{matrix}$$

indexcat

Purpose Returns the indices of the elements of a vector which fall into a specified category.

Format `y = indexcat(x, v);`

Input

x Nx1 vector.

v scalar or 2x1 vector.

If scalar, the function returns the indices of all elements of *x* equal to *v*.

If 2x1, then the function returns the indices of all elements of *x* that fall into the range:

$$v[1] < x \leq v[2].$$

If *v* 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 *v*. 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

```
let x = 1.0 4.0 3.3 4.2 6.0 5.7 8.1 5.5;
let v = 4 6;
y = indexcat(x, v);
```

```

      1.0
      4.0
      3.3
x =    4.2
      6.0
      5.7
      8.1
      5.5
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

indexcat

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

$$v = \begin{matrix} 4 \\ 6 \end{matrix}$$
$$y = \begin{matrix} 4 \\ 5 \\ 6 \\ 8 \end{matrix}$$

indices

Purpose	Processes a set of variable names or indices and returns a vector of variable names and a vector of indices.		
Format	{ <i>name</i> , <i>indx</i> } = indices (<i>dataset</i> , <i>vars</i>);		
Input	<i>dataset</i>	string, the name of the data set.	
	<i>vars</i>	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	<i>name</i>	Nx1 character vector, the names associated with <i>vars</i> .	
	<i>indx</i>	Nx1 numeric vector, the column indices associated with <i>vars</i> .	
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		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

indices2

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

<i>dataset</i>	string, the name of the data set.
<i>var1</i>	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.
<i>var2</i>	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 <i>var1</i> will be selected.

Output

<i>name1</i>	scalar character matrix containing the name of the variable associated with <i>var1</i> .
<i>indx1</i>	scalar, the column index of <i>var1</i> .
<i>name2</i>	Nx1 character vector, the names associated with <i>var2</i> .
<i>indx2</i>	Nx1 numeric vector, the column indices of <i>var2</i> .

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 **scalerrr**. 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.

indices2

Source indices2.src

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

indnv

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

```
let what = 8 7 3;
let where = 2 7 8 4 3;
z = indnv(what,where);
```

```

      8
what = 7
      3

      2
      7
where = 8
      4
      3
```

z = 3
2
5

- a
- b
- c
- d
- e
- f
- g
- h
- i**
- j
- k
- l
- m
- n
- o
- p
- q
- r
- s
- t
- u
- v
- w
- x y z

integrat2

integrat2

Purpose Integrates the following double integral, using user-defined functions f , g_1 and g_2 , and scalars a and b :

$$\int_a^b \int_{g_2(x)}^{g_1(x)} f(x, y) dy dx$$

Format `y = integrat2(&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.

_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 **integrat2** 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 **integrat2**.

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

Return a scalar constant, OR

Return a vector of function values. **integrat2** 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 **/**.

intgrat2

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

integrat3

integrat3

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 = integrat3(&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) .

`xl` 2x1 or 2xN matrix, the limits of x . These must be scalar limits.

`gl` 2x1 or 2xN matrix of function pointers. These procedures are functions of x .

`hl` 2x1 or 2xN matrix of function pointers. These procedures are functions of x and y .

For `xl`, `gl`, and `hl`, the first row is the upper limit and the second row is the lower limit. N integrations are computed.

`_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 **integrat3** 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 **integrat3**.

Output `y` Nx1 vector of the estimated integral(s) of $f(x, y, z)$ evaluated between the limits given by `xl`, `gl`, and `hl`.

intgrat3

Remarks User-defined functions f , and those used in $g1$ and $h1$, must either:

Return a scalar constant, OR

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 $*$ or $/$.

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;
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

intgrat3

```
xl = 5|-5;  
g0 = &g1|&g2;  
h0 = &h1|&h2;  
_intrec = 0;  
_intord = 40;  
y = intgrat3(&f,xl,g1,h1);
```

This will integrate the function $f(x,y,z) = 2$ 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`

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	$y = \text{intquad1}(\&f, xl);$	
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.
	$\_intord$	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	<pre>proc f(x); retp(x.*sin(x)); endp; xl = 1 0; y = intquad1(&f,xl);</pre> This will integrate the function $f(x) = xsin(x)$ between 0 and 1. Note the use of the $\_*$ instead of $*$.	
Source	integral.src	

intquad1

Globals `_intord, _intq12, _intq16, _intq2, _intq20,`
 `_intq24, _intq3, _intq32, _intq4, _intq40,`
 `_intq6, _intq8`

See also `intsimp, intquad2, intquad3, intgrat2, intgrat3`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

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	$y = \text{intquad2}(\&f, xl, yl);$	
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.
	_intord	global 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	_intrec	global 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 .
	y	Nx1 vector of the estimated integral(s) of $f(x,y)$ evaluated between the limits given by xl and yl .
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 .	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

intquad2

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 **`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	$y = \text{intquad3}(\&f, xl, yl, zl);$	
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.	
	$\_intord$	global 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.
	$\_intrec$	global 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 .	

intquad3

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 **zl** 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 give a single file sorted on that variable.		
Format	intrleav (<i>infile1</i> , <i>infile2</i> , <i>outfile</i> , <i>keyvar</i> , <i>keytyp</i>);		
Input	<i>infile1</i>	string, name of input file 1.	
	<i>infile2</i>	string, name of input file 2.	
	<i>outfile</i>	string, name of output file.	
	<i>keyvar</i>	string, name of key variable, this is the column the files are sorted on.	
	<i>keytyp</i>	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, i.e., 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.		
Example	<pre>intrleav("freq.dat", "freqata.dat", "intfile", "AGE", 1);</pre>		
Source	sortd.src		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

intrsect

intrsect

Purpose Returns the intersection of two vectors, with duplicates removed.

Format `y = intrsect(v1,v2,flag);`

Input

<i>v1</i>	Nx1 vector.
<i>v2</i>	Mx1 vector.
<i>flag</i>	scalar; if 1, <i>v1</i> , and <i>v2</i> are numeric, if 0, character.

Output

<i>y</i>	Lx1 vector containing all unique values that are in both <i>v1</i> and <i>v2</i> , 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 **unique** before calling **intrsect**.

Source intrsect.src

Example

```
let v1 = mary jane linda dawn;
let v2 = mary sally lisa ruth linda;
y = intrsect(v1,v2,0);
```

```
y =  LINDA
     MARY
```

intsimp

Purpose	Integrates a specified function using Simpson's method with end correction. A single integral is computed in one function call.	
Format	<code>y = intsimp(&f, xl, tol);</code>	
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	<pre>proc f(x); retp(sin(x)); endp;</pre>	
	<pre>let xl = { 1, 0 };</pre>	
	<pre>y = intsimp(&f,xl,1E-8);</pre>	
	<pre>y = 0.45969769</pre>	
	This will integrate the function between 0 and 1.	
Source	intsimp.src	
See also	intquad1, intquad2, intquad3, intgrat2, intgrat3	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

inv, invpd

inv, invpd

Purpose	inv returns the inverse of an invertible matrix. invpd returns the inverse of a symmetric, positive definite matrix.	
Format	$y = \text{inv}(x);$ $y = \text{invpd}(x);$	
Input	x	NxN matrix.
Output	y	NxN matrix containing the inverse of x .
Remarks	x can be any legitimate matrix expression that returns a matrix that is legal for the function.	
	For inv , x must be square and invertible.	
	For invpd , 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:	
	trap 1 , return error code	
	inv	invpd
	50	20
	trap 0 , terminate with 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;

bols = 1.017201  0.484244
```

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 1's (to create a constant term). The true coefficients are specified, and the dependent variable **y** is created. Then the **ols** coefficient estimates are computed.

See also `scalerr`, `trap`, `prcsn`

Technical Notes

For complex matrices, **inv** uses the ZGECO, ZGEDI path in the LINPACK routines. For real matrices, it uses the **croutp** function.

The **inv** function uses the Crout decomposition. The advantage of this routine is that on some platforms it allows most of the intermediate results to be computed in extended precision.

The **invpd** function uses the Cholesky decomposition and is based upon the LINPACK routines for positive definite matrices. On OS/2 and DOS, if **prcsn** 80 is in effect, all intermediate calculations and intermediate results will be in the 80-bit extended precision of the 80x87 temporary real format. The final results will be rounded to 64-bit double precision.

The tolerance used to determine singularity is 1.0e-14. This can be changed. See “Singularity Tolerance” in the *User's Guide*.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

invswp

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 four Moore-Penrose conditions:

- 1. $xyx = x$
- 2. $yxy = y$
- 3. xy is symmetric
- 4. yx is symmetric

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 “Appendix C” in the *User’s Guide*.

Example `let x[3,3] = 1 2 3 4 5 6 7 8 9;`
`y = invswp(x);`

`y =`

-1.66666667	0.66666667	0.00000000
1.33333333	-0.33333333	0.00000000
0.00000000	0.00000000	0.00000000

iscplx

Purpose Returns whether a matrix is complex or real.

Format $y = \text{iscplx}(x);$

Input x NxK matrix.

Output y scalar, 1 if x is complex, 0 if it is real.

Example $x = \{ 1, 2i, 3 \};$
 $y = \text{iscplx}(x);$

$y = 1$

See also `hasimag`, `iscplx`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

iscplxf

iscplxf

Purpose Returns whether a data set is complex or real.

Format $y = \text{iscplxf}(fh);$

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**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ismiss

Purpose Returns a 1 if its matrix argument contains any missing values, otherwise returns a 0.

Format $y = \text{ismiss}(x);$

Input x NxK matrix.

Output y scalar, 1 or 0.

Remarks y will be a scalar 1 if the matrix x contains any missing values, otherwise it will be 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, for $x = 1 + .i$, **ismiss**(x) will return a 0.

Example $x = \{ 1 \ 6 \ 3 \ 4 \};$
 $y = \text{ismiss}(x);$

$y = 0$

See also **scalmiss**, **miss**, **missrv**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

isSparse

isSparse

Purpose	Tests whether a matrix is a sparse matrix.
Format	$r = \text{isSparse}(x);$
Input	x MxN sparse or dense matrix.
Output	r scalar, 1 if x is sparse, 0 otherwise.
Source	sparse.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

keep (dataloop)

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

key

key

Purpose Returns the ASCII value of the next key available in the keyboard buffer.

Format `y = key;`

Portability UNIX
key gets input from the active graphic panel. If you are working in terminal mode, **key** does not “see” any keystrokes until ENTER is pressed.

Remarks 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.

Here are the values returned if the key pressed is not a standard ASCII character in the range of 1-255.

1015	SHIFT+TAB
1016-1025	ALT+Q, W, E, R, T, Y, U, I, O, P
1030-1038	ALT+A, S, D, F, G, H, J, K, L
1044-1050	ALT+Z, X, C, V, B, N, M
1059-1068	F1-F10
1071	HOME
1072	CURSOR UP
1073	PAGE UP
1075	LEFT ARROW
1077	RIGHT ARROW
1079	END
1080	DOWN ARROW
1081	PAGE DOWN
1082	INSERT
1083	DELETE
1084-1093	SHIFT+F1-F10
1094-1103	CTRL+F1-F10
1104-1113	ALT+F1-F10
1114	CTRL+PRINT SCREEN

1115	CTRL+LEFT ARROW
1116	CTRL+RIGHT ARROW
1117	CTRL+END
1118	CTRL+PAGE DOWN
1119	CTRL+HOME
1120-1131	ALT+1,2,3,4,5,6,7,8,9,0,HYPHEN, EQUAL SIGN
1132	CTRL+PAGE UP

Example

```
format /rds 1,0;
kk = 0;
do until kk == 27;
    kk = key;
    if kk == 0;
        continue;
    elseif kk == vals(" ");
        print "space \\" kk;
    elseif kk == vals("\r");
        print "carriage return \\" kk;
    elseif kk ≥ vals("0") and kk ≤ vals("9");
        print "digit \\" kk chrs(kk);
    elseif vals(upper(chrs(kk))) ≥ vals("A") and
        vals(upper(chrs(kk))) ≤ vals("Z");
        print "alpha \\" 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 ESC key (ASCII 27) is pressed.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

key

See also vals, chrs, upper, lower, con, cons

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

keyw

Purpose	Waits for and gets a key.
Format	<i>k</i> = keyw ;
Output	<i>k</i> scalar, ASCII value of the key pressed.
Portability	UNIX keyw gets the next key from the input buffer of the active graphic panel. If you are working in terminal mode, GAUSS will not see any input until you press the ENTER key.
Remarks	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 key for a table of return values for extended and function keys.
See also	key

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

keyword

keyword

Purpose Begins the definition of a keyword procedure. Keywords are user-defined functions with local or global variables.

Format **keyword** *name*(*str*);

Input

<i>name</i>	literal, name of the keyword. This name will be a global symbol.
<i>str</i>	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” in *User’s Guide*.

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;
```

keyword

This keyword will respond by printing:

```
Sum is: 15
```

See also `proc`, `local`, `endp`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

lag (data loop)

lag (data loop)

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 data loop, and is then automatically deleted.

lag1

Purpose	Lags a matrix by one time period for time series analysis.
Format	$y = \text{lag1}(x);$
Input	x 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.
Source	lag.src
See also	lagn

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

lag

lag

Purpose Lags a matrix a specified number of time periods for time series analysis.

Format $y = \text{lag}(x, t);$

Input

x	NxK matrix.
t	scalar, number of time periods.

Output y NxK matrix, x lagged t periods.

Remarks If t is positive, **lag** lags x back t time periods, so the first t observations of y are missing. If t is negative, **lag** lags x forward t time periods, so the last t observations of y are missing.

Source lag.src

See also lag1

let

Purpose Creates a matrix from a list of numeric or character values. The result is always of type MATRIX.

Format `let x = constant_list;`

Remarks Expressions and matrix 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 $dE\pm n$, 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. You will need the **let** for statements that you want to protect from the beautifier using the **-l** flag on the beautifier command line.

```
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
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

let

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. For more information on empty matrices, see “Language Fundamentals” in the *User’s Guide*.

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;`

```
x = 0
```

let

```
let x = { 1 2 3, 4 5 6, 7 8 9 };
```

```
      1 2 3  
x =   4 5 6  
      7 8 9
```

```
let x[3,3] = 1 2 3 4 5 6 7 8 9;
```

```
      1 2 3  
x =   4 5 6  
      7 8 9
```

```
let x[3,3] = 1;
```

```
      1 1 1  
x =   1 1 1  
      1 1 1
```

```
let x[3,3];
```

```
      0 0 0  
x =   0 0 0  
      0 0 0
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

let

```
let x = 1 2 3 4 5 6 7 8 9;
```

```
1  
2  
3  
4  
x = 5  
6  
7  
8  
9
```

```
let x = dog cat;
```

```
x = DOG  
CAT
```

```
let x = "dog" "cat";
```

```
x = dog  
cat
```

See also `con`, `cons`, `declare`, `load`

lib

Purpose Builds and updates library files.

Format **lib** *library* [*file*] [*-flag -flag...]*;

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 (default) add paths to entries without paths and expand relative paths.

-gausspath reset all paths using a normal file search.

-leavepath 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.

lib

To control location of temporary files for a complete library build:

- tmp** (default) use the directory pointed to by the **tmp_path** configuration variable. The directory will usually be on a RAM disk. 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.

Remarks 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** `[[ -l ] lib1, lib2, lib3, lib4;`
library;

Remarks If no arguments are given, the list of current libraries will be printed out.

The **-l** option will write a file containing a listing of libraries, files, and symbols for all active libraries. This file will reside in the directory defined by the **lib_path** configuration variable. Under Windows and UNIX, the file will have a unique name beginning with **liblst_**. Under OS/2 and DOS, the file will be called **gausslib.lst**; if it already exists it will be overwritten.

For more information about the library system, see “Libraries” in the *User’s Guide*.

The default extension for library files is **.lcg**.

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 = dog(x);
```

is encountered in a program, **dog** 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 **dog.g**

autodelete **off** return **Undefined symbol** error message

If **dog** calls **cat** and **cat** calls **bird** 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 **dog** in exactly the same sequence as the autoloader. The file containing **dog** will be displayed in the window and you can scroll up and down and look at the code and comments.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

library

Library files are simple ASCII files that you can create with the 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. Blank lines are okay.

Here is a debugging hint. If your program is acting strange and you suspect it is autoloading the wrong copy of a procedure, use the source browser or help facility to locate the suspected function. It will use the same search path that the autoloader uses.

See also `declare`, `external`, `lib`, `proc`

line

Purpose Draws lines in the graphics window.

Format `line x[, y];`

Input

x NxK matrix.

y LxM matrix, ExE conformable with *x*.

Portability DOS only

Remarks *x* and *y* must each have at least 2 elements or no lines will be drawn. The first line starts at *x*[1,1], *y*[1,1]. *P*-1 lines will be drawn where $P = \max(N,L) * \max(K,M)$. In other words, the elements of *x* and *y* will be combined in the same way as with other element-by-element operators.

If there is only one argument,

```
line x;
```

it must be an Nx4 or Nx5 matrix. N lines will be drawn. Each row describes a line. Each column describes:

[N, 1]	x coordinate of beginning of line.
[N, 2]	y coordinate of beginning of line.
[N, 3]	x coordinate of end of line.
[N, 4]	y coordinate of end of line.
[N, 5]	optional color of line.

The window must be in graphics mode. All coordinates are in pixels with (0,0) in the lower left.

See also `color`, `graph`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

#lineson, #linesoff

#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**

listwise (dataloop)

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**.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ln

ln

Purpose	Computes the natural log of all elements of a matrix.
Format	$y = \ln(x);$
Input	x NxK matrix.
Output	y NxK matrix 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	$y = \ln(16);$ $y = 2.7725887$

lncdfbvn

Purpose Computes natural log of bivariate Normal cumulative distribution function.

Format $y = \text{lncdfbvn}(x1, x2, r);$

Input $x1$ NxK matrix, abscissae.
 $x2$ LxM matrix, abscissae.
 r PxQ matrix, correlations.

Output y $\max(N,L,P) \times \max(K,M,Q)$ matrix, $\ln Pr(X < x1, X < x2 \mid r)$.

Remarks $x1$, $x2$, and r must be ExE conformable.

Source lncdfn.src

See also cdfbvn, lncdfmvn

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

lncdfbvn2

lncdfbvn2

Purpose Returns log of cdfbvn of a bounded rectangle.

Format `y = lncdfbvn2(h, dh, k, dk, r);`

Input

<i>h</i>	Nx1 vector, upper limits of integration for variable 1.
<i>dh</i>	Nx1 vector, increments for variable 1.
<i>k</i>	Nx1 vector, upper limits of integration for variable 2.
<i>dk</i>	Nx1 vector, increments for variable 2.
<i>r</i>	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 $\text{err} \geq \text{exp}(y)/100$.

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
```

lncdfbvn2

Example 3

```
trap 2,2;
```

```
lncdfbvn2(1, -1e-45, 1, 1e-45, 0.5);
```

```
WARNING: Dubious accuracy from lncdfbvn2:
```

```
0.0000e+000 ± 2.8e-060
```

```
-INF
```

See also **cdfbvn2, cdfbvn2e**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

lncdfmvn

lncdfmvn

Purpose	Computes natural log of multivariate Normal cumulative distribution function.		
Format	$y = \text{lncdfmvn}(x, r);$		
Input	x	KxL matrix, abscissae.	
	r	KxK matrix, correlation matrix.	
Output	y	Lx1 vector, $\ln Pr(X < x \mid r)$.	
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 $y = \text{lncdfn}(x);$

Input x NxK matrix, abscissae.

Output y NxK matrix, $\ln Pr(X < x)$.

Source lncdfn.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

lncdfn2

lncdfn2

Purpose	Computes natural log of interval of Normal cumulative distribution function.		
Format	$y = \text{lncdfn2}(x, r);$		
Input	x	MxN matrix, abscissae.	
	r	KxL matrix, ExE conformable with x , intervals.	
Output	y	max(M,K) x max(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:		
	$ x \leq 1$ and $dx \leq 1$		$\pm 1e-14$
	$1 < x < 37$ and $ dx < 1 x $		$\pm 1e-13$
	$\min(x, x + dx) > -37$ and $y > -690$		$\pm 1e-11$ or better
	A relative error of $\pm 1e-14$ implies that the answer is accurate to better than ± 1 in the 14th digit.		
Example	<pre>print lncdfn2(-10,29); -7.6198530241605269e-24 print lncdfn2(0,1); -1.0748623268620716e+00 print lncdfn2(5,1); -1.5068446096529453e+01</pre>		
Source	lncdfn.src		
See also	cdfn2		

lncdfnc

Purpose	Computes natural log of complement of Normal cumulative distribution function.		
Format	$y = \text{lncdfnc}(x);$		
Input	x	NxK matrix, abscissae.	
Output	y	NxK matrix, $\ln (1 - Pr(X < x))$.	
Source	lncdfn.src		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

lnfact

lnfact

Purpose	Computes the natural log of the factorial function and can be used to compute log gamma.
Format	$y = \text{lnfact}(x);$
Input	x NxK matrix, all elements must be positive.
Output	y NxK matrix containing the natural log of the factorial of each of the elements of 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 $x \geq 1$, 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: $\text{lngamma}(x) = \text{lnfact}(x-1);$
Example	<pre>let x = 100 500 1000; y = lnfact(x); 363.739375560 y = 2611.33045846 5912.12817849</pre>
Source	lnfact.src
See also	gamma

Technical Notes For $x > 1$, Stirling’s formula is used.
For $0 < x \leq 1$, **ln(gamma($x+1$))** is used.

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

lnpdfn

lnpdfn

Purpose Computes standard Normal log-probabilities.

Format `z = lnpdfn(x);`

Input `x` NxK matrix, data.

Output `z` NxK matrix, 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:

$$y = -\ln(\sqrt{2\pi}) - x.^2/2;$$

For multivariate log-probabilities, see **lnpdfmvn**.

Example `x = { .2, -1, 0, 1, 2 };`

`z = lnpdfn(x);`

`z =`
-2.9189385
-1.4189385
-0.91893853
-1.4189385
-2.9189385

lnpdfmvn

Purpose	Computes multivariate Normal log-probabilities.	
Format	$z = \text{lnpdfmvn}(x, s);$	
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 .	

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

load, loadf, loadk, loadm, loadp, loads**load, loadf, loadk, loadm, loadp, loads**

Purpose Loads from a disk file.

Format `load [[path=path]] x, y[[]=filename, z=filename];`

Remarks All the **loadxx** commands use the same syntax — they only differ in the types of symbols you use them with.

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

load, loadf, loadk, loadm, loadp, loads

have the symbol initialized at compile time. When the load executes, the dummy definition will be replaced with the saved definition.

```
proc corrmatrix; endp;

loadp corrmatrix;

y = corrmatrix;

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 reshaped 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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

load, loadf, loadk, loadm, loadp, loads

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 your file contains nine numbers (1 2 3 4 5 6 7 8 9), then the matrix *x* that was created would 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**.

load, loadf, loadk, loadm, loadp, loads

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**, **0y.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, save, let, con, cons, sysstate**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

loadadd

loadadd

Purpose	Loads a small data set.
Format	$y = \text{loadadd}(\text{dataset});$
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 <code>temp.dat</code> will be loaded. To load a matrix file, use an <code>.fmt</code> extension on dataset .
Source	<code>saveLoad.src</code>
Globals	<code>__maxvec</code>

loadwind

Purpose	Loads a previously saved graphic panel configuration.
Library	pgraph
Format	<i>err</i> = loadwind (<i>namestr</i>);
Input	<i>namestr</i> string, name of file to be loaded.
Output	<i>err</i> 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
See also	savewind

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

local

local

Purpose Declares 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 example above will be treated as a procedure whenever it is accessed in the procedure. What is actually passed in is a pointer to a procedure.

See “Procedures and Keywords” in the *User’s Guide*.

See also `proc`

locate

Purpose Positions the cursor in the window.

Format **locate** *m*, *n*;

Portability UNIX

locate locates the cursor in the active graphic panel. The row argument is ignored for graphic panel 1 and TTY graphic panels, making **locate** act like **tab** in those graphic panels.

Windows

Locates the cursor in the program I/O graphic panel.

OS/2

Locates the cursor in the main graphic panel.

Remarks *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 = csrln;
c = csrcl;
cls;
locate r,c;
```

In this example the window is cleared without affecting the cursor position.

See also **csrln**, **csrcl**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

loess

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, 829-36.

Source loess.src

log

Purpose Computes the $\log_{10}$ of all elements of a matrix.

Format $y = \log(x);$

Input x NxK matrix.

Output y NxK matrix containing the log 10 values of the elements of x .

Remarks **log** 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, **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 $x = \text{round}(\text{rndu}(3,3)*10+1);$
 $y = \log(x);$

```

      4.00000000000  2.00000000000  1.00000000000
x =  10.00000000000  4.00000000000  8.00000000000
      7.00000000000  2.00000000000  6.00000000000

```

```

      0.60205999    0.30103    0.30103
y =  1.00000000000  0.60205999  0.90308999
      0.8450980400    0.30103    0.77815125

```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

loglog

loglog

Purpose	Graphs X vs. Y using log coordinates.	
Library	pgraph	
Format	<code>loglog(x,y);</code>	
Input	<i>x</i>	Nx1 or NxM matrix. Each column contains the X values for a particular line.
	<i>y</i>	Nx1 or NxM matrix. Each column contains the Y values for a particular line.
Source	ploglog.src	
See also	<code>xy</code> , <code>logy</code> , <code>logx</code>	

logx

Purpose Graphs X vs. Y using log coordinates for the X axis.

Library `pgraph`

Format `logx( $x,y$ );`

Input

x	$N \times 1$ or $N \times M$ matrix. Each column contains the X values for a particular line.
y	$N \times 1$ or $N \times M$ matrix. Each column contains the Y values for a particular line.

Source `plogx.src`

See also `xy`, `logy`, `loglog`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

logy

logy

Purpose	Graphs X vs. Y using log coordinates for the Y axis.	
Library	pgraph	
Format	<code>logy(<i>x</i>,<i>y</i>) ;</code>	
Input	<i>x</i>	Nx1 or NxM matrix. Each column represents the X values for a particular line.
	<i>y</i>	Nx1 or NxM matrix. Each column represents the Y values for a particular line.
Source	plogy.src	
See also	<code>xy</code> , <code>logx</code> , <code>loglog</code>	

lower

Purpose	Converts a string or character matrix to lowercase.	
Format	$y = \text{lower}(x);$	
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	<pre>x = "MATH 401"; y = lower(x); print y;</pre> produces: math 401	
See also	upper	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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 $L = \text{lowmat}(x);$
 $L = \text{lowmat1}(x);$

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.

Example $x = \begin{Bmatrix} 1 & 2 & -1, \\ 2 & 3 & -2, \\ 1 & -2 & 1 \end{Bmatrix};$

$L = \text{lowmat}(x);$

$L1 = \text{lowmat1}(x);$

The resulting matrices are:

$L = \begin{Bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 1 & -2 & 1 \end{Bmatrix}$

$L1 = \begin{Bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{Bmatrix}$

Source diag.src

See also upmat, upmat1, diag, diagrv, crout, croutp

lpos

Purpose	Returns the current position of the print head within the printer buffer for the printer.
Format	<code>y = lpos;</code>
Portability	DOS only
Remarks	This function is basically equivalent to function csrcol but this returns the current column position for the standard printer. The value returned is the column position of the next character to be printed to the printer buffer. This does not necessarily reflect the actual physical position of the print head at the time of the call. If this function returns a number greater than 1, there are characters in the buffer for the standard printer which have not yet been sent to the printer. This buffer can be flushed at any time by lprint'ing a carriage return/line feed sequence, or a form feed character.
Example	<pre>if lpos > 60; lprint; endif;</pre> In this example, if the print buffer contains 60 characters or more, a carriage return/line feed sequence will be printed.
See also	lprint, lpwidth

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

lprint

lprint

Purpose Controls printing to the line printer.

Format `lprint` *[/typ] [/fnted] [/mf] [/jnt] [list of expressions separated by spaces] [:];*

Remarks This function was originally written for line printers. It is still supported for backwards compatibility purposes, but if you're using a page-oriented printer (such as a laser or inkjet printer), it may not give you the results you're expecting.

lprint statements work in essentially the same way that **print** statements work. The main difference is that **lprint** statements cannot be directed to the auxiliary output. Also, the **locate** statement has no meaning with **lprint**.

Two semicolons following an **lprint** statement will suppress the final line feed.

See **print** for information on */typ*, */fnted*, */mf*, and */jnt*.

A list of expressions is a list of GAUSS expressions, separated by spaces. In **lprint** statements, because a space is the delimiter between expressions, no spaces are allowed inside expressions unless they are within index brackets, they are in quotes, or the whole expression is in parentheses.

Printer width can be specified by the **lpwidth** statement:

```
lpwidth 132;
```

This statement remains in effect until cancelled. The default printer width is 80. That is, GAUSS automatically sends a line feed to the printer after printing 80 characters.

lpos can be used to determine the (column) position of the next character that will be printed in the buffer.

An **lprint** statement by itself will cause a blank line to be printed:

```
lprint;
```

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)
```

lprint

"\f" **form feed** (ASCII 12)
"\l" **line feed** (ASCII 10)
"\r" **carriage return** (ASCII 13)
" t" **tab** (ASCII 9)
"\###" the character whose ASCII value is "**###**" (decimal)

GAUSS also has an automatic line printer mode which causes the results of all global assignment statements to be printed out on the printer. This is controlled by the **lprint on** and **lprint off** commands. (See **lprint on**, **lprint off**.)

Example `lprint 3*4 5+2;`

See also `print, lprint on, lpos, lpwidth, format`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

lprint on, lprint off

lprint on, lprint off

This command is obsolete. See “**lprint on, lprint off**” on page A-9.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

lpwidth

This command is obsolete. See “**lpwidth**” on page A-11.

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

ltrisol

ltrisol

Purpose Computes the solution of $Lx = b$ where L is a lower triangular matrix.

Format $x = \text{ltrisol}(b, L);$

Input

b	PxK matrix.
L	PxP lower triangular matrix.

Output x PxK matrix.

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 `rndseed 13;`

`format /rd 10,4;`

`x = complex(rndn(3,3),rndn(3,3));`

`{ l,u } = lu(x);`

`x2 = l*u;`

x =
$$\begin{bmatrix} 0.1523 + 0.7685i & -0.8957 + 0.0342i & 2.4353 + 2.7736i \\ -1.1953 + 1.2187i & 1.2118 + 0.2571i & -0.0446 - 1.7768i \\ 0.8038 + 1.3668i & 1.2950 - 1.6929i & 1.6267 + 0.2844i \end{bmatrix}$$

l =
$$\begin{bmatrix} 0.2589 - 0.3789i & -1.2417 + 0.5225i & 1.0000 \\ 1.0000 & 0.0000 & 0.0000 \\ 0.2419 - 0.8968i & 1.0000 & 0.0000 \end{bmatrix}$$

u =
$$\begin{bmatrix} -1.1953 + 1.2187i & 1.2118 + 0.2571i & -0.0446 - 1.7768i \\ 0.0000 & 0.7713 - 0.6683i & 3.2309 + 0.6742i \\ 0.0000 & 0.0000 & 6.7795 + 5.7420i \end{bmatrix}$$

lu

$$\begin{matrix} & 0.1523 + 0.7685i & -0.8957 + 0.0342i & 2.4353 + 2.7736i \\ x2 = & -1.1953 + 1.2187i & 1.2118 + 0.2571i & -0.0446 - 1.7768i \\ & 0.8038 + 1.3668i & 1.2950 - 1.6929i & 1.6267 + 0.2844i \end{matrix}$$

See also crout, croutp, chol

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

lusol

Purpose	Computes the solution of $LUx = b$ where L is a lower triangular matrix and U is an upper triangular matrix.		
Format	$x = \text{lusol}(b, L, U);$		
Input	b	PxK matrix.	
	L	PxP lower triangular matrix.	
	U	PxP upper triangular matrix.	
Output	x	PxK matrix.	
Remarks	If b has more than one column, each column is solved for separately, i.e., lusol solves $LUx[:,i] = b[:,i]$.		

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

make (dataloop)

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 **externs**, 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 $ sex = lower(sex);`

See also **vector**

makevars

Purpose Creates separate global vectors from the columns of a matrix.

Format `makevars(x, vnames, xnames)`;

Input

<i>x</i>	NxK matrix whose columns will be converted into individual vectors.
<i>vnames</i>	string or Mx1 character vector containing names of global vectors to create. If 0, all names in <i>xnames</i> will be used.
<i>xnames</i>	string or Kx1 character vector containing names to be associated with the columns of the matrix <i>x</i> .

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**.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

makevars

```
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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

makewind

Purpose Creates a graphic panel of specific size and position and add it to the list of graphic panels.

Library pgraph

Format `makewind(xsize, ysize, xshft, yshft, typ);`

Input

<i>xsize</i>	scalar, horizontal size of the graphic panel in inches.
<i>ysize</i>	scalar, vertical size of the graphic panel in inches.
<i>xshft</i>	scalar, horizontal distance from left edge of window in inches.
<i>yshft</i>	scalar, vertical distance from bottom edge of window in inches.
<i>typ</i>	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 remains 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 “Publication Quality Graphics” in the *User’s Guide*.

Source pwindow.src

See also `window`, `endwind`, `setwind`, `getwind`, `begwind`, `nextwind`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

margin

margin

Purpose Sets the margins for the current graph graphic panel.

Library pgraph

Format `margin(l, r, t, b);`

Input

<i>l</i>	scalar, the left margin in inches.
<i>r</i>	scalar, the right margin in inches.
<i>t</i>	scalar, the top margin in inches.
<i>b</i>	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 9 x 6.855 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

maxc

Purpose Returns a column vector containing the largest element in each column of a matrix.

Format $y = \text{maxc}(x);$

Input x NxK 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**:

$y = \text{maxc}(\text{maxc}(x));$

Example $x = \text{rndn}(4, 2);$

$y = \text{maxc}(x);$

$x =$

-2.124474	1.376765
0.348110	1.172391
-0.027064	0.796867
1.421940	-0.351313

$y =$

1.421940
1.376765

See also **minc**, **maxindc**, **minindc**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

maxindc

maxindc

Purpose	Returns a column vector containing the index (i.e., row number) of the maximum element in each column in a matrix.		
Format	$y = \text{maxindc}(x);$		
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, maxc 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. 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);
y = maxc(x);
z = maxindc(x);
```

```
      1 -11  0  5
x =   0  0 -2 -6
     -8  0  3  2
     -11 5 -4  5

      1
y =   5
      3
      5
```

maxindc

$$z = \begin{matrix} 1 \\ 4 \\ 3 \\ 1 \end{matrix}$$

See also maxc, minindc, minc

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

maxvec

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. This must never be set to 8190.

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**.

On systems without virtual memory you can use 536870910. Otherwise a smaller value like 20000-30000 is necessary to prevent excessive disk thrashing. The trick is to allow the algorithm making the disk reads to execute entirely in RAM.

Example `y = maxvec;`
`print y;`
produces:
20000.000

Source `system.src`

mbesseli

Purpose Computes modified and exponentially scaled modified Bessels of the first kind of the n^{th} 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, $0 \leq alpha < 1$.

Output

y KxN matrix, evaluations of the modified Bessel or the exponentially scaled modified Bessel of the first kind of the n^{th} orders.

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 :

$$y[i, :] = I_{\alpha}(x[i]) \ I_{\alpha+1}(x[i]) \ \dots \ I_{\alpha+n-1}(x[i])$$

The remaining functions generate modified Bessels of only the specified order.

The exponentially scaled modified Bessels are related to the unscaled modified Bessels in the following way:

$$\text{mbesselei0}(x) = \exp(-x) * \text{mbesseli0}(x)$$

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

mbesseli

University Press, 1993.), where the dependent variable varies between $-\pi$ and π in a circular manner. The model is

$$y = \mu + G(XB)$$

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 $[-\pi, \pi]$ interval.

The log-likelihood for this model is from Fisher, N.I. ... 1993, 159;

$$\log L = -N x \ln((I_0(\kappa)) + \kappa) \sum_i^N \cos(y_i - \mu - G(X_i B))$$

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 $[-\pi, \pi]$ 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(mbesseli0(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(x),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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

meanc**meanc**

Purpose Computes the mean of every column of a matrix.

Format $y = \text{meanc}(x);$

Input x NxK matrix.

Output y Kx1 matrix containing the mean of every column of x .

Example $x = \text{meanc}(\text{rndu}(2000, 4));$

```
0.492446
0.503543
0.502905
0.509283
```

In this example, 4 columns of uniform random numbers are generated in a matrix, and the mean is computed for each column.

See also `stdc`

median

Purpose Computes the medians of the columns of a matrix.

Format $m = \text{median}(x);$

Input x NxK matrix.

Output m Kx1 vector containing the medians of the respective columns of x .

Example

```
x = { 8 4,  
      6 8,  
      3 7 };  
y = median(x);  
  
y = 6.00000000  
    7.00000000
```

Source median.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

medit

medit

Purpose Optional full-window matrix editor. Handles both character and numeric elements.

Format { *y*, *yv*, *yfmt* } = **medit**(*x*, *xv*, *xfmt*);

Input

<i>x</i>	LxM matrix to be edited.
<i>xv</i>	scalar, vector or matrix. <i>xv</i> is reshaped into a 1xM vector. The nonzero elements in <i>xv</i> mark the respective columns of <i>x</i> as numeric, 0's mark them as character.
<i>xfmt</i>	scalar, string or matrix. <i>xfmt</i> sets the initial column formats. If <i>xfmt</i> is a scalar then the following default formats are used:

Type	Format	Width	Precision
Numeric column	"*.*lg"	16	8
Character column	"*.*s"	8	8

If *xfmt* is a string, the first 8 characters are used for the format and the precision and field are set to a default value to give the global column format.

If *xfmt* is a matrix, it must have 3 columns. It will be reshaped to an Mx3 format matrix.

In all cases, the type values in *xv* override the formats specified in *xfmt*.

Output

<i>y</i>	LxM edited matrix.
<i>yv</i>	1xM vector of ones and zeros, 1 if the respective column of <i>y</i> is numeric, 0 if it is character.
<i>yfmt</i>	Mx3 matrix, each row containing format information (suitable for use with printfm) for the respective column of <i>y</i> .

Portability DOS only

Remarks **medit** displays only the real part of the matrix if the imaginary part is missing or all zeros. The display changes to complex when an edited element has a nonzero imaginary part or a row or column is added with a complex fill value.

ALT+X terminates the editor. If you are editing an element or executing a command, press ESC to abandon the element or terminate the command.

When ALT+X is used to terminate the editor, the edited matrix is saved in the file `_medit_x.fmt`. ESC also exits from the editor, but the results of the editing are not returned and the matrix is not saved to the file. The original value of the matrix remains unchanged.

The following keys allow you to move around the matrix:

LEFT ARROW	Move one cell left.
RIGHT ARROW	Move one cell right.
UP ARROW	Move one cell up.
DOWN ARROW	Move one cell down.
PAGE UP	Move one page up.
PAGE DOWN	Move one page down.
CTRL+LEFT	Move one page left.
CTRL+RIGHT	Move one page right.
HOME	Beginning of row.
HOME HOME	Beginning of matrix.
END	End of row.
END END	End of matrix.
ENTER	Move forward one cell.
BACKSPACE	Move back one cell.
CTRL+ENTER	Toggle the direction of ENTER and BACKSPACE.
ALT+G	Go to row and column.

The following alternate Wordstar keystrokes are supported:

CTRL+G	DEL	CTRL+R	PGUP
CTRL+H	BACKSPACE	CTRL+C	PGDN
CTRL+S	LEFT	CTRL+A	CTRL+LEFT
CTRL+D	RIGHT	CTRL+F	CTRL+RIGHT
CTRL+E	UP	CTRL+Q+S	HOME
CTRL+X	DOWN	CTRL+Q+D	END

The operation of the ENTER and BACKSPACE keys depends on the CTRL+ENTER setting. CTRL+ENTER toggles the direction of movement for ENTER from right to down to no movement. BACKSPACE moves in the opposite direction to ENTER. The arrow at the top left of the matrix indicates the current setting.

ALT+G allows you to jump to any element in the matrix. Pressing ALT+G prompts you for the row and column. Enter the row and column

medit

numbers separated by either a space or a comma and then press ENTER to execute the command. Invalid entries will cause a beep and can be edited. Pressing ESC will abandon the command.

The following editing keys are available to you when editing a matrix element:

LEFT ARROW	Move left.
RIGHT ARROW	Move right.
DELETE	Delete character at cursor.
BACKSPACE	Delete character to left of cursor.
HOME	Move to beginning of input.
END	Move to end of input.
ENTER	Save the new value.
CTRL+ENTER	Toggle the direction of movement.
ALT+P	Inserts π on the input line.
ALT+E	Inserts e on the input line.
ESC	Abandon editing of the element.

The current element is displayed at the top of the window. Character elements are displayed surrounded by “” (quotes). Numeric elements are displayed in exponential format to full precision. To edit the current element, just type in the new value. To save the new value, press ENTER. If the value is valid, it will be saved. Invalid values will cause a beep and can be edited. Pressing ESC abandons the editing and leaves the element unchanged.

Numbers can be entered in either integer, floating point, or exponential format. To enter a missing value in a numeric element, type ‘.’ and press ENTER. If the numeric value entered overflows, ∞ with the appropriate sign will be stored. If the numeric value entered underflows, +0 will be stored.

Complex numbers are of the form *number* $\pm$ *numberi* or *numberi*, e.g., 0.5-1i, . + .i, 5i. Note that there must be a number before the “i”, i.e., 5+i is invalid. If the complex part is zero it is not displayed by **medit** or printed by **printfm**.

Character elements are limited to 8 characters. To enter an empty character string, press the SPACEBAR and then BACKSPACE to delete the space, then press ENTER to store the empty string. All strings are padded to 8 characters with nulls before storing.

The following command keys are available to you when moving around the matrix:

ALT+H	Help.
ESC	Abandon matrix and quit medit .
ALT+X	Exit medit . Result saved in <code>_medit_x.fmt</code> .
ALT+R	Select rows.
ALT+C	Select columns.
ALT+V	Set fill value, use “” for character fill values.
ALT+I or INSERT	Insert row, column, or block.
ALT+F	Format column.

ALT+R and ALT+C allow a block of rows or columns to be selected using any of the cursor movement keys. The selected block can then be cut to the scrap buffer using keypad -, copied using keypad +, or the block can be deleted by pressing DELETE.

ALT+F allows the format of the current column to be changed. ALT+F+N sets the default numeric format, ALT+F+C sets the default character format and ALT+F+E allows you to edit the column format. The variable type vector *yv* and format matrix *yfmt* are updated to reflect the new format.

Using the format editor, the column format can be selected as well as the trailing character, column width, and precision. The format editor is intelligent and will automatically adjust the column width or precision to ensure all numbers in the column can be displayed in the selected format. Very large numbers in the column will cause the decimal format to automatically switch to an exponential format.

printfm can be used to print the matrix in the format displayed by **medit**.

```
{ x,v,fmt } = medit(x,1,1);
call printfm(x,v,fmt);
```

Note however that UNN's in **medit** are printed as numbers by **printfm**.

ALT+V sets the fill value to be used for inserting new rows and columns. Pressing ALT+V displays the current value and allows a new value to be entered. Enclose character values with “”. Character values must not exceed 8 characters. ESC abandons the entry of the new value and leaves the current fill value unchanged. The default fill value is a missing value.

medit

The insert commands work with the fill value to insert a new row or column into the matrix in front of the current cursor position. The new row or column is filled with the current fill value. ALT+I+R inserts a new row and ALT+I+C inserts a new column. For columns, the type of the fill value determines the type of the column. The ALT+I+B command allows a previously cut or copied block to be inserted in front of the cursor position. INSERT can be used instead of ALT+I.

Example `{ x,v,fmt } = medit(zeros(10,5),1,1);`

This example edits a matrix of zeros and assigns the result to **x**. The second and third arguments are reshaped to the correct size and the column formats are set to the default numeric format. These vectors are then updated by the format commands during the editing and the results assigned to **v** and **fmt**.

```
    { x,v,fmt } = medit(x,1~0,le);
```

This example sets all the odd columns to a scientific format and sets all the even columns to the default character format.

```
    def_fmt = "-lf,"~4~3;
```

```
    { x,v,fmt } = medit(x,1,def_fmt);
```

This example sets all the columns to decimal format, left-justified with a trailing comma. It also sets the precision to 3 and the minimum width to 4. The necessary width will be automatically determined by **medit**.

Source `medit.src`

Globals `_med_1, _med_2R, _med_2C`

See also `printfm, con, editm`

mergeby

Purpose Merges two sorted files by a common variable.

Format `mergeby(infile1, infile2, outfile, keytyp)`;

Input

<i>infile1</i>	string, name of input file 1.
<i>infile2</i>	string, name of input file 2.
<i>outfile</i>	string, name of output file.
<i>keytyp</i>	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 the key variable in the first column.
2. All of the values of the key variable within a file are unique.
3. Each file is already sorted on that 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 of *infile2* minus the key column from the second file.

If the inputs are null or 0, the procedure will ask for them.

Example `mergeby("freq", "freqdata", "mergex", 1);`

Source `sortd.src`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

mergevar

mergevar

Purpose	Accepts a list of names of global matrices, and concatenates the corresponding matrices horizontally to form a single matrix.
Format	<code>x = mergevar(vnames);</code>
Input	<i>vnames</i> string or Kx1 column vector containing the names of K global matrices.
Output	<i>x</i> NxM matrix that contains the concatenated matrices, where M is the sum of the columns in the K matrices specified in <i>vnames</i> .
Remarks	The matrices specified in <i>vnames</i> must be globals and they must all have the same number of rows. This function is the opposite of makevars.
Example	<pre>let vnames = age pay sex; x = mergevar(vnames);</pre> The matrices age, pay, and sex will be concatenated horizontally to create x.
Source	vars.src
See also	makevars

minc

Purpose Returns a column vector containing the smallest element in each column in a matrix.

Format `y = minc(x);`

Input `x` NxK 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 = randn(4,2);`

```
y = minc(x);
```

```

      -1.061321 -0.729026
x =  -0.021965  0.184246
      1.843242 -1.847015
      1.977621 -0.532307

```

```

y =  -1.061321
     -1.847015

```

See also `maxc`, `minindc`, `maxindc`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

minindc

minindc

Purpose	Returns a column vector containing the index (i.e., row number) of the smallest element in each column in a matrix.
Format	$y = \text{minindc}(x);$
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. 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);
```

```

      -5  6 -4 -1
      2 -2  1  3
x =   6  0  1 -7
      -6  0  8 -4
      7 -4  8  3
```

```

      -6
y =   -4
      -4
      -7
```

minindc

```
      4
x =   5
      1
      3
```

See also `maxindc`, `minc`, `maxc`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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 $y = \text{miss}(x, v);$
 $y = \text{missrv}(x, v);$

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**, **ismiss**, **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.

Example

```
v = -1~4~5;
y = miss(x,v);
```

If **x** has 3 columns, all -1's in the first column will be changed to missings, along with all 4's in the second column and 5's in the third column.

See also

counts, ismiss, maxc, maxindc, minc, minindc, missex, moment, packr, scalmiss, sortc

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

missex

missex

Purpose	Converts numeric values to the missing value code according to the values given in a logical expression.	
Format	$y = \text{missex}(x, e);$	
Input	x	NxK matrix.
	e	NxK logical matrix (matrix of 0's and 1's) that serves as a "mask" for x ; the 1's in e 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:	
	$y = \text{missex}(x, (x .> 10) .\text{and} (x .< 15));$	
	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.	
Example	This function is like miss , but is more general in that a range of values can be converted into missings.	
	$x = \text{rndu}(3, 2);$	
	/* logical expression */	
	$e = (x .> .10) .\text{and} (x .< .20);$	
	$y = \text{missex}(x, e);$	
	A 3x2 matrix of uniform random numbers is created. All values in the interval (0.10, 0.20) 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 $y = \text{moment}(x, d);$

Input

x	NxK matrix.
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 which equals $x'x$.

Remarks The fact that the moment matrix is symmetric is taken into account to cut execution time almost in half.

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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

moment

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.

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

momentd

Purpose Computes a moment ($X'X$) matrix from a GAUSS data set.

Format `m = momentd(dataset, vars);`

Input

<i>dataset</i>	string, name of data set.
<i>vars</i>	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.

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.

Global Input

<code>__con</code>	scalar, default 1. 1 a constant term will be added. 0 no constant term will be added.
<code>__miss</code>	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 <i>m</i> .
<code>__row</code>	scalar, default 0, the number of rows to read per iteration of the read loop. If 0, the number of rows will be calculated internally. If you get an Insufficient memory error message, 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 $M = K + \text{__con}$, 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.

momentd

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`

Globals `__con, __miss, __row`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

msym

Purpose	Allows the user to set the symbol that GAUSS uses when missing values are converted to ASCII and vice versa.
Format	msym <i>str</i> ;
Input	<i>str</i> 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, lprint, 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.
See also	print , lprint , printfm

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

nametype

nametype

Purpose Provides support for programs following the upper/lowercase convention in GAUSS data sets. (See “File I/O” in the *User’s Guide*.) Returns a vector of names of the correct case and a 1/0 vector of type information.

Format { *vname*, *vtype* } = **nametype**(*vname*, *vtype*);

Input

vname Nx1 character vector of variable names.

vtype scalar or Nx1 vector of 1’s and 0’s to determine the type and therefore the case of the output *vname*. If this is scalar 0 or 1 it will be expanded to Nx1. If -1, **nametype** will assume that *vname* follows the upper/lowercase convention.

Output

vname Nx1 character vector of variable names of the correct case, uppercase if numeric, lowercase if character.

vtype Nx1 vector of ones and zeros, 1 if variable is numeric, 0 if character.

Example

```
vn = { age, pay, sex };
vt = { 1, 1, 0 };
{ vn, vt } = nametype(vn,vt);
print $vn;
```

```
      AGE
vn =   PAY
      sex
```

Source nametype.src

Globals vartype

ndpchk

This command is obsolete. See “**ndpchk**” on page A-12.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ndpclex

ndpclex

This command is obsolete. See “**ndpclex**” on page A-14.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ndpcntrl

This command is obsolete. See “**ndpcntrl**” on page A-15.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

new

new

Purpose Erases everything in memory including the symbol table; closes all open files, 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 [[nos [,mps ]];`

Input

nos scalar, which indicates the maximum number of global symbols allowed. See your platform supplement for the maximum number of globals allowed in this implementation.

mps scalar, which indicates the number of bytes of main program space to be allocated. See your platform supplement for the maximum amount allowed in this implementation.

Portability UNIX

new closes all open graphic panels except graphic panel 1.

Remarks Procedures, user-defined functions, and global matrices strings and string arrays are all global symbols.

The main program space is the amount of memory available for nonprocedure, nonfunction program code.

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 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. */

new
200,100000;  /* clear global symbols, set */
              /* maximum number of global */
              /* symbols to 200, and allocate */
              /* 100000 bytes for main */
              /* program code. */

new ,100000; /* clear global symbols, */
              /* allocate 100000 bytes for */
              /* main program code, and leave */
              /* maximum number of globals */
              /* unchanged. */
```

See also `clear`, `delete`, `output`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

nextn, nextnevn**nextn, nextnevn**

Purpose Returns allowable matrix dimensions for computing FFT's.

Format $n = \text{nextn}(n0);$
 $n = \text{nextnevn}(n0);$

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. $n \geq n0$.

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:

$$2^p \times 3^q \times 5^r \times 7^s, \quad p, q, r \text{ nonnegative integers} \\ s = 0 \text{ or } 1$$

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 $n = \text{nextn}(456);$

$n = 480$

nextn, nextnevn

Source `optim.src`

See also `fftn, optn, optnevn, rfftn, rfftnp`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

nextwind

nextwind

Purpose Sets the current graphic panel to the next available graphic panel.

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 “Publication Quality Graphics” in the *User’s Guide*.

Source pwindow.src

See also endwind, begwind, setwind, getwind, makewind, window

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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:

$$x*b = 0 \quad (\text{NxK matrix of zeros})$$

and

$$b'b = I \quad (\text{MxM identity matrix})$$

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;
```

Source `null.src`

Globals `_qrdc, _qrs1`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

null1

null1

Purpose Computes an orthonormal basis for the (right) null space of a matrix.

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 zeros)
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, _qrs1

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.
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.
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** global 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** global scalar, default 1.
1 a constant term will be added, D = K+1.
0 no constant term will be added, D = K.

ols

A constant term will always be used in constructing the moment matrix *m*.

- __miss

global 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.
- __output

global scalar, default 1.

1

print the statistics.

0

do not print statistics.
- __row

global 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 **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

global scalar, default 0.

1

compute residuals (*resid*) and Durbin-Watson statistic(*dwstat*).

0

resid = 0, *dwstat* = 0.

Output	<i>vnam</i>	(K+2)×1 or (K+1)×1 character vector, the variable names used in the regression. If a constant term is used, this vector will be (K+2)×1, and the first name will be “CONSTANT”. The last name will be the name of the dependent variable.		
	<i>m</i>	M×M 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	<i>indvars</i>	<i>depvar</i>
		(constant)	(independent variables)	(dependent variable)
		A constant term is always used in computing <i>m</i> .		
	<i>b</i>	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

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**.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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** was put into a file called `ols.out` as well as being printed in the window. This example will compute a least squares regression of **y** on **x**. The return values were 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,depvar,indvars);`

`output off;`

In this example, the data set `olsdat.dat` was 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.

ols

Source	ols.src
Globals	_olsres, _olsrnam, __altnam, __con, __miss, __output, __row, __vpad
See also	olsqr

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

olsqr

olsqr

Purpose Computes OLS coefficients using QR decomposition.

Format $b = \text{olsqr}(y, x);$

Input

y	$N \times 1$ vector containing dependent variable.
x	$N \times P$ matrix containing independent variables.

Global Input `_olsqtol` global scalar, the tolerance for testing if diagonal elements are approaching zero. The default value is $10e-14$.

Output

b	$P \times 1$ 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.

Source `olsqr.src`

Globals `_olsqtol`

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

<i>y</i>	Nx1 vector containing dependent variable.
<i>x</i>	NxP matrix containing independent variables.

Global Input `_olsqtol` global scalar, the tolerance for testing if diagonal elements are approaching zero. The default value is 10e-14.

Output

<i>b</i>	Px1 vector of least squares estimates of regression of <i>y</i> on <i>x</i> . If <i>x</i> does not have full rank, then the coefficients that cannot be estimated will be zero.
<i>r</i>	Px1 vector of residuals. ($r = y - x*b$)
<i>p</i>	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.

Source `olsqr.src`

Globals `_olsqtol`

See also `olsqr, orth, qqr`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

ones

ones

Purpose Creates a matrix of ones.

Format `y = ones(r, c);`

Input

<i>r</i>	scalar, number of rows.
<i>c</i>	scalar, number of columns.

Output *y* RxC matrix of ones.

Remarks Noninteger arguments will be truncated to an integer.

Example `x = ones(3, 2);`

$$x = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

See also `zeros`, `eye`

open

Purpose Opens an existing GAUSS data file.

Format `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 **[[read]], append, update**

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.

open

The optional **varindx**i 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* 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 **varindx**i and **varindx** options cannot be used with **.fmt** matrix files because no column names are stored with them.

If **varindx**i is used, GAUSS will ignore the **Undefined symbol** error message 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 **varindx**i 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 message.

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 **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** 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 **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);
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

open

In the example above, the existing data set `/data/rawdat.dat` is opened for appending new data. The name of the file was 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 well 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**, **rpm**.

```
open f1 = dat1 varindxi;
dtx = readr(f1,100);
x = dtx[.,irpm ilub ivisc];
y = dtx[.,itemp];
call seekr(f1,1);
```

open

In this example, the data set `dat1.dat` is opened for reading (the `.dat` and the **for read** are implicit). **varindx** 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 = dat1 varindx;
open q2 = 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 give the matrix **y**. The index variables will thus give the correct positions of the variables in **y**.

```
open fx = x.fmt;
i = 1; rf = rowsf(fx);
sampsiz = round(rf*0.1);
rndsmpx = zeros(sampsiz,colsf(fx));
do until i > sampsiz;
    r = ceil(rndu(1,1)*rf);
    call seekr(fx,r);
    rndsmpx[i,.] = readr(fx,1);
    i = i+1;
endo;
fx = close(fx);
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

open

In this example, a 10% random sample of rows is drawn from the matrix file `x.fmt` and put into the matrix **rndsmplx**. 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 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 **rndsmplx**. This is continued until the complete sample has been obtained.

See also **create, close, closeall, readr, writer, seekr, eof**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

optn, optnevn

Purpose	Returns optimal matrix dimensions for computing FFT's.	
Format	$n = \text{optn}(n0);$ $n = \text{optnevn}(n0);$	
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. $n \geq n0$.
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: $2^p \times 3^q \times 5^r \times 7^s, \quad p, q, r \text{ nonnegative integers}$ $s = 0 \text{ or } 1$ 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 percent faster than a 32768x1 vector, because 33600 is a highly composite number, $2^6 \times 3 \times 5^2 \times 7$, 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.	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

optn, optnevn

Use the following table to determine what to call for a given function and matrix:

FFT Function	Vector Length	Matrix Rows	Matrix Columns
fftn	optn	optn	optn
rfftn	optnevn	optn	optnevn
rfftnp	optnevn	optn	optnevn

Example n = optn(231);
 n = 240.00000

See also fftn, nextn, nextnevn, rfftn, rfftnp

orth

Purpose	Computes an orthonormal basis for the column space of a matrix.
Format	$y = \text{orth}(x);$
Input	x NxK matrix.
Global Input	<code>_orthtol</code> global scalar, the tolerance for testing if diagonal elements are approaching zero. The default is $1.0e-14$.
Output	y NxL matrix such that $y'y = \mathbf{eye}(L)$ and whose columns span the same space as the columns of x ; L is the rank of x .
Example	$x = \begin{Bmatrix} 6 & 5 & 4, \\ 2 & 7 & 5 \end{Bmatrix};$ $y = \text{orth}(x);$ $y = \begin{bmatrix} -0.58123819 & -0.81373347 \\ -0.81373347 & 0.58123819 \end{bmatrix}$
Source	<code>qqr.src</code>
Globals	<code>_orthtol</code>
See also	<code>qqr</code> , <code>olsqr</code>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

output

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]] [[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.

Portability **UNIX**
output commands affect the active graphic panel. Each graphic panel “remembers” its own settings, even when it is no longer the active graphic panel.
Each graphic panel can have its own auxiliary output file, or several graphic panels can write to the same output file.

DOS

filename can be any legal file name or applicable device. Therefore, **CON**, **LPT1**, **LPT2** and **LPT3** are all legal.

In interactive mode, CTRL+F3 will close the auxiliary output file and load it into the editor. That is, after a program has been run, pressing CTRL+F3 will perform **output off** and load the output file into the editor. The file must be closed prior to being edited.

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 exit 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 in 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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

output

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)

outtyp (dataloop)

Purpose Specifies the precision of the output data set.

Format outtyp num_constant;

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;

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

outwidth

outwidth

Purpose Specifies the width of the auxiliary output.

Format `outwidth n;`

Portability UNIX
outwidth affects the active graphic panel. Each graphic panel “remembers” its own setting, even when it is no longer the active graphic panel.

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 expression 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 `lpwidth`, `output`, `print`

packr

Purpose	Deletes the rows of a matrix that contain any missing values.
Format	$y = \text{packr}(x);$
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	<pre>x = miss(ceil(rndu(3,3)*10),1); y = packr(x);</pre> $x = \begin{bmatrix} . & 9 & 10 \\ 4 & 2 & . \\ 3 & 4 & 9 \end{bmatrix}$ $y = \begin{bmatrix} 3 & 4 & 9 \end{bmatrix}$ In this example, the matrix x is formed with random integers and missing values. packr is used to delete rows with missing values.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

packr

```
open fp = mydata;
obs = 0;
sum = 0;
do until eof(fp);
    x = packr(readr(fp,100));
    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, misrv**

parse

Purpose	Parses a string, returning a character vector of tokens.		
Format	<i>tok</i> = parse (<i>str</i> , <i>delim</i>) ;		
Input	<i>str</i>	string consisting of a series of tokens and/or delimiters.	
	<i>delim</i>	NxK character matrix of delimiters that might be found in <i>str</i> .	
Output	<i>tok</i>	Mx1 character vector consisting of the tokens contained in <i>str</i> . All tokens are returned; any delimiters found in <i>str</i> are ignored.	
Remarks	The tokens in <i>str</i> must be 8 characters or less in size. If they are longer, the contents of <i>tok</i> is unpredictable.		
See also	token		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

pause

pause

Purpose	Pauses for a specified number of seconds.
Format	<code>pause(<i>sec</i>) ;</code>
Input	<i>sec</i> seconds to pause.
Source	<code>pause.src</code>
See also	<code>wait</code>

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

pdfn

Purpose	Computes the standard Normal (scalar) probability density function.
Format	$y = \text{pdfn}(x);$
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: $y = (1/\text{sqrt}(2*\text{pi})) * \exp(-(x.*x)/2);$
Example	<pre>x = rndn(2, 2); y = pdfn(x);</pre> $x = \begin{bmatrix} -1.828915 & 0.514485 \\ -0.550219 & -0.275229 \end{bmatrix}$ $y = \begin{bmatrix} 0.074915 & 0.349488 \\ 0.342903 & 0.384115 \end{bmatrix}$

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

pi

pi

Purpose Returns the mathematical constant π .

Format `y = pi;`

Example `format /rdn 16,14;`
`print pi;`

produces:
3.14159265358979

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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	$y = \text{pinv}(x);$
Input	x NxM matrix.
Global Input	_svdtol global 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 global scalar, if not all of the singular values can be computed _svderr will be nonzero.
Example	$x = \{ \ 654, \ 275 \ \} ;$ $y = \text{pinv}(x);$ $y = \begin{bmatrix} 0.22017139 & -0.16348055 \\ -0.05207647 & 0.13447594 \\ -0.0151615 & 0.07712591 \end{bmatrix}$
Source	svd.src
Globals	_svdtol , _svderr

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

plot

plot

Purpose Plots the elements of two matrices against each other in text mode.

Format `plot x, y;`

Portability OS/2, Windows
plot writes to the main GAUSS graphic panel.

UNIX

plot writes to the active graphic panel. It is supported only for Text graphic panels.

Remarks *x* and *y* are two matrices that must be conformable in the standard fashion of element-by-element operators. Horizontal elements come from the first matrix listed. Vertical coordinates come from the second matrix listed. Noninteger elements are rounded to the nearest integer to obtain the coordinates.

Plotting is done in text mode in the window, not graphics mode.

All that is displayed by **plot** are the points being plotted. Axes are not displayed.

The window is treated as an axis system, with (0,0) in the lower left, and (79,24) in the upper right of the window.

If coordinates fall outside of the allowable range, points are plotted along the corresponding edge of the window.

The symbol or character to be plotted is controlled by the **plotsym** statement. The default symbol is an asterisk (*).

Example

```
print "Number of points: ";
n = con(1,1);
cls;
x = floor(rndu(n,1)*80);
y = floor(rndu(n,1)*25);
plot x,y;
```

plot

In this example two matrices of uniform random numbers, scaled so their elements will fall in the right range, are plotted. The **con** command is used to ask you to specify the size of the matrices.

See also **graph, plotsym, screen**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

plotsym

plotsym

Purpose Controls the symbol or character to be plotted by **plot**.

Format **plotsym** *n*;

Remarks *n* is a scalar containing the ASCII value of the symbol to be plotted. This value must be in the range 0-255.

n may be any legal expression that returns a scalar. Nonintegers will be truncated to an integer before being evaluated by **plotsym**.

The default symbol is an asterisk (*), which has ASCII value 42. Thus, if no **plotsym** statement is encountered, an asterisk will be used by **plot**.

The **plotsym** statement remains in effect until the next **plotsym** statement is encountered.

Example **plotsym** 249;

Assuming a standard ASCII terminal font (as under DOS), this example will cause centered dots to be used as the symbol for **plot**.

See also **plot**, **graph**, **screen**

polar

Purpose	Graphs data using polar coordinates.		
Library	pgraph		
Format	polar(<i>radius</i> , <i>theta</i>);		
Input	<i>radius</i>	Nx1 or NxM matrix. Each column contains the magnitude for a particular line.	
	<i>theta</i>	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		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

polychar

polychar

Purpose	Computes the characteristic polynomial of a square matrix.
Format	$c = \text{polychar}(x);$
Input	x NxN matrix.
Output	c (N+1)x1 vector of coefficients of the N^{th} order characteristic polynomial of x : $p(z)=c[1]*z^n + c[2]*z^{(n-1)} + ... + c[n]*z + c[n+1];$
Remarks	The coefficient of z^n is set to unity ($c[1]=1$).
Source	poly.src
See also	polymake, polymult, polyroot, polyeval

polyeval

Purpose Evaluates polynomials. Can either be 1 or more scalar polynomials or a single matrix polynomial.

Format $y = \text{polyeval}(x, c);$

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):

$$y = (c[1,.].*x^p + c[2,.].*x^{(p-1)} + \dots + c[p+1,.])';$$

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.

$$y = \text{polyeval}(x, 1|\text{zeros}(n, 1));$$

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

polyint

polyint

Purpose Calculates an N^{th} order polynomial interpolation.

Format `y = polyint(xa,ya,x);`

Input

<i>xa</i>	Nx1 vector, <i>X</i> values.
<i>ya</i>	Nx1 vector, <i>Y</i> values.
<i>x</i>	scalar, <i>X</i> value to solve for.

Global Input `_poldeg` global scalar, the degree of polynomial required, default 6.

Output *y* result of interpolation or extrapolation.

Global Output `_polerr` global scalar, interpolation error.

Remarks Calculates an N^{th} 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 N^{th} 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. Tevkolsky, and W.T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. NY: Cambridge Press, 1986.

polymake

Purpose	Computes the coefficients of a polynomial given the roots.
Format	$c = \text{polymake}(r);$
Input	r Nx1 vector containing roots of the desired polynomial.
Output	c (N+1)x1 vector containing the coefficients of the N^{th} order polynomial with roots r : $p(z)=c[1]*z^n + c[2]*z^{(n-1)} + ... + c[n]*z + c[n+1];$
Remarks	The coefficient of z^n is set to unity ($c[1]=1$).
Example	<pre> r = { 2, 1, 3 }; c = polymake(r); -1.0000000 c = -6.0000000 11.0000000 -6.0000000 </pre>
Source	poly.src
See also	polychar, polymult, polyroot, polyeval

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

polymat**polymat**

Purpose Returns a matrix containing the powers of the elements of x from 1 to p .

Format $y = \text{polymat}(x, p);$

Input

x	NxK matrix.
p	scalar, positive integer.

Output y Nx($p \times K$) matrix containing powers of the elements of x from 1 to p . The first K columns will contain first powers, the second K columns contain the 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

polymult

Purpose	Multiplies polynomials.		
Format	$c = \text{polymult}(c1, c2);$		
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	$c1 = \{ 2, 1 \};$		
	$c2 = \{ 2, 0, 1 \};$		
	$c = \text{polymult}(c1, c2);$		
		4.0000000	
	$c =$	2.0000000	
		2.0000000	
		1.0000000	
Source	poly.src		
See also	polymake, polychar, polyroot, polyeval		
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: $c1=5 0 6 3$. (Note that zeros must be explicitly given if there are powers of x missing.)		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

polyroot

polyroot

Purpose Computes the roots of a polynomial given the coefficients.

Format $y = \text{polyroot}(c);$

Input c $(N+1) \times 1$ vector of coefficients of an N^{th} order polynomial:
$$p(z) = c[1] * z^N + c[2] * z^{(N-1)} + \dots + c[n] * z + c[n+1]$$

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.

Output y $N \times 1$ vector, the roots of c .

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**'ed in the reverse order that they are **push**'ed, therefore the statements:

```
goto label(x,y,z);
```

```
·
·
·
```

```
label:
```

```
pop c;
```

```
pop b;
```

```
pop a;
```

```
c = z
```

```
b = y
```

```
a = x
```

Note that matrices are **pop**'ed in reverse order, and that there is a separate **pop** statement for each matrix popped.

See also `gosub, goto, return`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

prcsn

prcsn

Purpose Sets the computational precision of some of the matrix operators.

Format `prcsn n;`

Input *n* scalar, 64 or 80.

Portability **UNIX, Windows**
This function has no effect under UNIX or Windows. All computations are done in 64-bit precision.

Remarks *n* is a scalar containing either 64 or 80. The operators affected by this command are **chol**, **solpd**, **invpd**, and *b/a* (when neither *a* nor *b* is scalar and *a* is not square).

prcsn 80 is the default. Precision is set to 80 bits (10 bytes), which corresponds to about 19 digits of precision.

prcsn 64 sets the precision to 64 bits (8 bytes), which is standard IEEE double precision. This corresponds to 15-16 digits of precision. 80-bit precision is still maintained within the 80x87 math coprocessor so that actual precision is better than double precision.

When **prcsn** 80 is in effect, all temporary storage and all computations for the operators listed above are done in 80 bits. When the operator is finished, the final result is rounded to 64-bit double precision.

See also `chol`, `solpd`, `invpd`

princomp

Purpose	Computes principal components of a data matrix.	
Format	$\{p, v, a\} = \text{princomp}(x, j);$	
Input	x	NxK data matrix, $N > K$, full rank.
	j	scalar, number of principal components to be computed ($J \leq K$).
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.	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

print

print

Purpose Prints matrices or strings to the window and/or auxiliary output.

Format `print` `[/flush]` `[/typ]` `[/fnted]` `[/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 (`/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`.

`/fnted` literal, enable formatting flag.

`/on`, `/off`

Enable/disable formatting. When formatting is disabled, the contents of a variable are dumped to the window in a “raw” format.

`/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.

print

/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 $[] \text{ #####.#####}$ 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 $[] \text{ \#.\#\#E}\pm\text{####}$, 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 $[] \text{ \#E}\pm\text{####}$ 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.

print

/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 suppressed 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 $[] #.##E\pm###$, 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 $[] #E\pm###$ 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.

print

/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 suppressed 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 with a comma.

t The number will be followed immediately with a tab character.

n No trailing character.

The default when GAUSS is first started is:

```
format /m1 /r0 16,8;
```

;; Double semicolons following a **print** statement will suppress the final carriage return/line feed.

Portability UNIX

print writes to the active graphic panel. The refresh setting of the graphic panel determines when output is displayed. To force immediate display, use the **/flush** flag:

```
print /flush expr1 expr2 ...;;
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

print**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 followed by the character “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.

print

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.

A **print** statement by itself will cause a blank line to be printed:

```
print;
```

GAUSS also has an *automatic print mode* which causes the results of all global assignment statements to be printed out. This is controlled by the **print on** and **print off** commands. (See **print on**.)

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
-1.54675893	-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

print

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;
```

produces:

AGE

PAY

SEX

See also `lprint`, `print on`, `lprint on`, `printfm`, `printdos`

print on, print off

Purpose Switches the automatic window print mode on and off.

Format `print on;`
`print off;`

Portability UNIX
print on/off affects the active graphic panel. Each graphic panel “remembers” its own setting, even when it is no longer the active graphic panel.

Remarks After the **print on** command is encountered, the results of any global assignment statements (that is, statements that make assignments to global matrices or strings) will be printed in the window. Assignments to local matrices or strings will not be printed.

The name and dimensions of the resulting matrix will also be printed. If the result being assigned will be inserted into a submatrix of the target matrix, the name will be followed by an empty set of square brackets.

Example `y = rndn(1000,3);`

`print on;`

`s = stdc(y)';`

`m = meanc(y)';`

`print off;`

`mm = diag(y'y);`

produces:

`S[1,3]=`

1.021418 1.034924 1.040514

`M[1,3]=`

-0.047845 0.007045 -0.035395

In this example, a large (1000x3) matrix of random numbers is created, and some statistics are printed out using **print on**.

See also `lprint on`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

printdos

printdos

Purpose	Prints a string to the standard output.
Format	printdos <i>s</i> ;
Input	<i>s</i> string, containing the string to be printed to the standard output.
Portability	UNIX printdos writes to the active graphic panel. The refresh setting of the graphic panel determines when output is displayed. To force immediate display, use the WinRefresh command. DOS printdos uses the DOS interrupt 21 hex, function 9. The DOS function 9 uses a dollar sign as a terminator, so this function will not print a dollar sign.
Remarks	This function is useful for printing messages to the window when screen off is in effect. The output of this function will not go to the auxiliary output. This function can also be used to send escape sequences to the <code>ansi.sys</code> device driver.
Example	<pre>printdos "\27[7m"; /* set for reverse video */ printdos "\27[0m"; /* set for normal text */</pre> See the DOS manuals for more complete information.
See also	print, lprint, printfm, screen

printfm

Purpose Prints a matrix using a different format for each column of the matrix.

Format `y = printfm(x, mask, fmt);`

Input

<i>x</i>	NxK matrix which is to be printed and which may contain both character and numeric data.
<i>mask</i>	LxM matrix, ExE conformable with <i>x</i> , containing ones and zeros which is used to specify whether the particular row, column, or element is to be printed as a string (0) or numeric (1) value.
<i>fmt</i>	Kx3 or 1x3 matrix where each row specifies the format for the respective column of <i>x</i> .

Output

<i>y</i>	scalar, 1 if the function is successful and 0 if it fails.

Portability UNIX
printfm writes to the active graphic panel. The refresh setting of the graphic panel determines when output is displayed. To force immediate display, use the **WinRefresh** command.

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"

printfm

```

/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:

```

"*.l f,"  right-justified decimal followed by a comma.
"-.*.s"   left-justified string followed by a space.
"*.l f"   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*.l f" 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 */

```

printfm

```
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+000  
PAY 1.235, 1.2346E+000  
SEX 1.145, 3.4471E+000  
JOB 4.114, 8.5564E+000
```

When the column of *x* to be printed contains all string 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 string 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 strings.

Remember, the mask value controls whether an element will be printed as a number or a string.

See also `print`, `lprint`, `printdos`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

printfmt

printfmt

Purpose	Prints character, numeric, or mixed matrix using a default format controlled by the functions formatcv and formatnv .										
Format	$y = \text{printfmt}(x, \text{mask});$										
Input	<table><tr><td>x</td><td>NxK matrix which is to be printed.</td></tr><tr><td>mask</td><td>scalar, 1 if x is numeric or 0 if x is character.</td></tr><tr><td></td><td>or</td></tr><tr><td></td><td>1xK vector of 1's and 0's.</td></tr><tr><td></td><td>The corresponding column of x will be printed as numeric where $\text{mask} = 1$ and as character where $\text{mask} = 0$.</td></tr></table>	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 $\text{mask} = 1$ and as character where $\text{mask} = 0$.
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 $\text{mask} = 1$ and as character where $\text{mask} = 0$.										
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	$x = \text{rndn}(5, 4);$ call $\text{printfmt}(x, 1);$										
Source	gauss.src										
Globals	__fmtcv, __fmtnv										
See also	formatcv, formatnv										

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);`

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 */
```

proc

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” in the *User’s Guide*.

See also **keyword, call, endp, local, retp**

prodc

Purpose Computes the products of all elements in each column of a matrix.

Format $y = \text{prodc}(x);$

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 ($\cdot'$).

To find the products of all of the elements in a matrix, use the **vecr** function before applying **prodc**.

Example

```
let x[3,3] = 1 2 3
              4 5 6
              7 8 9;

y = prod(x);

      28
y =   80
      162
```

See also `sumc`, `meanc`, `stdc`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

putf

putf

Purpose Writes the contents of a string to a file.

Format *ret* = **putf**(*filename*, *str*, *start*, *len*, *mode*, *append*);

Input

<i>filename</i>	string, name of output file.
<i>str</i>	string to be written to <i>filename</i> . All or part of <i>str</i> may be written out.
<i>start</i>	scalar, beginning position in <i>str</i> of output string.
<i>len</i>	scalar, length of output string.
<i>mode</i>	scalar, output mode, (0) ASCII or (1) binary.
<i>append</i>	scalar, file write mode, (0) overwrite or (1) append.

Output *ret* scalar, return code.

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**.

ret can have the following values:

- 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** append specified but file did not exist; file was created
(warning only)

Source `putf.src`

See also `getf`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

QNewton

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

Portability UNIX

If you are running in (-v) mode, GAUSS will not see any input until you press ENTER.

Remarks Pressing C on the keyboard will terminate iterations, and pressing P will toggle iteration output.

Example This example computes maximum likelihood coefficients and standard errors for a Tobit model:

```
/*
** qnewton.e - a Tobit model
*/

z = load("tobit"); /* get data */
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;
```

QNewton

```
print "coefficients standard errors";
print b~sqrt(diag(invpc(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

QProg

Purpose Solves the quadratic programming problem.

Format $\{ x, u1, u2, u3, u4, ret \} = \text{QProg}(start, q, r, a, b, c, d, bnds);$

Input

<i>start</i>	Kx1 vector, start values.
<i>q</i>	KxK matrix, symmetric model matrix.
<i>r</i>	Kx1 vector, model constant vector.
<i>a</i>	MxK matrix, equality constraint coefficient matrix, or scalar 0, no equality constraints.
<i>b</i>	Mx1 vector, equality constraint constant vector, or scalar 0, will be expanded to Mx1 vector of zeros.
<i>c</i>	NxK matrix, inequality constraint coefficient matrix, or scalar 0, no inequality constraints.
<i>d</i>	Nx1 vector, inequality constraint constant vector, or scalar 0, will be expanded to Nx1 vector of zeros.
<i>bnds</i>	Kx2 matrix, bounds on <i>x</i> , the first column contains the lower bounds on <i>x</i> , and the second column the upper bounds. If scalar 0, the bounds for all elements will default to $\pm 1e200$.

Global Input `_qprog_maxit` scalar, maximum number of iterations.
Default = 1000.

Output

<i>x</i>	Kx1 vector, coefficients at the minimum of the function.
<i>u1</i>	Mx1 vector, Lagrangian coefficients of equality constraints.
<i>u2</i>	Nx1 vector, Lagrangian coefficients of inequality constraints.
<i>u3</i>	Kx1 vector, Lagrangian coefficients of lower bounds.
<i>u4</i>	Kx1 vector, Lagrangian coefficients of upper bounds.
<i>ret</i>	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

QProg

Remarks **QProg** solves the standard quadratic programming problem:

$$\min \frac{1}{2} x' Q x - x' R$$

subject to constraints,

$$A x = B$$

$$C x \geq D$$

and bounds,

$$x_{low} \leq x \leq x_{up}$$

Source `qprog.src`

qqr

Purpose Computes the orthogonal-triangular (QR) decomposition of a matrix X , such that:

$$X = Q_1 R$$

Format $\{ q1, r \} = \text{qqr}(x);$

Input x $N \times P$ matrix.

Output $q1$ $N \times K$ unitary matrix, $K = \min(N, P)$.
 r $K \times P$ 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

$$Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}$$

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 $X'X$.

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**.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qqr

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:

$$Q'X = \begin{bmatrix} R_1 & R_2 \end{bmatrix}$$

where R_1 is a $P \times P$ upper triangular matrix and R_2 is $P \times (N - P)$. Thus Q is a $P \times P$ matrix and R is a $P \times N$ 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:

$$X[:, E] = Q_1 R$$

Format $\{ q1, r, e \} = \text{qqre}(x);$

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.,

$$Q'X[:, E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X[:, E] = Q_1 R$$

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:

$$X[:, E] = \begin{bmatrix} X_1 & X_2 \end{bmatrix}$$

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qqre

where X_1 is $N \times M$ and X_2 is $N \times (P - M)$. Further partition R in the following way:

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

where R_{11} is $M \times M$ and R_{12} is $M \times (P - M)$. Then

$$A = R_{11}^{-1} R_{12}$$

and

$$X_2 = X_1 A$$

that is, A is an $M \times (P - N)$ matrix defining the linear combinations of X_2 with respect to X_1 .

If $N < P$ the factorization assumes the form:

$$Q'X = \begin{bmatrix} R_1 & R_2 \end{bmatrix}$$

where R_1 is a $P \times P$ upper triangular matrix and R_2 is $P \times (N - P)$. Thus Q is a $P \times P$ matrix and R is a $P \times N$ 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

$$\mathbf{b} = \mathbf{qrsol}(Q'Y, R1) \mid \mathbf{zeros}(N-P, 1);$$

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`, `qtyre`, `olsqr`

qqrep

Purpose Computes the orthogonal-triangular (QR) decomposition of a matrix X , such that:

$$X[:, E] = Q_1 R$$

Format $\{ q1, r, e \} = \text{qqrep}(x, pvt);$

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.,

$$Q'X[:, E] = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. If we partition

$$Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}$$

where Q_1 has P columns, then

$$X[:, E] = \begin{bmatrix} Q_1 & R \end{bmatrix}$$

is the QR decomposition of $X[:, E]$.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qqrep

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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qr

Purpose Computes the orthogonal-triangular (QR) decomposition of a matrix X , such that:

$$X = Q_1 R$$

Format $r = \text{qr}(x);$

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

$$Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}$$

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 $X'X$.

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 $Q'Y$, $Q'Y$, or $Q'Y$, $Q_1'Y$, for some Y , are required. For these cases see **qtyr** and **qyr**.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qr

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:

$$Q'X = \begin{bmatrix} R_1 & R_2 \end{bmatrix}$$

where R_1 is a $P \times P$ upper triangular matrix and R_2 is $P \times (N - P)$. Thus Q is a $P \times P$ matrix and R is a $P \times N$ 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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qre

Purpose Computes the orthogonal-triangular (QR) decomposition of a matrix X , such that:

$$X[:, E] = Q_1 R$$

Format $\{r, e\} = \text{qre}(x);$

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

$$X[:, E] = Q_1 R$$

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 $Q'Y$, $Q'Y$, or QY , Q_1Y , for some Y , are required. For these cases see **qtyre** and **qyre**.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qre

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:

$$X[., E] = \begin{bmatrix} X_1 & X_2 \end{bmatrix}$$

where X_1 is $N \times M$ and X_2 is $N \times (P - M)$. Further partition R in the following way:

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

where R_{11} is $M \times M$ and R_{12} is $M \times (P - M)$. Then

$$A = R_{11}^{-1} R_{12}$$

and

$$X_2 = X_1 A$$

that is, A is an $M \times (P - N)$ matrix defining the linear combinations of X_2 with respect to X_1 .

If $N < P$ the factorization assumes the form:

$$Q'X = \begin{bmatrix} R_1 & R_2 \end{bmatrix}$$

where R_1 is a $P \times P$ upper triangular matrix and R_2 is $P \times (N - P)$. Thus Q is a $P \times P$ matrix and R is a $P \times N$ 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

$$b = \text{qrsol}(Q'Y, R1) \mid \text{zeros}(N-P, 1);$$

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:

$$X[:, E] = Q_1 R$$

Format $\{ r, e \} = \text{qrep}(x, pvt);$

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

$$X[:, E] = Q_1 R$$

qrep

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_1Y , 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 $x = \text{qrsol}(b, R);$

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qrtsol

qrtsol

Purpose	Computes the solution of $R'x = b$ where R is an upper triangular matrix.		
Format	$x = \text{qrtsol}(b,R);$		
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 $\{ qty, r \} = \text{qtyr}(y, x);$

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,

$$G = Q_1'Y = qty[1 : P, .]$$

and $Q_2'Y$ is the remaining submatrix:

qtyr

$$H = Q_2' Y = qty[P + 1 : N, .]$$

Suppose that X is an $N \times K$ data set of independent variables, and v is an $N \times 1$ vector of dependent variables. Then it can be shown that

$$b = R^{-1} G$$

and

$$s_j = \sum_{i=1}^{N-P} H_{i,j}, j = 1, 2, \dots, L$$

where b is a $P \times L$ matrix of least squares coefficients and s is a $1 \times L$ 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 superior numerical 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),.];
b = qrsol(q1ty,r); /* LS coefficients */
s2 = sumc(q2ty^2); /* residual sums of squares */
```

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 $\{ qty, r, e \} = \text{qtyre}(y, x);$

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

$$X[., E] = Q_1 R$$

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:

$$X[., E] = \begin{bmatrix} X_1 & X_2 \end{bmatrix}$$

where X_1 is $N \times M$ and X_2 is $N \times (P - M)$. Further partition R in the following way:

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qtyre

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

where R_{11} is $M \times M$ and R_{12} is $M \times (P - M)$. Then

$$A = R_{11}^{-1} R_{12}$$

and

$$X_2 = X_1 A$$

that is, A is an $M \times (P - N)$ matrix defining the linear combinations of X_2 with respect to X_1 .

For most problems Q or Q_1 is not what is required. Rather, we require $Q'Y$ or $Q'_i Y$ where Y is an $N \times L$ 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. $Q'_i Y$ will be a submatrix of $Q'Y$. In particular,

$$Q'_1 Y = qty[1:P, .]$$

and $Q'_2 Y$ is the remaining submatrix:

$$Q'_2 Y = qty[P + 1 : N, .]$$

Suppose that X is an $N \times K$ data set of independent variables and Y is an $N \times 1$ vector of dependent variables. Suppose further that X contains linearly dependent columns, i.e., X has rank $M < P$. Then define

$$C = Q'_2 Y [1 : M, .]$$

$$A = R[1:M, 1:M]$$

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:

$$Q'X[:, E] = \begin{bmatrix} R_1 & R_2 \end{bmatrix}$$

where R_1 is a $P \times P$ upper triangular matrix and R_2 is $P \times (N - P)$. Thus Q is a $P \times P$ matrix and R is a $P \times N$ 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

$$b = \text{qrsol}(Q'Y, R1) \mid \text{zeros}(N-P, 1);$$

Source `qtyr.src`

See also `qqr`, `qre`, `qtyr`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qtyrep

qtyrep

Purpose Computes the orthogonal-triangular (QR) decomposition of a matrix X using a pivot vector and returns $Q'Y$ and R .

Format $\{ qty, r, e \} = \text{qtyrep}(y, x, pvt);$

Input

y	$N \times L$ matrix.
x	$N \times P$ matrix.
pvt	$P \times 1$ 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	$N \times L$ unitary matrix.
r	$K \times P$ upper triangular matrix, $K = \min(N, P)$.
e	$P \times 1$ 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

$$X[., E] = Q_1 R$$

is the QR decomposition of $X[., E]$.

qtyrep

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*.

Example

```
y = { 4 7 2,
      5 9 1,
      6 3 3 };
x = { 12 9 5,
      4 3 5,
      4 2 7 };
pvt = { 11, 10, 3 };
{ qty, r, e } = qtyrep(y,x,pvt);

      -6.9347609      -9.9498744  -3.0151134
qty =   4.0998891   3.5527137e-15   2.1929640
      3.4785054      6.3245553   0.31622777

      -13.266499   -9.6483630  -8.1408063
r =    0.00000000  -0.95346259   4.7673129
      0.00000000   0.00000000   3.1622777

      1.00000000
e =    2.00000000
      3.00000000
```

Source qtyr.src

See also qrep, qtyre

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

quantile

quantile

Purpose Computes quantiles from data in a matrix, given specified probabilities.

Format `y = quantile(x,e)`

Input

<i>x</i>	NxK matrix of data.
<i>e</i>	Lx1 vector, quantile levels or probabilities.

Output

<i>y</i>	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

```
rndseed 345567;
x = rndn(1000,4); /* data */
e = { .025, .5, .975 }; /* quantile levels */
y = quantile(x,e);
```

```
print "medians";
print y[2,.];
print;
print "95 percentiles";
print y[1,.];
print y[3,.];

produces:
medians
-0.0020      -0.0408      -0.0380      -0.0247
95 percentiles
-1.8677      -1.9894      -2.1474      -1.8747
```

quantile			
1.9687	2.0899	1.8576	2.0545

Source quantile.src

- a
- b
- c
- d
- e
- f
- g
- h
- i
- j
- k
- l
- m
- n
- o
- p
- q**
- r
- s
- t
- u
- v
- w
- x y z

quantiled

quantiled

Purpose	Computes quantiles from data in a data set, given specified probabilities.		
Format	$y = \text{quantiled}(\text{dataset}, e, \text{var});$		
Input	<i>dataset</i>	string, data set name, or NxM matrix of data.	
	<i>e</i>	Lx1 vector, quantile levels or probabilities.	
	<i>var</i>	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 <i>dataset</i> is a matrix <i>var</i> cannot be a character vector.	
Output	<i>y</i>	LxK matrix, quantiles.	
Remarks	quantiled will not succeed if $N * \text{minc}(e)$ is less than 1, or $N * \text{maxc}(e)$ 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	$y = \text{quantiled}(\text{"tobit"}, e, 0);$ 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		

quantiled

	-1.1198	1.0000	-1.8139	-2.3143
	2.3066	1.0000	1.4590	1.6954
Source	quantile.src			

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

qyr

qyr

Purpose Computes the orthogonal-triangular (QR) decomposition of a matrix X and returns QY and R .

Format $\{ qy, r \} = \text{qyr}(y, x);$

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**.

Example

```
x = { 1 11, 7 3, 2 1 };
y = { 2 6, 5 10, 4 3 };
{ qy, r } = qyr(y, x);
```

```
          4.6288991  9.0506281
qy =    -3.6692823 -7.8788202
          3.1795692  1.0051489
```

```
          -7.3484692 -4.6268140
r =      0.0000000  10.468648
```

Source qyr.src

See also qqr, qyre, qyrep, olsqr

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qyre

qyre

Purpose Computes the orthogonal-triangular (QR) decomposition of a matrix X and returns QY and R .

Format $\{ qy, r, e \} = \text{qyre}(y, x);$

Input

y	$N \times L$ matrix.
x	$N \times P$ matrix.

Output

qy	$N \times L$ unitary matrix.
r	$K \times P$ upper triangular matrix, $K = \min(N, P)$.
e	$P \times 1$ 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

$$X[., E] = Q_1 R$$

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 $N \times L$ matrix, which will be a much smaller matrix.

If either $Q'Y$ or $Q_i'Y$ are required, see **qtyre**.

If $N < P$ the factorization assumes the form:

$$Q'X[., E] = \begin{bmatrix} R_1 & R_2 \end{bmatrix}$$

where R_1 is a $P \times P$ upper triangular matrix and R_2 is $P \times (N - P)$. Thus Q is a $P \times P$ matrix and R is a $P \times N$ matrix containing R_1 and R_2 .

Example

```
x = { 1 11, 7 3, 2 1 };
y = { 2 6, 5 10, 4 3 };
{ qy, r, e } = qyre(y,x);
```

```
      -0.5942276  -3.0456088
qy =  -6.2442636  -11.647846
      2.3782485  -0.22790230
```

```
r =  -11.445523  -2.9705938
      0.0000000  -6.7212776
```

```
e =  2.0000000
      1.0000000
```

Source

qyr.src

See also

qqr, qre, qyr

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

qyrep

qyrep

Purpose Computes the orthogonal-triangular (QR) decomposition of a matrix X using a pivot vector and returns QY and R .

Format $\{ qy, r, e \} = \text{qyrep}(y, x, pvt);$

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

$$X[., E] = Q_1 R$$

is the QR decomposition of $X[., E]$.

qyrep

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_I 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 $N \times L$ matrix, which will be a much smaller matrix.

If either $Q'Y$ or $Q_I'Y$ are required, see **qtyrep**.

If $N < P$ the factorization assumes the form:

$$Q'X[:, E] = \begin{bmatrix} R_1 & R_2 \end{bmatrix}$$

where R_1 is a $P \times P$ upper triangular matrix and R_2 is $P \times (N - P)$. Thus Q is a $P \times P$ matrix and R is a $P \times N$ matrix containing R_1 and R_2 .

Source `qyr.src`

See also `qr`, `qqrep`, `qrep`, `qtyrep`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rank

Purpose	Computes the rank of a matrix, using the singular value decomposition.	
Format	$k = \text{rank}(x);$	
Input	x	NxP matrix.
Global Input	<code>_svd_tol</code>	global 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	<code>_svd_err</code>	global scalar, if not all of the singular values can be computed <code>_svd_err</code> will be nonzero.
Source	<code>svd.src</code>	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rankindx

rankindx

Purpose	Returns the vector of ranks of a vector.			
Format	$y = \text{rankindx}(x, flag);$			
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	let $x = 12 \ 4 \ 15 \ 7 \ 8;$			
	$r = \text{rankindx}(x, 1);$			
		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(fl,r);`

Input

<i>fl</i>	scalar, file handle of an open file.
<i>r</i>	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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

real

real

Purpose Returns the real part of a matrix.

Format `zr = real(x);`

Input *x* NxK matrix.

Output *zr* NxK matrix, the real part of *x*.

Remarks If *x* is not complex, *zr* will be equal to *x*.

Example

```
x = { 1 11,
      7i 3,
      2+i 1 };
zr = real(x);

      1.0000000 11.0000000
zr =  0.0000000  3.0000000
      2.0000000  1.0000000
```

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

<i>x</i>	Nx1 vector to be recoded (changed).
<i>e</i>	NxK matrix of 1's and 0's.
<i>v</i>	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 K^{th} column of *e* is 1, the K^{th} 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 };
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);
e = e1~e2~e3~e4;
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

recode

```
v = { 1,
      2,
      3,
      4 };

y = recode(x,e,v);

      20
      45
x = 32
      63
      29

      0 0 0 0
      0 0 1 0
e = 0 1 0 0
      0 0 0 0
      1 0 0 0

      1
v = 2
      3
      4

      20
      3
y = 2
      63
      1
```

Source datatran.src

See also code, substute

recode (dataloop)

recode (dataloop)

Purpose	Changes the value of a variable with different values based on a set of logical expressions.
Format	<pre> recode [#] [\$] <i>var</i> with <i>val_1</i> for <i>expression_1</i>, <i>val_2</i> for <i>expression_2</i>, . . . <i>val_n</i> for <i>expression_n</i>; </pre>
Input	<i>var</i> literal, the new variable name. <i>val</i> scalar, value to be used if corresponding expression is true. <i>expression</i> logical scalar-returning expression that returns nonzero <i>TRUE</i> or zero <i>FALSE</i>.
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 <i>TRUE</i> for a given row (observation). If none of the expressions is <i>TRUE</i> 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 externs, or as the result of a previous make, vector, or code statement.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

recode (dataloop)**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

recserrar

Purpose Computes a vector of autoregressive recursive series.

Format `y = recserrar(x,y0,a);`

Input

<i>x</i>	NxK matrix
<i>y0</i>	PxK matrix.
<i>a</i>	PxK matrix.

Output *y* NxK matrix containing the series.

Remarks **recserrar** 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 *P*th order “autoregressive” recursion, with weights given by *a*, and then by adding the result to the corresponding elements of *x*. That is, the *t*th row of *y* is given by:

$$y[t, \cdot] = x[t, \cdot] + a[1, \cdot] * y[t - 1, \cdot] + \dots + a[P, \cdot] * y[t - P, \cdot], t = P + 1, \dots, N$$

and

$$y[t, \cdot] = y0[t, \cdot], t = 1, \dots, P$$

Note that the first *P* rows of *x* are not used.

Example

```
n = 10;
fn multnorm(n,sigma) =
    rndn(n,rows(sigma))*chol(sigma);
let sig[2,2] = { 1 -.3, -.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;
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

recserar

```
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:

$$y[1,t] = \text{rho}[1,1]*y[1,t-1] + x[t,]*b + e[1,t]$$
$$y[2,t] = \text{rho}[2,1]*y[2,t-1] + x[t,]*b + e[2,t]$$

The error terms (**e[1,t]** and **e[2,t]**) are not individually serially correlated, but they are contemporaneously correlated with each other. The variance-covariance matrix is **sig**.

See also **recsercp, recserrc**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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 $y = \text{recsercp}(x,z);$

Input

x	$N \times K$ or $1 \times K$ matrix
z	$N \times K$ or $1 \times K$ matrix.

Output y $N \times K$ matrix in which each column is a series generated by a recursion of the form:

$$y(1) = x(1) + z(1)$$

$$y(t) = y(t-1) \times x(t) + z(t), t = 2, \dots, N$$

Remarks The following GAUSS code could be used to emulate **recsercp** when the number of rows in x and z is the same:

```
n = rows(x); /* assume here that rows(z) */
               /* is also n */
```

```
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

recsercp(x, θ) will produce the cumulative products of the elements in x .

Example

```
c1 = c[1, .];  
n = rows(c) - 1;  
y = recsercp(x, trim(c ./ c1, 1, 0));  
p = c1 .* y[n, .];
```

If $\mathbf{x}$ is a scalar and $\mathbf{c}$ is an $(N+1) \times 1$ vector, the result $\mathbf{p}$ will contain the value of the polynomial whose coefficients are given in $\mathbf{c}$. That is:

$$p = c[1, .] \times x^n + c[2, .] \times x^{(n-1)} + \dots + c[n+1, .]$$

Note that both $\mathbf{x}$ and $\mathbf{c}$ 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 $\mathbf{x} = 2$, and if $\mathbf{c}$ contains the digits of the binary representation of a number, then $\mathbf{p}$ will be the decimal representation of that number.

See also

recserar, recserrc

recserrc

Purpose Computes a recursive series involving division.

Format $y = \text{recserrc}(x,z);$

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:

$$\begin{aligned} y[1] &= x \bmod z[1], \quad x = \text{trunc}(x / z[1]) \\ y[2] &= x \bmod z[2], \quad x = \text{trunc}(x / z[2]) \\ y[3] &= x \bmod z[3], \quad x = \text{trunc}(x / z[3]) \\ &\dots \\ y[n] &= x \bmod z[n] \end{aligned}$$

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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rerun

rerun

Purpose	Displays the most recently created graphics file.
Library	pgraph
Format	rerun;
Portability	DOS only rerun invokes the graphics utility pqgrun.exe.
Remarks	rerun is used by the endwind function.
Source	pcart.src
Globals	_pcmdlin, _pnotify, _psilent, _ptek, _pzoom

reshape

Purpose Reshapes a matrix.

Format $y = \text{reshape}(x, r, c);$

Input

x	NxK matrix.
r	scalar, new row dimension.
c	scalar, new column dimension.

Output y RxC 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 $y = \text{reshape}(x, 2, 6);$

If $x = \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$ then $y = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{matrix}$

If $x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$ then $y = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 & 2 & 3 \end{matrix}$

If $x = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{matrix}$ then $y = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{matrix}$

If $x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$ then $y = \begin{matrix} 1 & 2 & 3 & 4 & 1 & 2 \\ 3 & 4 & 1 & 2 & 3 & 4 \end{matrix}$

If $x = 1$ then $y = \begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$

reshape

See also submat, vec

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

retp

Purpose Returns from a procedure or keyword.

Format `retp;`
`retp(x,y,...);`

Remarks For more details, see “Procedures and Keywords” in the *User’s Guide*.
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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

return

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`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

rev

Purpose Reverses the order of the rows in a matrix.

Format $y = \text{rev}(x);$

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 $x = \text{round}(\text{rndn}(5,3)*10);$
 $y = \text{rev}(x);$

```

      10  7  8
      7  4 -9
x =  -11  0 -3
      3 18  0
      9 -1 20
      9 -1 20
      3 18  0
y =  -11  0 -3
      7  4 -9
      10  7  8

```

See also `sortc`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rfft

rfft

Purpose Computes a real 1- or 2-D Fast Fourier transform.

Format $y = \text{rfft}(x);$

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.

Example $x = \{ 6 \ 9, \ 8 \ 1 \};$

$y = \text{rfft}(x);$

$y = \begin{matrix} 6.0000000 & 1.0000000 \\ 1.5000000 & -2.5000000 \end{matrix}$

See also `rfffti`, `fft`, `ffti`, `fftm`, `fftm_i`

rfffti

Purpose	Computes inverse real 1- or 2-D Fast Fourier transform.		
Format	$y = \text{rfffti}(x);$		
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 fffti .		
Example	$x = \{ 6 \ 1, \ 1.5 \ -2.5 \};$ $y = \text{rfffti}(x);$ $y = \begin{matrix} 6.0000000 & 9.0000000 \\ 8.0000000 & 1.0000000 \end{matrix}$		
See also	rfft, fft, ffti, fftm, fftmi		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rffftip

rffftip

Purpose Computes an inverse real 1- or 2-D FFT. Takes a packed format FFT as input.

Format $y = \text{rffftip}(x);$

Input x NxK matrix or K-length vector.

Output y LxM real matrix or M-length vector.

Remarks **rffftip** assumes that its input is of the same form as that output by **rffftip** and **rffftnp**.

rffftip 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, **rffftip** can handle any matrix whose dimensions can be expressed as:

$$2^p \times 3^q \times 5^r \times 7^s, \quad p, q, r \geq 0 \\ s = 0 \text{ or } 1$$

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 **rffftip** 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 **fffti**. Note, however, that **fffti** expects a full FFT, including negative frequency information, for input.

Do not pass **rffftip** the output from **rfft** or **rfftn**— it will return incorrect results. Use **rfffti** with those routines.

rffftip

See also `fft`, `ffti`, `fftm`, `fftmf`, `fftn`, `rfft`, `rffti`, `rfftn`, `rfftnf`, `rfftf`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rfftn

rfftn

Purpose Computes a real 1- or 2-D FFT.

Format $y = \text{rfftn}(x);$

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:

$$2^p \times 3^q \times 5^r \times 7^s, \quad \begin{array}{l} p, q, r \geq 0 \text{ for rows of matrix} \\ p > 0, q, r \geq 0 \text{ for columns of matrix} \\ p > 0, q, r \geq 0 \text{ for length of vector} \\ s = 0 \text{ or } 1 \text{ for all dimensions} \end{array}$$

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, $2^6 \times 3 \times 5^2 \times 7$, 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.

rfftn

(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`, `fftm_i`, `fftn`, `rfft`, `rffti`, `rfftip`, `rfftnp`, `rfftp`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rfftnp**rfftnp**

Purpose Computes a real 1- or 2-D FFT. Returns the results in a packed format.

Format $y = \text{rfftnp}(x);$

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:

$$2^p \times 3^q \times 5^r \times 7^s, \quad \begin{array}{l} p, q, r \geq 0 \text{ for rows of matrix} \\ p > 0, q, r \geq 0 \text{ for columns of matrix} \\ p > 0, q, r \geq 0 \text{ for length of vector} \\ s = 0 \text{ or } 1 \text{ for all dimensions} \end{array}$$

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

rfftnp

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, $2^6 \times 3 \times 5^2 \times 7$, 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**, **fftmf**, **fftn**, **rfft**, **rffti**, **rfftip**, **rfftn**, **rfftp**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rfftp

rfftp

Purpose Computes a real 1- or 2-D FFT. Returns the results in a packed format.

Format $y = \text{rfftp}(x);$

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**, **fft1**, **fftm**, **fftm1**, **fftn**, **rfft**, **rfft1**, **rfftip**, **rfftn**, **rfftnp**

rndbeta

Purpose Computes pseudo-random numbers with beta distribution.

Format $x = \text{rndbeta}(r, c, a, b);$

Input

- r scalar, number of rows of resulting matrix.
- c scalar, number of columns of resulting matrix.
- a MxN matrix, ExE conformable with RxC resulting matrix, shape parameters for beta distribution.
- b KxL matrix, ExE conformable with RxC resulting matrix, shape parameters for beta distribution.

Output x RxC matrix, beta distributed pseudo-random numbers.

Remarks The properties of the pseudo-random numbers in x are:

$$\begin{aligned}
 E(x) &= a / (a + b) \\
 Var(x) &= a \times b / (a + b + 1) \times (a + b)^2 \\
 x &> 0 \\
 x &< 1 \\
 a &> 0 \\
 b &> 0
 \end{aligned}$$

Source random.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

`rndcon, ..mod, ..mult, ..seed`

`rndcon, ..mod, ..mult, ..seed`

Purpose Resets the parameters of the uniform and Normal random number generators.

Format `rndcon c;`
`rndmod m;`
`rndmult a;`
`rndseed seed;`

Portability Windows, UNIX, OS/2
Parameter default values and ranges:

seed	time(0),	$0 < \text{seed} < 2^{32}$
a	1664525	$0 < a < 2^{32}$
c	1013904223	$0 \leq c < 2^{32}$
m	2^{32}	cannot be changed

DOS

seed	(system clock)
a	397204094
c	0
m	$2^{31}-1$

Parameters must be integers in the range $0 < p < 2^{31}$.

Remarks A linear congruential uniform random number generator is used by **rndu**, and is also called by **rndn**. These statements allow the parameters of this generator to be changed.

The procedure used to generate the uniform random numbers is as follows. First, the current “seed” is used to generate a new seed:

$$\text{new\_seed} = (a * \text{seed} + c) \% m$$

(where % is the mod operator). Then a number between 0 and 1 is created by dividing the new seed by *m*:

$$x = \text{new\_seed} / m$$

rndcon resets *c*.

rndmod resets *m*.

`rndcon, ..mod, ..mult, ..seed`

`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, 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, and so that should not usually be a problem.

The parameters set by these commands remain in effect until new commands are encountered, or until GAUSS is restarted.

See also `rndu, rndn, rndus, rndns`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rndgam

rndgam

Purpose Computes pseudo-random numbers with gamma distribution.

Format $x = \text{rndgam}(r,c,alpha);$

Input

r scalar, number of rows of resulting matrix.

c scalar, number of columns of resulting matrix.

$alpha$ MxN matrix, ExE conformable with RxC resulting matrix, shape parameters for gamma distribution.

Output x RxC matrix, gamma distributed pseudo-random numbers.

Remarks The properties of the pseudo-random numbers in x are:

$$\begin{aligned} E(x) &= alpha \\ Var(x) &= alpha \\ x &> 0 \\ alpha &> 0 \end{aligned}$$

To generate **gamma** ($alpha, theta$) pseudo-random numbers where $theta$ is a scale parameter, multiply the result of **rndgam** by $theta$. Thus:

$$z = theta * \text{rndgam}(1,1,alpha)$$

has the properties

$$\begin{aligned} E(z) &= alpha \times theta \\ Var(z) &= alpha \times theta^2 \\ z &> 0 \\ alpha &> 0 \\ theta &> 0 \end{aligned}$$

Source random.src

rndn

Purpose	Creates a matrix of standard Normal (pseudo) random numbers.
Format	$y = \text{rndn}(r, c);$
Input	r scalar, row dimension. c scalar, column dimension.
Output	y RxC matrix of Normal random numbers having a mean of 0 and standard deviation of 1.
Remarks	r and c will be truncated to integers if necessary. The Normal random number generator is based upon the uniform random number generator. To reseed them both, use the rndseed statement or use the alternate function rndns. The other parameters of the uniform generator can be changed using rndcon, rndmod, and rndmult.
Example	<pre>x = rndn(8100,1); m = meanc(x); s = stdc(x); m = 0.002810 s = 0.997087</pre> In this example, a sample of 8100 Normal random numbers is drawn, and the mean and standard deviation are computed for the sample.
See also	rndu , rndns , rndcon
Technical Notes	This function uses the fast acceptance-rejection algorithm proposed by Kinderman, A. J., and J. G. Ramage. "Computer Generation of Normal Random Numbers." <i>Journal of the American Statistical Association</i> . Vol. 71 No. 356, Dec. 1976, 893-96.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rndnb**rndnb**

Purpose Computes pseudo-random numbers with negative binomial distribution.

Format $x = \text{rndnb}(r, c, k, p);$

Input

- r scalar, number of rows of resulting matrix.
- c scalar, number of columns of resulting matrix.
- k MxN matrix, ExE conformable with RxC resulting matrix, “event” parameters for negative binomial distribution.
- p KxL matrix, ExE conformable with RxC resulting matrix, probability parameters for negative binomial distribution.

Output x RxC matrix, negative binomial distributed pseudo-random numbers.

Remarks The properties of the pseudo-random numbers in x are:

$$\begin{aligned}
 E(x) &= k \times p / (1 - p) \\
 Var(x) &= k \times p / (1 - p)^2 \\
 x &= 0, 1, 2, \dots, k \\
 k &> 0 \\
 p &> 0 \\
 p &< 1
 \end{aligned}$$

Source random.src

rndns, rndus

Purpose	An alternate way of generating Normal (rndns) or uniform (rndus) random numbers using a seed value as one of the arguments.
Format	<pre>y = rndns(r,c,s);</pre> <pre>y = rndus(r,c,s);</pre>
Input	<i>r</i> scalar, row dimension. <i>c</i> scalar, column dimension. <i>s</i> scalar, starting seed.
Output	<i>y</i> RxC matrix of numbers satisfying the properties of either the Normal distribution (rndns) or the uniform distribution (rndus).
Portability	Windows, UNIX, OS/2 The value of the seed must be in the range: $0 < \text{seed} < 2^{32}$. DOS The value of the seed must be in the range: $0 < \text{seed} < 2^{31}$.
Remarks	The value of the seed must be in the range: $0 < \text{seed} < 2^{31} - 1$. The seed must be a variable; it cannot be a constant or an expression. It will be updated automatically during the call.
Example	<pre>seed1 = 3937841;</pre> <pre>x = rndns(1000,2,seed1);</pre> With this statement a 1000x2 matrix of Normal random variables will be created using the value in seed1 as the starting seed. seed1 will be updated during the call.
See also	rndn, rndu

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rndp

rndp

Purpose	Computes pseudo-random numbers with Poisson distribution.		
Format	$x = \text{rndp}(r,c,\text{lambda}) ;$		
Input	r	scalar, number of rows of resulting matrix.	
	c	scalar, number of columns of resulting matrix.	
	lambda	MxN matrix, ExE conformable with RxC resulting matrix, shape parameters for Poisson distribution.	
Output	x	RxC matrix, Poisson distributed pseudo-random numbers.	
Remarks	The properties of the pseudo-random numbers in x are:		
	$E x = \text{lambda}$		
	$Var(x) = \text{lambda}$		
	$x = 0, 1, 2, \dots$		
	$\text{lambda} > 0$		
Source	random.src		

rndu

Purpose Creates a matrix of uniform (pseudo) random variables.

Format $y = \text{rndu}(r,c);$

Input

r scalar, row dimension.

c scalar, column dimension.

Output y RxC matrix of uniform random variables between 0 and 1.

Remarks r and c will be truncated to integers if necessary.

This generator is automatically seeded using the clock when GAUSS is first started. However, that can be overridden using the **rndseed** statement or by using **rndus**.

The seed is automatically updated as a random number is generated (see above under **rndcon**). Thus, 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. This is a 32-bit generator, though, so the range is sufficient for most applications.

Example

```
x = rndu(8100,1);
y = meanc(x);
z = stdc(x);
```

$y = 0.500205$

$z = 0.289197$

In this example, a sample of 8100 uniform random numbers is generated, and the mean and standard deviation are computed for the sample.

See also **rndn**, **rndcon**, **rndmod**, **rndmult**, **rndseed**

Technical Notes This function uses a multiplicative-congruential method. This method is discussed in Kennedy, W.J., Jr., and J.E. Gentle. *Statistical Computing*. Marcel Dekker, Inc., NY, 1980, 136-147.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rndvm

rndvm

Purpose	Computes von Mises pseudo-random numbers.		
Format	$x = \text{rndvm}(r, c, m, k);$		
Input	r	scalar, number of rows of resulting matrix.	
	c	scalar, number of columns of resulting matrix.	
	m	NxK matrix, ExE conformable with $r \times c$, means for von Mises distribution.	
	k	LxM matrix, ExE conformable with $r \times c$, shape argument for von Mises distribution.	
Output	x	$r \times c$ matrix, von Mises distributed random numbers.	
Source	random.src		

rotater

Purpose Rotates the rows of a matrix.

Format $y = \text{rotater}(x, r);$

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 $(r \% \text{cols}(x))$.

Example $y = \text{rotater}(x, r);$

If $x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ and $r = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ Then $y = \begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \end{bmatrix}$

If $x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$ and $r = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix}$ Then $y = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 4 & 5 \\ 8 & 9 & 7 \\ 10 & 11 & 12 \end{bmatrix}$

See also `shiftr`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

round

round

Purpose	Rounds to the nearest integer.
Format	$y = \text{round}(x);$
Input	x NxK matrix.
Output	y NxK matrix containing the rounded elements of x .
Example	<pre>let x = { 77.68 -14.10, 4.73 -158.88 }; y = round(x); y = 78 -14 5 -159</pre>
See also	<code>trunc</code> , <code>floor</code> , <code>ceil</code>

rows

Purpose	Returns the number of rows in a matrix.
Format	$y = \text{rows}(x);$
Input	x NxK 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	<pre>x = ones(3,5); y = rows(x); 1 1 1 1 1 x = 1 1 1 1 1 1 1 1 1 1 y = 3</pre>
See also	cols , show

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

rowsf

rowsf

Purpose Returns the number of rows in a GAUSS data set (.dat) file or GAUSS matrix (.fmt) file.

Format $y = \text{rowsf}(f);$

Input f file handle of an open file.

Output y scalar, number of rows in the specified file.

Example open fp = myfile;
 $r = \text{rowsf}(fp);$
 $c = \text{colsf}(fp);$

See also colsf, open, typef

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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

```
let x[3,3] = 1 2 3
              4 5 6
              7 8 9;
```

```
y = rref(x);
```

```
      1 0 -1
y =  0 1  2
      0 0  0
```

Source `rref.src`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

run

run

Purpose Runs a source code or compiled code program.

Format `run filename;`

Input *filename* literal or ^string, name of file to run.

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.

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.prc;` No additional search will be made if the file is not found.

`run myprog.prc;` The directories listed in **src_path** will be searched for `myprog.prc` 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.prc;`

Example 2

```
name = "myprog.prg";  
run ^name;
```

See also `#include`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

save

save

Purpose Saves matrices, strings, or procedures to a disk file.

Format **save** *[[vflag]]* **[[path=path]]** *x*, **[[lpath=]]***y*;

Input

vflag version flag.

-v89 supported on DOS, OS/2, Windows

-v92 supported on UNIX, Windows

-v96 supported on all platforms

See also “File I/O” in the *User’s Guide* 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 **saves**.

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 saved:

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

save

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 a 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 get a backslash, because it is the escape character in quoted strings. It is not required when specifying literals.

```
save path=/procs;
```

Changes 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 **load**, **saveall**, **saved**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

saveall

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, 0 if fail.

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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

savewind

savewind

Purpose	Saves the current graphic panel configuration to a file.
Library	pgraph
Format	<i>err</i> = savewind (<i>filename</i>);
Input	<i>filename</i> Name of file.
Output	<i>err</i> 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 graphic panels in “Publication Quality Graphics” in the <i>User’s Guide</i> .
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.
Library	pgraph
Format	scale (<i>x,y</i>);
Input	<i>x</i> matrix, the X axis data. <i>y</i> matrix, the Y axis data.
Remarks	<i>x</i> and <i>y</i> 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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

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.

Library **pgraph**

Format **scale3d**(*x,y,z*);

Input

<i>x</i>	matrix, the X axis data.
<i>y</i>	matrix, the Y axis data.
<i>z</i>	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, generally the return argument of a function or procedure call.

Output *y* scalar, which is returned as a 0 if its argument is not a scalar error code. If the argument is an error code, then scalerr returns the value of the error code as an integer.

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 which are predefined in GAUSS or an error code which the user has defined using **error**.

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 affected by the trap state:

function	trap 1 error code	trap 0 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

scalerr

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**

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.

The **ismiss** function will test each element of the matrix and return 1 if it encounters any missing values. **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 **ismiss** would return a 1 for complex `x = . + 1i`, 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 that is 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.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

schtoc

schtoc

Purpose	To reduce any 2x2 blocks on the diagonal of the real Schur matrix returned from schur . The transformation matrix is also updated.		
Format	{ <i>schc</i> , <i>transc</i> } = schtoc (<i>sch</i> , <i>trans</i>);		
Input	<i>sch</i>	real NxN matrix in Real Schur form, i.e., upper triangular except for possibly 2x2 blocks on the diagonal.	
	<i>trans</i>	real NxN matrix, the associated transformation matrix.	
Output	<i>schc</i>	NxN matrix, possibly complex, strictly upper triangular. The diagonal entries are the eigenvalues.	
	<i>transc</i>	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 <i>sch</i> 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	{ <i>schc</i> , <i>transc</i> } = 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 $\{ s, z \} = \text{schur}(x)$

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 similarity 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

$$s = z'xz$$

and since z is orthogonal,

$$x = zsz'$$

Example let $x[3,3] = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9; \end{matrix}$
 $\{ s, z \} = \text{schur}(x);$

$s = \begin{matrix} 16.11684397 & 4.89897949 & 0.00000000 \\ -0.00000000 & -1.11684397 & -0.00000000 \\ 0.00000000 & 0.00000000 & 0.00000000 \end{matrix}$

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

schur

$$z = \begin{matrix} & 0.23197069 & 0.88290596 & 0.40824829 \\ 0.52532209 & 0.23952042 & -0.81649658 \\ 0.81867350 & -0.40386512 & 0.40824829 \end{matrix}$$

See also **hess**

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

screen

Purpose Controls output to the window.

Format `screen on;`
`screen off;`
`screen out;`
`screen;`

Portability UNIX

screen on|off affects the active graphic panel. Each graphic panel “remembers” its own setting, even when it is no longer the active graphic panel.

screen out affects the active graphic panel. Not supported for PQG graphic panels.

DOS

screen out will take a snapshot of the window and dump it to the current auxiliary output. If the output is open, the new data will be appended. If the output is closed, it will be opened for append and the window will be written and then it will be closed again. This is handy for dumping a text window image to a file or printer.

It also allows a complex window image, created with **locate** statements, to be sent to a file or printer. This is not a graphics window dump—the window must be in text mode.

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 do **output off** and **screen on**.

screen with no arguments will print “**Screen is on**” or “**Screen is off**” on the console.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

screen

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`

scroll

Purpose	Scrolls a section of the window.
Format	scroll <i>v</i> ;
Input	<i>v</i> 6x1 vector
Portability	UNIX
	scroll scrolls a region of the active graphic panel. Supported only for Text graphic panels.
Remarks	The elements of <i>v</i> are defined as: [1] coordinate of upper left row. [2] coordinate of upper left column. [3] coordinate of lower right row. [4] coordinate of lower right column. [5] number of lines to scroll. [6] value of attribute. This assumes the origin at (1,1) in the upper left just like the locate command. The window will be scrolled the number of lines up or down (positive or negative 5th element) and the value of the 6th element will be used as the attribute as follows: 7 regular text 112 reverse video 0 graphics black If the number of lines (element 5) is 0, the entire graphic panel will be blanked.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

scroll

Example `let v = 1 1 12 80 5 7;`
 `scroll v;`

This call would scroll a graphic panel 80 columns wide covering the upper twelve rows of the window. The graphic panel would be scrolled up 5 lines and the new lines would be displayed in regular text mode.

See also `locate`, `printdos`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

seekr

Purpose Moves the pointer in a `.dat` or `.fmt` file to a particular row.

Format `y = seekr(fh,r);`

Input

<i>fh</i>	scalar, file handle of an open file.
<i>r</i>	scalar, the row number to which the pointer is to be moved.

Output

<i>y</i>	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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

select (dataloop)

select (dataloop)

Purpose	Selects specific rows (observations) in a data loop based on a logical expression.
Format	select <i>logical_expression</i> ;
Remarks	Selects only those rows for which <i>logical_expression</i> is <i>TRUE</i> . 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.
Example	<code>select age > 40 AND sex \$== 'MALE';</code>
See also	<code>delete</code>

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

<i>x</i>	NxK matrix.
<i>e</i>	Nx1 vector of 1's and 0's.

Output *y* MxK matrix 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);`
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**.

Source `datatran.src`

See also `delif`, `scalmiss`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

seqa, seqm

seqa, seqm

Purpose **seqa** creates an additive sequence. **seqm** creates a multiplicative sequence.

Format $y = \text{seqa}(start, inc, n);$
 $y = \text{seqm}(start, inc, n);$

Input *start* scalar specifying the first element.
inc scalar specifying increment.
n scalar specifying the number of elements in the sequence.

Output *y* Nx1 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, ... 10¹⁰.

Example $a = \text{seqa}(2, 2, 10)';$
 $m = \text{seqm}(2, 2, 10)';$

$a = 2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20$

$m = 2\ 4\ 8\ 16\ 32\ 64\ 128\ 256\ 512\ 1024$

Note that the results have been transposed in this example. Both functions return Nx1 (column) vectors.

See also **recserar**, **recsercp**

setdif

Purpose Returns the unique elements in one vector that are not present in a second vector.

Format `y = setdif(v1,v2,flag);`

Input

<i>v1</i>	Nx1 vector.	
<i>v2</i>	Mx1 vector.	
<i>flag</i>	scalar,	if 0, case-sensitive character comparison. if 1, numeric comparison. if 2, case-insensitive character comparison.

Output *y* Lx1 sorted vector containing all unique values that are in *v1* and are not in *v2*, or a scalar missing.

Example

```
let v1 = mary jane linda john;
let v2 = mary sally;
flag = 0;
y = setdif(v1,v2,flag);
```

```
      JANE
y =    JOHN
      LINDA
```

Source setdif.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

setvars

setvars

Purpose	Reads the variable names from a data set header and creates global matrices with the same names.		
Format	<i>nvec</i> = setvars (<i>dataset</i>);		
Input	<i>dataset</i>	string, the name of the GAUSS data set. Do not use a file extension.	
Output	<i>nvec</i>	Nx1 character vector, containing the variable names defined in the data set.	
Remarks	setvars is designed to be used interactively.		
Example	<i>nvec</i> = setvars ("freq");		
Source	vars.src		
See also	makevars		

setvmode

This command is obsolete. See “**setvmode**” on page A-18.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

setwind

setwind

Purpose	Sets the current graphic panel to a previously created graphic panel number.
Library	pgraph
Format	<code>setwind(<i>n</i>);</code>
Input	<i>n</i> 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 “Publication Quality Graphics in the <i>User’s Guide</i>.
Source	pwindow.src
See also	<code>begwind</code> , <code>endwind</code> , <code>getwind</code> , <code>nextwind</code> , <code>makewind</code> , <code>window</code>

shell

Purpose	Executes an operating system command.
Format	<code>shell [[s]];</code>
Input	<i>s</i> literal or ^string, the command to be executed.
Portability	UNIX Control and output go to the controlling terminal, if there is one. This function may be used in terminal mode. OS/2, Windows The shell function opens a new terminal. DOS This is equivalent to the DOS command.
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) ^. You can use > instead of the word shell.
Example	<pre>comstr = "ls ./src"; shell ^comstr;</pre> This lists the contents of the <code>./src</code> subdirectory, then returns to GAUSS. <pre>shell cmp n1.fmt n1.fmt.old;</pre> This compares the matrix file <code>n1.fmt</code> to an older version of itself, <code>n1.fmt.old</code>, to see if it has changed. When cmp finishes, control is returned to GAUSS.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

shell

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**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

shiftr

Purpose Shifts the rows of a matrix.

Format $y = \text{shiftr}(x, s, f);$

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 $y = \text{shiftr}(x, s, f);$

If $x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $s = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ and $f = \begin{bmatrix} 99 \\ 999 \end{bmatrix}$

Then $y = \begin{bmatrix} 99 & 1 \\ 4 & 999 \end{bmatrix}$

If $x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ and $s = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}$ and $f = 0$

Then $y = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 7 \end{bmatrix}$

See also rotater

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

show, lshow

show, lshow

Purpose Displays the global symbol table. The output from **lshow** is sent to the printer.

Format **show** *[-flags]* *[[symbol]]*;
lshow *[-flags]* *[[symbol]]*;

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
- n** no pause

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:

Memory used	Address	Name	Info	Cplx	Type	References
32 bytes at [00081b74]	AREA	1=1			FUNCTION	local refs
32 bytes at [00081a14]	dotfeq	1=2			PROCEDURE	global refs
1144 bytes at [0007f1b4]	indices2	4=3			PROCEDURE	local refs
144 bytes at [0007f874]	X	3,3	C		MATRIX	
352 bytes at [0007f6ec]	_IXCAT	44,1			MATRIX	
8 bytes at [0007f6dc]	_olsrnam	7 char			STRING	

32000 bytes program space, 0% used
4336375 bytes workspace, 4325655 bytes free
53 global symbols, 1500 maximum, 6 shown

show, lshow

The “Memory used” column is the amount of memory used by the item.

The “Address” column is the address where the item is stored (hexadecimal format). This will change as matrices and strings change in size.

The “Name” column is the name of the symbol.

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, then the Info column gives the number of rows and columns. If the symbol is a string, then the Info column gives the number of characters in the string. As follows:

Rets=Args if procedure or function

Row,Col if matrix

Length if string

The “Cplx” column contains a “C” if the symbol is a complex matrix.

The “Type” column specifies the symbol table type of the symbol. It can be **function**, **keyword**, **matrix**, **procedure**, or **string**.

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.

The program space is the area of space reserved for all nonprocedure, nonfunction program code. It can be changed in size with the **new** command. The workspace is the memory used to store matrices, strings, procedures, and functions.

Example `show /fpg eig*;`

This command will show all functions and procedures that have global references and begin with **eig**.

`show /mn;`

This command will show all matrices without pausing when the window is full.

See also `new, delete`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sin

sin

Purpose	Returns the sine of its argument.
Format	$y = \sin(x);$
Input	x NxK matrix.
Output	y NxK matrix containing sine of x .
Remarks	For real matrices, x should contain angles measured in radians. To convert degrees to radians, multiply the degrees by $\frac{\pi}{180}$.
Example	<pre>let x = { 0, .5, 1, 1.5 }; y = sin(x); 0.00000000 y = 0.47942554 0.84147098 0.99749499</pre>
See also	atan, cos, sinh, pi

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);`

`-1.570796`

`-0.785398`

`x = 0.000000`

`0.785398`

`1.570796`

`3.141593`

`-2.301299`

`-0.868671`

`y = 0.000000`

`0.868671`

`2.301299`

`11.548739`

Source `trig.src`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sleep

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. The DOS version always sleeps the full number of seconds, so it always returns 0.

A program may sleep for longer than *secs* seconds, due to system scheduling.

solpd

Purpose Solves a set of positive definite linear equations.

Format $x = \text{solpd}(b,A);$

Input

b	NxK matrix.
A	NxN symmetric positive definite matrix.

Output x NxK matrix, 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 $\text{inv}(A)*b$.

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);

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

solpd

Example

```
n = 5; format 20,8;
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
-0.36334089    -0.36334089    0.000000000
 0.19683330     0.19683330    8.32667268E-017
 0.99361330     0.99361330    2.22044605E-016
-1.84167681    -1.84167681    0.000000000
-0.88455829    -0.88455829    1.11022302E-016
```

See also scalerr, 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 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 $-\infty$.

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);`

```

      4 7 3
x =  1 3 2
      3 4 8
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sortc, sortcc

$$y = \begin{matrix} & 1 & 3 & 2 \\ 3 & 4 & 8 \\ 4 & 7 & 3 \end{matrix}$$

See also **rev**

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

sortd

Purpose Sorts a data file on disk with respect to a specified variable.

Format `sortd(infile,outfile,keyvar,keytyp);`

Input

<i>infile</i>	string, name of input file.
<i>outfile</i>	string, name of output file, must be different.
<i>keyvar</i>	string, name of key variable.
<i>keytyp</i>	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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sorthc, sorthcc

Purpose Sorts a matrix of numeric or character data.

Format $y = \text{sorthc}(x, c);$
 $y = \text{sorthcc}(x, c);$

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.

sorthc assumes that the column to sort on is numeric. **sorthcc** 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 $-\infty$.

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 $\text{let } x[3, 3] = \begin{matrix} 4 & 7 & 3 \\ 1 & 3 & 2 \\ 3 & 4 & 8; \end{matrix}$
 $y = \text{sorthc}(x, 1);$

$x = \begin{matrix} 4 & 7 & 3 \\ 1 & 3 & 2 \\ 3 & 4 & 8 \end{matrix}$

sorthc, sorthcc

```
      1 3 2
y =   3 4 8
      4 7 3
```

See also **sortc, rev**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sortind, sortindc

Purpose Returns the sorted index of x .

Format $ind = \text{sortind}(x);$
 $ind = \text{sortindc}(x);$

Input x NX1 column vector.

Output ind NX1 vector representing sorted index of x .

Remarks **sortind** assumes x contains numeric data. **sortindc** assumes 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 $\text{let } x = 5 \ 4 \ 4 \ 3 \ 3 \ 2 \ 1;$
 $ind = \text{sortind}(x);$
 $y = x[ind];$

sortmc

Purpose	Sorts a matrix on multiple columns.	
Format	$y = \text{sortmc}(x, v);$	
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.
Remarks	The function works recursively and the number of sort columns is limited by the available workspace.	
Source	sortmc.src	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sparseCols

sparseCols

Purpose	Returns the number of columns in a sparse matrix.
Format	$c = \text{sparseCols}(x);$
Input	x MxN sparse matrix.
Output	c scalar, number of columns.
Source	sparse.src

sparseEye

Purpose Returns a sparse identity matrix.

Format $y = \text{sparseEye}(n);$

Input n scalar, order of identity matrix.

Output y NxN sparse identity matrix.

Example $y = \text{sparseEye}(3);$

$d = \text{denseSubmat}(y, 0, 0);$

$$d = \begin{bmatrix} 1.0000000 & 0.0000000 & 0.0000000 \\ 0.0000000 & 1.0000000 & 0.0000000 \\ 0.0000000 & 0.0000000 & 1.0000000 \end{bmatrix}$$

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sparseFD

sparseFD

Purpose	Converts dense matrix gto sparse matrix.
Format	$y = \text{sparseFD}(x, \textit{eps}) ;$
Input	x MxN dense matrix. $\textit{eps}$ scalar, elements of x less than $\textit{eps}$ will be treated as zero.
Output	y MxN sparse matrix.
Remarks	A dense matrix is just a normal format matrix.
Source	<code>sparse.src</code>

sparseFP

Purpose Converts packed matrix to sparse matrix.

Format $y = \text{sparseFP}(x, r, c);$

Input

x	Mx3 packed matrix, see remarks for format.
r	scalar, rows of output matrix.
c	scalar, columns of output matrix.

Output y RxC sparse matrix.

Remarks x contains the nonzero elements of the sparse matrix. The first column of x contains the element value, the second column the row number, and the third column the column number.

Source sparse.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sparseHConcat

sparseHConcat

Purpose	Horizontally concatenates two sparse matrices.		
Format	$z = \text{sparseHConcat}(y,x);$		
Input	y	MxN sparse matrix, left hand matrix.	
	x	MxL sparse matrix, right hand matrix.	
Output	z	Mx(N+L) sparse matrix.	
Source	sparse.src		

sparseNZE

Purpose Returns the number of nonzero elements in a sparse matrix.

Format $r = \text{sparseNZE}(x);$

Input x MxN sparse matrix.

Output r scalar, number of nonzero elements in x .

Source sparse.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sparse0nes

sparse0nes

Purpose	Generates sparse matrix of ones and zeros		
Format	$y = \text{sparse0nes}(x,r,c);$		
Input	x	Mx2 matrix, first column contains row numbers of the ones, and the second column contains column numbers.	
	r	scalar, rows of full matrix.	
	c	scalar, columns of full matrix.	
Output	y	sparse matrix of ones.	
Source	sparse.src		

sparseRows

Purpose Returns the number of rows in a sparse matrix.

Format $r = \text{sparseRows}(x);$

Input x MxN sparse matrix.

Output r scalar, number of rows.

Source sparse.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sparseSet

sparseSet

Purpose	Resets sparse library global matrices to default values.
Format	<code>sparseSet;</code>
Globals	<code>_sparse_ARnorm, _sparse_Acond, _sparse_Anorm,</code> <code>_sparse_Atol, _sparse_Btol, _sparse_CondLimit,</code> <code>_sparse_Damping, _sparse_NumIters,</code> <code>_sparse_RetCode, _sparse_Rnorm, _sparse_Xnorm</code>
Source	<code>sparse.src</code>

sparseSolve

Purpose Solves $Ax = B$ for x when A is a sparse matrix.

Format $x = \text{sparseSolve}(A,B);$

Input A MxN sparse matrix.
 B Nx1 vector.

Global Input **_sparse_Damping** scalar, if nonzero, damping coefficient for damped least squares solve, i.e.,

$$\begin{bmatrix} A \\ dI \end{bmatrix} = \begin{bmatrix} B \\ 0 \end{bmatrix}$$

is solved for X where $d = \text{\_sparse\_Damping}$, I is a conformable identity matrix, and 0 a conformable matrix of zeros.

_sparse_Atol scalar, an estimate of the relative error in A . If zero, **_sparse_Atol** is assumed to be machine precision. Default = 0.

_sparse_Btol an estimate of the relative error in B . If zero, **_sparse_Btol** is assumed to be machine precision. Default = 0.

_sparse_CondLimit upper limit on condition of A . Iterations will be terminated if a computed estimate of the condition of A exceeds **_sparse_CondLimit**. If zero, set to 1 / machine precision.

_sparse_NumIters maximum number of iterations.

Output x Nx1 vector, solution of $Ax = B$.

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

sparseSolve

**Global
Output**

_sparse_RetCode

- scalar, termination condition.
- 0 x is the exact solution, no iterations performed.
- 1 solution is nearly exact with accuracy on the order of **_sparse_Atol** and **_sparse_Btol**.
- 2 solution is not exact and a least squares solution has been found with accuracy on the order of **_sparse_Atol**.
- 3 the estimate of the condition of A has exceeded **_sparse_CondLimit**. The system appears to be ill-conditioned.
- 4 solution is nearly exact with reasonable accuracy.
- 5 solution is not exact and a least squares solution has been found with reasonable accuracy.
- 6 iterations halted due to poor condition given machine precision.
- 7 **_sparse_NumIters** exceeded.

_sparse_Anorm

scalar, estimate of Frobenius norm of

$$\begin{bmatrix} A \\ dI \end{bmatrix}$$

_sparse_Acond

estimate of condition of A .

_sparse_Rnorm

estimate of norm of

$$\begin{bmatrix} A \\ dI \end{bmatrix}_x - \begin{bmatrix} B \\ 0 \end{bmatrix}$$

_sparse_ARnorm

estimate of norm of

$$\begin{bmatrix} A \\ dI \end{bmatrix}' \begin{bmatrix} A \\ dI \end{bmatrix}$$

_sparse_XAnorm

estimate of norm of x .

Source

`sparse.src`

sparseSubmat

Purpose Returns (sparse) submatrix of sparse matrix.

Format $e = \text{sparseSubmat}(x, r, c);$

Input

x	MxN sparse matrix.
r	Kx1 vector, row indices.
c	Lx1 vector, column indices.

Output

e	KxL sparse matrix.

Remarks If r or c are scalar zeros, all rows or columns will be returned.

Source sparse.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sparseTD

sparseTD

Purpose Multiplies sparse matrix by dense matrix.

Format $z = \text{sparseTD}(s,d);$

Input s MxN sparse matrix.
 d NxL dense matrix.

Output z MxL dense matrix, the result of $s \times d$.

Source sparse.src

sparseTrTD

Purpose Multiplies sparse matrix transposed by dense matrix.

Format $z = \text{sparseTrTD}(s,d);$

Input s NxM sparse matrix.
 d NxL dense matrix.

Output z MxL dense matrix, the result of $s'd$.

Source sparse.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sparseVConcat

sparseVConcat

Purpose	Vertically concatenates two sparse matrices.		
Format	$z = \text{sparseVConcat}(y,x);$		
Input	y	MxN sparse matrix, top matrix.	
	x	LxN sparse matrix, bottom matrix.	
Output	z	(M+L)xN sparse matrix.	
Source	sparse.src		

spline

Purpose Computes a two-dimensional interpolatory spline.

Format `{ u,v,w } = spline(x,y,z,sigma,g);`

Input

<i>x</i>	1xK vector, x-abscissae (x-axis values).
<i>y</i>	Nx1 vector, y-abscissae (y-axis values).
<i>z</i>	KxN matrix, ordinates (z-axis values).
<i>sigma</i>	scalar, tension factor.
<i>g</i>	scalar, grid size factor.

Output

<i>u</i>	1xK*G vector, x-abscissae, regularly spaced.
<i>v</i>	N*Gx1 vector, y-abscissae, regularly spaced.
<i>w</i>	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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sqpSolve

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.

sqpSolve**_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]*[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:

$$\_sqp\_C * X \geq \_sqp\_D$$

where X is the Kx1 unknown parameter vector.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sqpSolve**_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:

$$P[1] * P[2] \geq P[3]$$

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
```


sqpSolve**_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 satisfied **sqpSolve** will exit the iterations.

_sqp_ParNames

Kx1 character vector, parameter names.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sqpSolve

_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: <table> <tr> <td>“lineq”</td><td>Lagrangeans of linear equality constraints</td></tr> <tr> <td>“nlineq”</td><td>Lagrangeans of nonlinear equality constraints</td></tr> <tr> <td>“linineq”</td><td>Lagrangeans of linear inequality constraints</td></tr> <tr> <td>“nlinineq”</td><td>Lagrangeans of nonlinear inequality constraints</td></tr> <tr> <td>“bounds”</td><td>Lagrangeans of bounds</td></tr> </table> Whenever a constraint is active, its associated Lagrangean will be nonzero.	“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						
“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																
$retcode$	return code: <table> <tr> <td>0</td><td>normal convergence</td></tr> <tr> <td>1</td><td>forced exit</td></tr> <tr> <td>2</td><td>maximum number of iterations exceeded</td></tr> <tr> <td>3</td><td>function calculation failed</td></tr> <tr> <td>4</td><td>gradient calculation failed</td></tr> <tr> <td>5</td><td>Hessian calculation failed</td></tr> <tr> <td>6</td><td>line search failed</td></tr> <tr> <td>7</td><td>error with constraints</td></tr> </table>	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
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																

sqpSolve

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 `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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sqrt

sqrt

Purpose Computes the square root of every element in a matrix.

Format `y = sqrt(x);`

Input *x* NxK matrix.

Output *y* NxK matrix, the square roots of each element of *x*.

Remarks 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, **sqrt** will generate an error for negative inputs.

If *x* is already complex, the complex number state doesn't matter; **sqrt** will compute a complex result.

Example

```
let x[2,2] = 1 2 3 4;
y = sqrt(x);

x = 1.00000000 2.00000000
    3.00000000 4.00000000

y = 1.00000000 1.41421356
    1.73205081 2.00000000
```

stdc

Purpose Computes the standard deviation of the elements in each column of a matrix.

Format $y = \text{stdc}(x);$

Input x NxK matrix.

Output y Kx1 vector, the standard deviation of each column of x .

Remarks This function essentially computes:

$$\text{sqrt}(1/(N-1) * \text{sumc}((x - \text{meanc}(x))')^2))$$

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

$$\text{sqrt}((N-1)/N)$$

Example $y = \text{rndn}(8100,1);$

$\text{std} = \text{stdc}(y);$

$\text{std} = 1.008377$

In this example, 8100 standard Normal random variables are generated, and their standard deviation is computed.

See also `meanc`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

stocv

stocv

Purpose Converts a string to a character vector.

Format $v = \text{stocv}(s);$

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 $s = \text{"Now is the time for all good men"};$

$v = \text{stocv}(s);$

$v =$
 "Now is t"
 "he time "
 "for all "
 "good men"

See also **cvtos, vget, vlist, vput, vread**

stof

Purpose	Converts a string to floating point.	
Format	$y = \text{stof}(x);$	
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	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

stop

stop

Purpose	Stops a program and returns to the command prompt. Does not close files.
Format	stop ;
Portability	UNIX
	stop does not close any graphic panels. If you want to close a program's graphic panels when it is done, use end .
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

strindx

Purpose	Finds the index of one string within another string.
Format	<code>y = strindx(<i>where</i>,<i>what</i>,<i>start</i>);</code>
Input	<i>where</i> string or scalar, the data to be searched. <i>what</i> string or scalar, the substring to be searched for in <i>where</i>. <i>start</i> scalar, the starting point of the search in <i>where</i> for an occurrence of <i>what</i>. The index of the first character in a string is 1.
Output	<i>y</i> scalar containing the index of the first occurrence of <i>what</i> , within <i>where</i> , which is greater than or equal to <i>start</i> . 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: <pre>z = "Whatchmacallit"; x = "call"; y = strindx(z,x,1);</pre> <i>y</i> = 9 This function is used with strsect for extracting substrings.
See also	strrindx , strlen , strsect , strput

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

strlen

strlen

Purpose	Returns the length of a string.
Format	$y = \text{strlen}(x);$
Input	x string or NxK matrix of character data.
Output	y scalar containing the exact length of the string x or NxK matrix of the lengths of the elements in the matrix 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	$x = \text{"How long?";}$ $y = \text{strlen}(x);$ $y = 9$
See also	<code>strsect</code> , <code>strindx</code> , <code>strrindx</code>

strput

Purpose	Lays a substring over a string.
Format	<code>y = strput(<i>substr</i>,<i>str</i>,<i>off</i>);</code>
Input	<i>substr</i> string, the substring to be laid over the other string. <i>str</i> string, the string to receive the substring. <i>off</i> scalar, the offset in <i>str</i> to place <i>substr</i> . The offset of the first byte is 1.
Output	<i>y</i> string, the new string.
Example	<pre>str = "max"; sub = "imum"; f = 4; y = strput(sub, str, f); print y;</pre> Produces: maximum
Source	strput.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

strrindx

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));
else;
    name = "";
endif;
```

```
pos = 11
name = "ols.src"
```

strrindx can be used with **strsect** for extracting substrings.

See also `strindx`, `strlen`, `strsect`, `strput`

strsect

Purpose	Extracts a substring of a string.	
Format	<code>y = strsect(str,start,len);</code>	
Input	<i>str</i>	string or scalar from which the segment is to be obtained.
	<i>start</i>	scalar, the index of the substring in <i>str</i> . The index of the first character is 1.
	<i>len</i>	scalar, the length of the substring.
Output	<i>y</i>	string, the extracted substring, or a null string if <i>start</i> is greater than the length of <i>str</i> .
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 <i>str</i> is a matrix containing character data, it must be scalar.	
Example	<code>strng = "This is an example string."</code>	
	<code>y = strsect(strng,12,7);</code>	
	<code>y = example</code>	
See also	<code>strlen</code> , <code>strindx</code> , <code>strrindx</code>	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

submat

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

<i>x</i>	NxK matrix.
<i>r</i>	LxM matrix of row indices.
<i>c</i>	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

```
let x[3,4] = 1 2 3 4 5 6 7 8 9 10 11 12;  
let v1 = 1 3;  
let v2 = 2 4;  
y = submat(x,v1,v2);  
z = submat(x,0,v2);
```

x =

1	2	3	4
5	6	7	8
9	10	11	12

y =

2	4
10	12

z =

2	4
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 $y = \text{subscat}(x, v, s);$

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:

$$\begin{array}{ll}
 v[1] < x \leq v[1] \rightarrow & s[1] \\
 x \leq v[2] \rightarrow & s[2] \\
 & \dots \\
 v[p-1] < x \leq v[p] \rightarrow & s[p] \\
 x > v[p] \rightarrow & \text{the original value of } x
 \end{array}$$

If missing is not a category specified in v , missings in x are passed through without change.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

subscat

Example `let x = 1 2 3 4 5 6 7 8 9 10;`
 `let v = 4 5 8;`
 `let s = 10 5 0;`
 `y = subscat(x,v,s);`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

10
10
10
10
 $y =$ 5
0
0
0
9
10

substute

Purpose	Substitutes new values for old values in a matrix, depending on the outcome of a logical expression.	
Format	$y = \text{substute}(x, e, v);$	
Input	x	NxK matrix containing the data to be changed.
	e	LxM matrix, ExE conformable with x containing 1's and 0's. Elements of x will be changed if the corresponding element of e is 1.
	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	<pre> x = { Y 55 30, N 57 18, Y 24 3, N 63 38, Y 55 32, N 37 11 }; e = x[:,1] ".\$==" "Y" .and x[:,2] ".≥ 55 .and x[:,3] ".≥ 30; x[:,1] = substute(x[:,1],e,0\$"R"); </pre>	

substute

$$e = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Here is what x looks like after substitution:

$$y = \begin{pmatrix} R & 55 & 30 \\ N & 57 & 18 \\ Y & 24 & 3 \\ N & 63 & 38 \\ R & 55 & 32 \\ N & 37 & 11 \end{pmatrix}$$

Source `datatran.src`

See also `code, recode`

sumc

Purpose Computes the sum of each column of a matrix.

Format `y = sumc(x);`

Input `x` NxK matrix.

Output `y` Kx1 vector, the sums of each column of `x`.

Remarks To find the sums of the elements in each row of a matrix, transpose before applying **sumc**. If `x` is complex, use the bookkeeping transpose (`.'`).
To find the sums of all of the elements in a matrix, use the **vecr** function before applying **sumc**.

Example `x = round(rndu(5,3)*5);`

`y = sumc(x);`

```

      2 4 3
      2 1 2
x =   5 1 3
      5 1 1
      3 3 4

```

```

      17
y =   10
      13

```

See also `cumsumc`, `meanc`, `stdc`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

surface

surface

Purpose Graphs a 3-D surface.

Library pgraph

Format `surface(x,y,z);`

Input

<i>x</i>	1xK vector, the X axis data.
<i>y</i>	Nx1 vector, the Y axis data.
<i>z</i>	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. Example: <code>_pzclr = 15;</code> 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: <code>_pzclr = { 1, 10, 15 };</code>

surface

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 reset 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));
```

0 Black	8 Dark Grey
1 Blue	9 Light Blue
2 Green	10 Light Green
3 Cyan	11 Light Cyan
4 Red	12 Light Red
5 Magenta	13 Light Magenta
6 Brown	14 Yellow
7 Grey	15 White

Source `psurface.src`

See also `volume, view`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

svd

svd

Purpose	Computes the singular values of a matrix.	
Format	$s = \text{svd}(x);$	
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 Output	<code>_svderr</code>	global scalar, if not all of the singular values can be computed <code>_svderr</code> will be nonzero. The singular values in $s[_\text{svderr}+1], \dots s[M]$ will be correct.
Remarks	Error handling is controlled with the low bit of the trap flag. trap 0 set <code>_svderr</code> and terminate with message trap 1 set <code>_svderr</code> and continue execution	
Example	$x = \begin{Bmatrix} 4 & 3 & 6 & 7, \\ & 8 & 2 & 9 & 5 \end{Bmatrix},$ $y = \text{svd}(x);$ $y = \begin{matrix} 16.521787 \\ 3.3212254 \end{matrix}$	
Source	svd.src	
See also	svd1, svd2, svds	

svd1

Purpose	Computes the singular value decomposition of a matrix so that: $x = u * s * v'$.	
Format	{ <i>u,s,v</i> } = svd1 (<i>x</i>);	
Input	<i>x</i>	NxP matrix whose singular values are to be computed.
Output	<i>u</i>	NxN matrix, the left singular vectors of <i>x</i> .
	<i>s</i>	NxP diagonal matrix, containing the singular values of <i>x</i> arranged in descending order on the principal diagonal.
	<i>v</i>	PxP matrix, the right singular vectors of <i>x</i> .
Global Output	_svderr	global 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 <i>s</i> 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	<i>x</i> = rndu(3,3);	
	{ <i>u</i> , <i>s</i> , <i>v</i> } = svd1 (<i>x</i>);	
	$ \begin{array}{rcl} & 0.97847012 & 0.20538614 & 0.59906497 \\ x = & 0.85474208 & 0.79673540 & 0.22482095 \\ & 0.33340653 & 0.74443792 & 0.75698778 \\ \\ & -0.57955818 & 0.65204491 & 1.48882486 \\ u = & -0.61005618 & 0.05056673 & -0.79074298 \\ & -0.54031821 & -0.75649219 & 0.36847767 \end{array} $	

svd1

$s =$

1.84994646	0.00000000	0.00000000
0.00000000	0.60370542	0.00000000
0.00000000	0.00000000	0.47539239

$v =$

-0.68578561	0.71062560	-0.15719208
-0.54451302	-0.64427479	-0.53704336
-0.48291165	-0.28270348	0.82877927

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	global 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	
Globals	_svderr	
See also	svd, svd1, svdcusv	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

svdcusv

svdcusv

Purpose Computes the singular value decomposition of a matrix so that:
 $x = u * s * v'$ (compact u).

Format $\{ u, s, v \} = \text{svdcusv}(x);$

Input x NxP matrix whose singular values are to be computed.

Output u NxN or NxP matrix, the left singular vectors of x . If $N > P$, u is NxP containing only the P left singular vectors of x .

s NxP or PxP diagonal matrix, the singular values of x arranged in descending order on the principal diagonal. If $N > P$, s is PxP.

v PxP matrix, the right singular vectors of x .

Remarks If not all the singular values can be computed, $s[l, l]$ 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**($s[l, l]$) are correct. If **scalerr**($s[l, l]$) returns a 0, all the singular values have been computed.

See also svd2, svds, svdusv

svds

Purpose	Computes the singular values of a matrix.	
Format	$s = \text{svds}(x);$	
Input	x	NxP matrix whose singular values are to be computed.
Output	s	min(N,P)x1 vector, the singular values of x arranged in descending order.
Remarks	If not all 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 the singular values have been computed.	
See also	svd, svdcusv, svdusv	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

svdusv

svdusv

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

Format { *u,s,v* } = **svdusv**(*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, the singular values of *x* arranged in descending order on the principal diagonal.
v PxP matrix, the right singular vectors of *x*.

Remarks If not all the singular values can be computed, *s*[*I,I*] 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*[*I,I*]) are correct. If **scalerr**(*s*[*I,I*]) returns a 0, all the singular values have been computed.

See also **svd1**, **svdcusv**, **svds**

sysstate

Purpose Gets or sets general system parameters.

Format { *rets...* } = **sysstate**(*case,y*);

Input

Case 1	Version Information Returns the current GAUSS version information in an 8-element numeric vector.
Cases 2 through 7	GAUSS System Paths Get or set 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 Get and set trailing character for the imaginary part of a complex number.
Case 10	Printer Width Get and set lprint width.
Case 11	Auxiliary Output Width Get and set the auxiliary output width.
Case 12	Precision Get and set precision for positive definite matrix routines.
Case 13	LU Tolerance Get and set singularity tolerance for LU decomposition.
Case 14	Cholesky Tolerance Get and set singularity tolerance for Cholesky decomposition.
Case 15	Screen State Get and set window state as controlled by screen command.
Case 16	Automatic print Mode Get and set automatic print mode.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sysstate

Case 17	Automatic lprint Mode Get and set automatic lprint mode.
Case 18	Auxiliary Output Get auxiliary output parameters.
Case 19	Get/Set Format Get and set format parameters.
Case 21	Imaginary Tolerance Get and set the imaginary tolerance.
Case 22	Source Path Get and set the path the compiler will search for source files.
Case 24	Dynamic Library Directory Get and set the path for the default dynamic library directory.
Case 25	Temporary File Path
Case 26	Interface Mode Returns the current interface mode.
Case 28	Random Number Generator Parameters
Case 30	Base Year Toggle Specifies whether year value returned by date is to include base year (1900) or not.

Case 1: Version Information
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.

sysstate

vi[4] indicates the type of machine on which GAUSS is running:

- 1 Intel x86
- 2 Sun SPARC
- 3 IBM RS/6000
- 4 HP 9000
- 5 SGI MIPS
- 6 DEC Alpha

vi[5] indicates the operating system on which GAUSS is running:

- 1 DOS
- 2 SunOS 4.1.x
- 3 Solaris 2.x
- 4 AIX
- 5 HP-UX
- 6 IRIX
- 7 OSF/1
- 8 OS/2
- 9 Windows

**Cases 2
through 7:****GAUSS System Paths**

Get or set 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.

sysstate

Case 8: Complex Number Toggle

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

Get and set 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

Get and set **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.
This may also be set with the **lpwidth** command.

sysstate**See also** `lpwidth`**Case 11: Auxiliary Output Width**

Get and set 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 12: Precision**

Get and set precision for positive definite matrix routines.

Format `oldprec = sysstate(12,prec);`**Input** `prec` scalar, 64 or 80.**Output** `oldprec` scalar, the original value.**Portability** `Windows, UNIX`

This function has no effect under Windows or UNIX. All computations are done in 64-bit precision (except for operations done entirely within the 80x87 on Intel machines).

Remarks The precision will be changed if `prec` is either 64 or 80. Any other number will leave the precision unchanged.**See also** `prcsn`**Case 13: LU Tolerance**

Get and set singularity tolerance for LU decomposition.

Format `oldtol = sysstate(13,tol);`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sysstate

Input	<i>tol</i>	scalar, new tolerance.
Output	<i>oldtol</i>	scalar, the original tolerance.
Remarks	The tolerance must be ≥ 0 . If <i>tol</i> is negative, the tolerance is returned and left unchanged.	
See also	croutp , inv	

Case 14: **Cholesky Tolerance**
Get and set singularity tolerance for Cholesky decomposition.

Format	<i>oldtol</i> = sysstate(14, <i>tol</i>) ;	
Input	<i>tol</i>	scalar, new tolerance.
Output	<i>oldtol</i>	scalar, the original tolerance.
Remarks	The tolerance must be ≥ 0 . If <i>tol</i> is negative, the tolerance is returned and left unchanged.	
See also	chol , invpd , solpd	

Case 15: **Screen State**
Get and set window state as controlled by **screen** command.

Format	<i>oldstate</i> = sysstate(15, <i>state</i>) ;	
Input	<i>state</i>	scalar, new window state.
Output	<i>oldstate</i>	scalar, the original window state.
Remarks	If <i>state</i> = 1, window output is turned on. If <i>state</i> = 0, window output is turned off. If <i>state</i> = -1, the state is returned unchanged.	
See also	screen	

Case 16: **Automatic print Mode**
Get and set automatic **print** mode.

sysstate

Format	<i>oldmode</i> = sysstate (16, <i>mode</i>);	
Input	<i>mode</i>	scalar, mode.
Output	<i>oldmode</i>	scalar, original mode.
Remarks	If <i>mode</i> = 1, automatic print mode is turned on. If <i>mode</i> = 0, it is turned off. If <i>mode</i> = -1, the mode is returned unchanged.	
See also	print on/off	
Case 17:	Automatic lprint Mode	
	Get and set automatic lprint mode.	
Format	<i>oldmode</i> = sysstate (17, <i>mode</i>);	
Input	<i>mode</i>	scalar, mode.
Output	<i>oldmode</i>	scalar, original mode.
Remarks	If <i>mode</i> = 1, automatic lprint mode is turned on. If <i>mode</i> = 0, it is turned off. If <i>mode</i> = -1, the mode is returned unchanged.	
See also	lprint on/off	
Case 18:	Auxiliary Output	
	Get auxiliary output parameters.	
Format	{ <i>state,name</i> } = sysstate (18, <i>dummy</i>);	
Input	<i>dummy</i>	scalar, a dummy argument.
Output	<i>state</i>	scalar, auxiliary output state, 1 - on, 0 - off.
	<i>name</i>	string, auxiliary output filename.
See also	output	
Case 19:	Get/Set Format	
	Get and set format parameters.	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

sysstate

Format *oldfmt* = **sysstate**(19,*fmt*);

Input *fmt* scalar or 11x1 column vector containing the new format parameters. 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:

[1]	format type.
[2]	justification.
[3]	sign.
[4]	leading zero.
[5]	trailing character.
[6]	row delimiter.
[7]	carriage line feed position.
[8]	automatic line feed for row vectors.
[9]	field.
[10]	precision.
[11]	formatted flag

Remarks If *fmt* is scalar 0, then the format parameters will be left unchanged.

See also **format**

Case 21: **Imaginary Tolerance**
Get and set the imaginary tolerance.

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.

See also **hasimag**

Case 22: Source Path

Get and set 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

Get and set 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' default DLL, **gauss.dll**. **gauss.dll** must always be located in the GAUSSHOME directory.

OS/2

sysstate

Changing the path has no effect on GAUSS's default DLL, `gauss.dll`. `gauss.dll` must always be located in the same directory as the GAUSS executable, `gauss.exe`.

DOS

sysstate 24 has no effect, as **dlibrary** and **dllcall** are not supported.

See also `dlibrary`, `dllcall`

Case 25 Temporary File Path

Get or set 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

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

Get and set the random number generator (RNG) parameters.

sysstate

Format	<i>oldprms</i> = sysstate (28, <i>prms</i>);	
Input	<i>prms</i>	scalar 0 to get parameters, or 3x1 matrix of new parameters. [1] seed, 0 < seed < 2^32 [2] multiplier, 0 < mult < 2^32 [3] constant, 0 ≤ const < 2^32
Output	<i>oldprms</i>	3x1 matrix, current parameters.
Portability	Not supported for DOS.	
Remarks	If <i>prms</i> 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 , rndns , rndus , rndn , rndu	
Case 30:	Base Year Toggle Specifies whether year value returned by date is to include base year (1900) or not.	
Format	<i>oldstate</i> = sysstate (30, <i>state</i>);	
Input	<i>state</i>	scalar, 1, 0, or missing value.
Output	<i>oldstate</i>	scalar, the original state.
Portability	DOS sysstate 30 has no effect. It always returns a 4-digit year.	
Remarks	Internally, date acquires the number of years since 1900. sysstate 30 specifies whether date should add the base year to that value or not. If <i>state</i> = 1, date adds 1900, returning a fully-qualified 4-digit year. If state = 0, date returns the number of years since 1900. If <i>state</i> is a missing value, the current state is returned. The default state is 1.	

system

system

Purpose	Quits GAUSS and returns to the operating system.
Format	<code>system;</code> <code>system c;</code>
Input	<i>c</i> 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	<code>exec</code>

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.

Portability UNIX

You can tab backward as well as forward in Text graphic panels, but only forward in TTY graphic panels, including graphic panel 1.

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);
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

tan

tan

Purpose Returns the tangent of its argument.

Format $y = \tan(x);$

Input x NxK matrix.

Output y NxK matrix.

Remarks For real matrices, x should contain angles measured in radians.
To convert degrees to radians, multiply the degrees by $\frac{\pi}{180}$.

Example

```
let x = 0 .5 1 1.5;
y = tan(x);

0.00000000
y = 0.54630249
1.55740772
14.10141995
```

See also atan, pi

tanh

Purpose Computes the hyperbolic tangent.

Format $y = \tanh(x);$

Input x NxK matrix.

Output y NxK matrix containing the hyperbolic tangents of the elements of x .

Example

```
let x = -0.5 -0.25 0 0.25 0.5 1;
x = x * pi;
y = tanh(x);
```

```

-1.570796
-0.785398
x =  0.000000
    0.785398
    1.570796
    3.141593
```

```

-0.917152
-0.655794
y =  0.000000
    0.655794
    0.917152
    0.996272
```

Source trig.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

tempname

tempname

Purpose	Creates a temporary file with a unique name.		
Format	<i>tname</i> = tempname (<i>path</i> , <i>pre</i> , <i>suf</i>) ;		
Input	<i>path</i>	string, path where the file will reside.	
	<i>pre</i>	string, a prefix to begin the file name with.	
	<i>suf</i>	string, a suffix to end the file name with.	
Output	<i>tname</i>	string, unique temporary file name of the form <i>prexxxxnnnnnsuf</i> , where <i>nnnnn</i> is the process id of the calling process.	
Remarks	Any or all of the inputs may be a null string or 0. If <i>path</i> 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.		

time

Purpose	Returns the current system time.
Format	<code>y = time;</code>
Output	<code>y</code> 4x1 numeric vector, the current time in the order: hours, minutes, seconds, and hundredths of a second.
Example	<pre>print time;</pre> <pre>7.000000</pre> <pre>31.000000</pre> <pre>46.000000</pre> <pre>33.000000</pre>
See also	<code>date</code> , <code>datestr</code> , <code>datestring</code> , <code>datestrymd</code> , <code>hsec</code> , <code>timestr</code>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

timestr

timestr

Purpose	Formats a time in a vector to a string.
Format	$ts = \text{timestr}(t);$
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	<pre>t = { 7, 31, 46, 33 }; ts = timestr(t); print ts;</pre> 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	<i>tc</i> = timeutc ;
Output	<i>tc</i> scalar, number of seconds since January 1, 1970 Greenwich Mean Time.
Example	<pre>tc = timeutc; utv = utctodtv(tc); tc = 939235033 utv = 1999 10 6 11 37 13 3 278</pre>
See also	dtvnormal , utctodtv

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

title

title

Purpose	Sets the title for the graph.
Library	pgraph
Format	title(<i>str</i>);
Input	<i>str</i> string, the title to display above the graph.
Global Output	_ptitle
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\L Second title line\L Third title line”); Fonts may be specified in the title string. For instructions on using fonts, see “Publication Quality Graphics” in the <i>User’s Guide</i> .
Source	pgraph.src
See also	xlabel, ylabel, fonts

tocart

Purpose	Converts from polar to cartesian coordinates.		
Format	$xy = \text{tocart}(r, \theta);$		
Input	r	NxK real matrix, radius.	
	θ	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		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

toeplitz

toeplitz

Purpose Creates a Toeplitz matrix from a column vector.

Format `t = toeplitz(x);`

Input `x` Kx1 vector.

Output `t` KxK Toeplitz matrix.

Example `x = seqa(1,1,5);`
`y = toeplitz(x);`

```
      1
      2
x =   3
      4
      5

      1 2 3 4 5
      2 1 2 3 4
y =   3 2 1 2 3
      4 3 2 1 2
      5 4 3 2 1
```

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 Here is a keyword that uses token to parse its string parameter.

```
keyword add(s);
    local tok,sum;
    sum = 0;
    do until s $== "";
        { tok, s } = token(s);
        sum = sum + stof(tok);
    endo;
    format /rd 1,2;
    print "Sum is: " sum;
endp;
```

If you type:

```
add 1 2 3 4 5 6;
```

add will respond:

```
Sum is: 21.00
```

Source token.src

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

token

See also **parse**

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

topolar

Purpose	Converts from cartesian to polar coordinates.	
Format	{ <i>r,theta</i> } = topolar (<i>xy</i>);	
Input	<i>xy</i>	NxK complex matrix containing the <i>X</i> coordinate in the real part and the <i>Y</i> coordinate in the imaginary part.
Output	<i>r</i>	NxK real matrix, radius.
	<i>theta</i>	NxK real matrix, angle in radians.
Source	coord.src	

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

trace

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	verbose trace
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 4, you'll be doing a verbose trace of your program, but the output won't 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.

trace

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

<code>trace 1+8;</code>	<code>trace fn/proc calls/returns to standard output</code>
<code>trace 2+8;</code>	<code>trace line numbers to standard output</code>
<code>trace 1+2+8;</code>	<code>trace line numbers and fn/proc calls/returns to standard output</code>
<code>trace 1+16;</code>	<code>trace fn/proc calls/returns to printer</code>
<code>trace 2+16;</code>	<code>trace line numbers to printer</code>
<code>trace 1+2+16;</code>	<code>trace line numbers and fn/proc calls/returns to printer</code>
<code>trace 4+8;</code>	<code>verbose trace to screen</code>

See also `#lineson`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

trap

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 `x = eye(3)`
`oldval = trapchk(1)`
`trap 1,1`
`y = inv(x)`
`trap oldval,1`

trap

```
if scalerr(y)
    errorlog "WARNING: x is singular";
else
    print "y"y:
endif;
```

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**.

Run the example

```
x = eye(3)
```

It is inverted.

Now try

```
ones(3,3)
```

It isn't.

See also **scalerr, trapchk, error**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

trapchk

trapchk

Purpose Tests the value of the trap flag.

Format $y = \text{trapchk}(m);$

Input m scalar mask value.

Output y scalar which is the result of the bitwise logical AND of the trap flag and the mask.

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:

$y = \text{trapchk}(257);$

trapchk

	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

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

trimr**trimr**

Purpose Trims rows from the top and/or bottom of a matrix.

Format $y = \text{trimr}(x, t, b);$

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

```
x = rndu(5, 3);
y = trimr(x, 2, 1);
```

	0.76042751	0.33841579	0.01844780
	0.05334503	0.38939785	0.65029973
$x =$	0.93077511	0.06961078	0.04207563
	0.53640701	0.06640062	0.07222560
	0.14084669	0.06033813	0.69449247
$y =$	0.93077511	0.06961078	0.04207563
	0.53640701	0.06640062	0.07222560

See also `submat`, `rotater`, `shiftr`

trunc

Purpose Converts numbers to integers by truncating the fractional portion.

Format $y = \text{trunc}(x);$

Input x NxK matrix.

Output y NxK matrix containing the truncated elements of x .

Example $x = 100 * \text{rndn}(2, 2);$

$$x = \begin{bmatrix} 77.68 & -14.10 \\ 4.73 & -158.88 \end{bmatrix}$$
$$y = \text{trunc}(x);$$
$$y = \begin{bmatrix} 77.00 & -14.00 \\ 4.00 & -158.00 \end{bmatrix}$$

See also `ceil`, `floor`, `round`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

type

type

Purpose Returns the symbol table type of the argument.

Format `t = type(x);`

Input `x` matrix or string, can be an expression.

Output

<code>t</code>	scalar.
6	matrix
13	string
15	string array

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 matrices, strings, and string arrays; **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;
if type(k) == 6;
    k == "$+k;
endif;
print k;
produces
+DEN
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.
Format	$y = \text{typecv}(x);$
Input	x string or Nx1 character vector 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:

- 6 Matrix (Numeric, Character, or Mixed)
- 8 Procedure (**proc**)
- 9 Function (**fn**)
- 5 Keyword (**keyword**)
- 13 String
- 15 String Array

It will return the GAUSS missing value code if the symbol is not found, so **typecv** may be used to determine if a symbol is defined or not.

Example

```
xvar = sqrt(5);
yvar = "Montana";
fn area(r) = pi*r*r;
let names = xvar yvar area;
y = typecv(names);
```

```

          XVAR
names =   YVAR
          AREA
```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

typecv

$$y = \begin{matrix} 6 \\ 13 \\ 9 \end{matrix}$$

See also type, typef, varput, varget

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

typedef

Purpose Returns the type of data (the number of bytes per element) in a GAUSS data set.

Format `y = typedef(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

```
infile = "dat1";
outfile = "dat2";
open fin = ^infile;
names = getname(infile);
create fout = ^outfile with ^names,0,typedef(fin);
```

In this example a file `dat2.dat` is created which has the same variables and variable type as the input file, `dat1.dat`. **typedef** is used to return the type of the input file for the **create** statement.

See also `colsf`, `rowsf`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

union

union

Purpose	Returns the union of two vectors with duplicates removed.		
Format	$y = \text{union}(v1,v2,flag);$		
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	let v1 = mary jane linda john;		
	let v2 = mary sally;		
	x = union(v1,v2,0);		
		JANE	
		JOHN	
	x =	LINDA	
	MARY		
	SALLY		

uniqindx

Purpose Computes the sorted index of x , leaving out duplicate elements.

Format $index = \text{uniqindx}(x, flag);$

Input x $N \times 1$ or $1 \times N$ vector.
 $flag$ scalar, 1 if numeric data, 0 if character.

Output $index$ $M \times 1$ 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;`
`ind = uniqindx(x,1);`
`y = x[ind];`

```

      7
      6
ind = 5
      2
      1

```

```

      1
      2
y = 3
      4
      5

```

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

unique

unique

Purpose	Sorts and removes duplicate elements from a vector.
Format	$y = \text{unique}(x, flag);$
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	let $x = 5\ 4\ 4\ 3\ 3\ 2\ 1;$ $y = \text{unique}(x, 1);$ $y =$ 1 2 3 4 5

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 $u = \text{upmat}(x);$
 $u = \text{upmat1}(x);$

Input x NxK matrix.

Output u NxK matrix containing the upper elements of the matrix. The lower elements are replaced with zeros. **upmat** returns the main diagonal intact. **upmat1** replaces the main diagonal with ones.

Example $x = \begin{Bmatrix} 1 & 2 & -1, \\ 2 & 3 & -2, \\ 1 & -2 & 1 \end{Bmatrix};$

$u = \text{upmat}(x);$

$u1 = \text{upmat1}(x);$

The resulting matrices are

$$u = \begin{bmatrix} 1 & 2 & -1 \\ 0 & 3 & -2 \\ 0 & 0 & 1 \end{bmatrix}$$

$$u1 = \begin{bmatrix} 1 & 2 & -1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix}$$

Source diag.src

See also lowmat, lowmat1, diag, diagrv, crout

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

upper

upper

Purpose	Converts a string or matrix of character data to uppercase.	
Format	$y = \text{upper}(x);$	
Input	x	string or NxK matrix containing the character data to be converted to uppercase.
Output	y	string or NxK matrix containing the uppercase equivalent of 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	$x = \text{"uppercase"};$ $y = \text{upper}(x);$ $y = \text{UPPERCASE}$	
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;
external proc 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;
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;
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 PQG programs. Other

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

use

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**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

utctodtv

Purpose	Converts a number of seconds, since or before January 1, 1970 Greenwich Mean Time, to a 1x8 date and time vector.						
Format	<i>dtv</i> = utctodtv (<i>utc</i>);						
Input	<i>utc</i>	scalar, number of seconds since, or before January 1, 1970 Greenwich Mean Time.					
Output	<i>dtv</i>	1x8 date and time vector where:					
	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 current year.					
Example	<pre>tc = timeutc; utv = utctodtv(tc); tc = 939235033 utv = 1999 10 6 11 37 13 3 278</pre>						
See also	dtvnormal , timeutc						

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

utrisol

utrisol

Purpose	Computes the solution of $Ux = b$ where U is an upper triangular matrix.		
Format	$x = \text{utrisol}(b,U);$		
Input	b	PxK matrix.	
	U	PxP upper triangular matrix.	
Output	x	PxK matrix.	
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 $Ux[:,i] = b[:,i]$.		

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 `k0:`

```

    k = key;
    if not k;
        goto k0;
    endif;
    if k == vals("Y") or k == vals("y");
        goto doit;
    else;
        end;
    endif;

```

`doit:`

In this example the **key** function is used to read the keyboard. 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 jump to the label **doit**, otherwise the program will end.

See also `chrs`, `ftos`, `stof`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

varget

varget

Purpose Accesses a global variable whose name is given as a string argument.

Format $y = \text{varget}(s);$

Input s string containing the name of the global symbol you wish to access.

Output y contents of the matrix or string 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 $\text{dog} = \text{rndn}(2, 2);$
 $y = \text{varget}(\text{"dog"});$

 $\text{dog} = \begin{matrix} -0.83429985 & 0.34782433 \\ 0.91032546 & 1.75446391 \end{matrix}$

 $y = \begin{matrix} -0.83429985 & 0.34782433 \\ 0.91032546 & 1.75446391 \end{matrix}$

See also **typecv, varput**

vargetl

Purpose	Accesses a local variable whose name is given as a string argument.
Format	<code>y = vargetl(s);</code>
Input	<i>s</i> string containing the name of the local symbol you wish to access.
Output	<i>y</i> contents of the matrix or string whose name is in <i>s</i> .
Remarks	This function searches the local symbol list for the symbol whose name is in <i>s</i> 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	<pre> proc dog; local x,y; x = rndn(2,2); y = vargetl("x"); print "x" x; print "y" y; retp(y); endp; z = dog; print "z" z; </pre>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

varget l

Produces:

x		
	-0.543851	-0.181701
	-0.108873	0.0648738
y		
	-0.543851	-0.181701
	-0.108873	0.0648738
z		
	-0.543851	-0.181701
	-0.108873	0.0648738

See also varput l

varput

Purpose Allows a matrix or string to be assigned to a global symbol whose name is given as a string argument.

Format `y = varput(x,n);`

Input

<i>x</i>	NxK matrix or string which is to be assigned to the target variable.
<i>n</i>	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 not varput(source,targname);
    print "Symbol table full";
end;
endif;
```

```
source = -0.93519984  0.40642598
         -0.36867581  2.57623519
```

```
target = -0.93519984  0.40642598
         -0.36867581  2.57623519
```

See also `varget`, `typecv`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

varputl

varputl

Purpose Allows a matrix or string to be assigned to a local symbol whose name is given as a string argument.

Format `y = varputl(x,n);`

Input

<i>x</i>	NxK matrix or string which is to be assigned to the target variable.
<i>n</i>	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 dog(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;
    do while putvar $/= vars;
        print "Assign x (" $vars ")": " ;;
        putvar = upper(cons);
        print;
    endo;
    call varputl(x,putvar);
    retp(a+b*c-d/e);
endp;
format /rds 2,1;
i = 0;
```



```
do until i ≥ 5;  
    z = dog(17);  
    print " z is " z;  
    i = i + 1;  
endo;
```

Produces:

```
Assign x ( A B C D E ): a  
    z is 22.3  
Assign x ( A B C D E ): b  
    z is 51.3  
Assign x ( A B C D E ): c  
    z is 34.3  
Assign x ( A B C D E ): d  
    z is 4.6  
Assign x ( A B C D E ): e  
    z is 6.7
```

See also **vargetl**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

vartype

vartype

Purpose	Returns a vector of ones and zeros that indicate whether variables in a data set are character or numeric.
Format	$y = \text{vartype}(\text{names});$
Input	<i>names</i> NX1 character vector of variable names retrieved from a data set header file with the getname function.
Output	<i>y</i> NX1 vector of ones and zeros, 1 if variable is numeric, 0 if character.
Remarks	This function is being obsoleted. See vartypef . If a variable name in <i>names</i> is lowercase, a 0 will be returned in the corresponding element of the returned vector.
Example	<pre>names = getname("freq"); y = vartype(names); print \$names; print y; AGE PAY sex WT 1.0000000 1.0000000 0.0000000 1.0000000</pre>
Source	vartype.src

vartypef

Purpose	Returns a vector of ones and zeros that indicate whether variables in a data set are character or numeric.	
Format	$y = \text{vartypef}(f);$	
Input	f	file handle of an open file.
Output	y	$N \times 1$ 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.	

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

vcm, vcx

vcm, vcx

Purpose	Computes a variance-covariance matrix.	
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.
Source	corr.src	
See also	momentd	

vec, vecr

Purpose Creates a column vector by appending the columns/rows of a matrix to each other.

Format $yc = \text{vec}(x);$
 $yr = \text{vecr}(x);$

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 $x = \begin{Bmatrix} 1 & 2, \\ 3 & 4 \end{Bmatrix};$
 $yc = \text{vec}(x);$
 $yr = \text{vecr}(x);$

$x = \begin{bmatrix} 1.000000 & 2.000000 \\ 3.000000 & 4.000000 \end{bmatrix}$

$yc = \begin{bmatrix} 1.000000 \\ 3.000000 \\ 2.000000 \\ 4.000000 \end{bmatrix}$

$yr = \begin{bmatrix} 1.000000 \\ 2.000000 \\ 3.000000 \\ 4.000000 \end{bmatrix}$

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

vech

vech

Purpose	Vectorizes a symmetric matrix by retaining only the lower triangular portion of the matrix.
Format	$v = \text{vech}(x);$
Input	x NxN symmetric matrix.
Output	v $(N*(N+1)/2) \times 1$ 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 triangular portion of the matrix into a column vector in row-wise order.
Example	$x = \text{seqa}(10,10,3) + \text{seqa}(1,1,3)';$ $v = \text{vech}(x);$ $sx = \text{xpnd}(v);$ $x = \begin{matrix} & 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{matrix}$ $v = \begin{matrix} 11 \\ 21 \\ 22 \\ 31 \\ 32 \\ 33 \end{matrix}$ $sx = \begin{matrix} 11 & 21 & 31 \\ 21 & 22 & 32 \\ 31 & 32 & 33 \end{matrix}$
See also	xpnd

vector (dataloop)

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 **externs**, or as the result of a previous **make**, **vector**, or **code** statement.

Example `vector const = 1;`

See also `make`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

vget

vget

Purpose Extracts a matrix or string from a data buffer constructed with **vput**.

Format `{ x,dbufnew } = vget(dbuf,name);`

Input

<i>dbuf</i>	Nx1 vector, a data buffer containing various strings and matrices.
<i>name</i>	string, the name of the string or matrix to extract from <i>dbuf</i> .

Output

<i>x</i>	LxM matrix or string, the item extracted from <i>dbuf</i> .
<i>dbufnew</i>	Kx1 vector, the remainder of <i>dbuf</i> after <i>x</i> has been extracted.

Source `vpack.src`

See also `vlist`, `vput`, `vread`

view

Purpose Sets the position of the observer in workbox units for 3-D plots.

Library pgraph

Format `view(x,y,z);`

Input

<i>x</i>	scalar, the X position in workbox units.
<i>y</i>	scalar, the Y position in workbox units.
<i>z</i>	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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

viewxyz

viewxyz

Purpose Sets the position of the observer in plot coordinates for 3-D plots.

Library pgraph

Format `viewxyz(x,y,z);`

Input

<i>x</i>	scalar, the X position in plot coordinates.
<i>y</i>	scalar, the Y position in plot coordinates.
<i>z</i>	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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

vlist

Purpose	Lists the contents of a data buffer constructed with vput .
Format	vlist (<i>dbuf</i>);
Input	<i>dbuf</i> Nx1 vector, a data buffer containing various strings and matrices.
Remarks	vlist lists the names of all the strings and matrices stored in <i>dbuf</i> .
Source	vpack.src
See also	vget , vput , vread

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

vnamecv

vnamecv

Purpose	Returns the names of the elements of a data buffer constructed with vput .
Format	<i>cv</i> = vnamecv (<i>dbuf</i>);
Input	<i>dbuf</i> Nx1 vector, a data buffer containing various strings and matrices.
Output	<i>cv</i> Kx1 character vector containing the names of the elements of <i>dbuf</i> .
See also	vget , vput , vread , vtypecv

volume

Purpose	Sets the length, width, and height ratios of the 3-D workbox.
Library	pgraph
Format	<code>volume(x,y,z);</code>
Input	<i>x</i> scalar, the X length of the 3-D workbox. <i>y</i> scalar, the Y length of the 3-D workbox. <i>z</i> 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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

vput

vput

Purpose	Inserts a matrix or string into a data buffer.	
Format	<i>dbufnew</i> = vput (<i>dbuf</i> , <i>x</i> , <i>xname</i>) ;	
Input	<i>dbuf</i>	Nx1 vector, a data buffer containing various strings and matrices. If <i>dbuf</i> is a scalar 0, a new data buffer will be created.
	<i>x</i>	LxM matrix or string, item to be inserted into <i>dbuf</i> .
	<i>xname</i>	string, the name of <i>x</i> , will be inserted with <i>x</i> into <i>dbuf</i> .
Output	<i>dbufnew</i>	Kx1 vector, the data buffer after <i>x</i> and <i>xname</i> have been inserted.
Remarks	If <i>dbuf</i> already contains <i>x</i> , the new value of <i>x</i> 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	$x = \text{vread}(\text{dbuf}, \text{xname});$		
Input	<i>dbuf</i>	Nx1 vector, a data buffer containing various strings and matrices.	
	<i>xname</i>	string, the name of the matrix or string to read from <i>dbuf</i> .	
Output	<i>x</i>	LXM matrix or string, the item read from <i>dbuf</i> .	
Remarks	vread , unlike vget , does not change the contents of <i>dbuf</i> . Reading <i>x</i> from <i>dbuf</i> does not remove it from <i>dbuf</i> .		
Source	vpack.src		
See also	vget , vlist , vput		

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

vtypecv

vtypecv

Purpose	Returns the types of the elements of a data buffer constructed with vput .
Format	<i>cv</i> = vtypecv (<i>dbuf</i>);
Input	<i>dbuf</i> Nx1 vector, a data buffer containing various strings and matrices.
Output	<i>cv</i> Kx1 character vector containing the types of the elements of <i>dbuf</i> .
See also	vget , vput , vread , vnamecv

wait, waitc

Purpose Waits until any key is pressed.

Format `wait;`
 `waitc;`

Portability **UNIX**

These functions wait for a keystroke in the active graphic panel. If you are working in terminal mode, they don't "see" any keystrokes until ENTER is pressed.

Remarks `waitc` clears any pending keystrokes before waiting until another key is pressed.

Source `wait.src, waitc.src`

See also `pause`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinClear

WinClear

Purpose Clear a window.

Format `winClear(whandle)`;

Input *whandle* scalar, window handle.

Portability UNIX only

Remarks **WinClear** clears the window to the background color. The effect of **WinClear** on window data depends upon the window type, as follows:

If *whandle* is a PQG window, the current graphics data are discarded. The image can only be restored by repeating the sequence of commands that originally generated it.

If *whandle* is a Text window, its text buffer is cleared, destroying all data for the window. The previous contents of the window can only be restored by repeating the sequence of commands that originally generated it.

If *whandle* is a TTY window, the current output line is panned to the top of the window. Window data is not affected.

See Also `WinClearTTYLog`

WinClearArea

Purpose Clear a rectangular region of a Text window.

Format `WinClearArea(whandle, row, col, nrows, ncols)`;

Input

<i>whandle</i>	scalar, window handle.
<i>row</i>	scalar, the topmost row to clear.
<i>col</i>	scalar, the leftmost column to clear.
<i>nrows</i>	scalar, the number of rows to clear.
<i>ncols</i>	scalar, the number of columns to clear.

Portability UNIX only

Remarks `WinClearArea` clears a rectangular region of a Text window to the background color. The region has its top-left corner at (*row,col*), and is *nrows* $\times$ *ncols* in size.

The corresponding region of the text buffer is cleared, destroying all data for the region. The previous contents of the region can only be restored by repeating the sequence of commands that originally generated it.

`WinClearArea` is supported only for Text windows.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinClearTTYLog

WinClearTTYLog

Purpose Clear the output log of a TTY window.

Format `WinClearTTYLog(ttywhandle) ;`

Input *ttywhandle* scalar, window handle of a TTY window.

Portability UNIX only

Remarks **WinClearTTYLog** deletes the output log of a TTY window, destroying the window data. The window is automatically cleared and refreshed.

See Also **WinClear**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinClose

Purpose	Close a window.
Format	<i>ret</i> = WinClose (<i>whandle</i>);
Input	<i>whandle</i> scalar, window handle.
Output	<i>ret</i> scalar 0.
Portability	UNIX only
Remarks	WinClose closes (destroys) the window specified by <i>whandle</i>. When a window is closed, GAUSS destroys all of the internal information required to maintain the window. Any further commands which reference <i>whandle</i> will result in an error. When a window is closed, the window handle should be set to zero, to let your program know that it no longer refers to a valid window. The easiest way to do this is: <pre>wh = WinClose(wh);</pre> If you close the active window, you will have no active window. To set a new one, use WinSetActive. Window 1 cannot be closed.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinCloseAll

WinCloseAll

Purpose Close all windows except window 1.

Format WinCloseAll;

Portability UNIX only

Remarks WinCloseAll closes all open windows except window 1. You should set the affected window handles to zero after a WinCloseAll, to let your program know that they no longer refer to valid windows. The easiest way to do this is:

```
clear wh1, wh2 [,wh3...];
```

If window 1 is not the active window, you will have no active window. To set a new one, use WinSetActive.

The **end** command also closes all open windows (except window 1). If you want to terminate a program but leave its windows open, use **stop**.

window

Purpose	Partitions the window into tiled regions of equal size.	
Library	pgraph	
Format	window (<i>row,col,typ</i>) ;	
Input	<i>row</i>	scalar, number of rows of graphic panels.
	<i>col</i>	scalar, number of columns of graphic panels.
	<i>typ</i>	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 (<i>row</i>)$\times$(<i>col</i>) 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 “Publication Quality Graphics” in the <i>User’s Guide</i>.)	
Source	pwindow.src	
See also	endwind , begwind , setwind , nextwind , getwind , makewind	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinGetActive

WinGetActive

Purpose Get the active window.

Format *whandle* = **WinGetActive**;

Output *whandle* scalar, the active window handle.

Portability **UNIX only**

Remarks **WinGetActive** returns the window handle of the active window. If there is no active window, **WinGetActive** returns a -1.

WinGetAttributes

Purpose	Get window attributes.	
Format	<i>wattr</i> = WinGetAttributes (<i>whandle</i>);	
Input	<i>whandle</i>	scalar, window handle.
Output	<i>wattr</i>	25×1 vector, window attributes, or scalar - 1 if error.
	PQG Window	
	[1] X window position in pixels	
	[2] Y window position in pixels	
	[3] X window size in pixels	
	[4] Y window size in pixels	
	[5] unused	
	[6] unused	
	[7] unused	
	[8] colormap	
	[9] foreground color	
	[10] background color	
	[11] aspect ratio attribute	
	[12] unused	
	[13] resizing attribute	
	[14] x pixel at left edge	
	[15] y pixel at top edge	
	[16] GAUSS window handle	
	[17] system window handle	
	[18] active window flag	
	[19] window type	
	[20] unused	
	[21] unused	
	[22] unused	
	[23] bits per pixel	
	[24] unused	
	[25] unused	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinGetAttributes**Text Window**

- [1]** X window position in pixels
- [2]** Y window position in pixels
- [3]** window rows
- [4]** window columns
- [5]** buffer rows
- [6]** buffer columns
- [7]** font (handle)
- [8]** colormap
- [9]** foreground color
- [10]** background color
- [11]** text wrapping mode
- [12]** automatic refresh mode
- [13]** resizing attribute
- [14]** buffer row at top of window
- [15]** buffer column at left side of window
- [16]** GAUSS window handle
- [17]** system window handle
- [18]** active window flag
- [19]** window type
- [20]** cursor text row
- [21]** cursor text column
- [22]** cursor type
- [23]** bits per pixel
- [24]** character height in pixels
- [25]** character width in pixels

TTY Window

- [1]** X window position in pixels
- [2]** Y window position in pixels
- [3]** window rows
- [4]** window columns
- [5]** number of lines to log
- [6]** number of lines logged
- [7]** font (handle)
- [8]** colormap
- [9]** foreground color

WinGetAttributes

- [10] background color
- [11] text wrapping mode
- [12] automatic refresh mode
- [13] resizing attribute
- [14] buffer line at top of window
- [15] buffer column at left of window
- [16] GAUSS window handle
- [17] system window handle
- [18] active window flag
- [19] window type
- [20] cursor text row (number of lines logged+1)
- [21] cursor text column
- [22] cursor type
- [23] bits per pixel
- [24] character height in pixels
- [25] character width in pixels

Portability UNIX only

Remarks **WinGetAttributes** returns the attributes of the window specified by *whandle*. If *whandle* is 1, the attributes of the GAUSS command window are returned. If *whandle* is 2, the attributes of the GAUSS help window are returned.

wattr[1] and *wattr*[2] indicate the position on the display of the upper left corner of the window border (i.e., the border the window manager supplies).

wattr[3] and *wattr*[4] indicate the size of the window. For PQG windows, these values are given in pixels, for Text and TTY windows, these values are given in rows and columns.

wattr[8] is the colormap number as specified in the **WinSetColormap** command.

For a PQG window, *wattr*[11] indicates whether the aspect ratio is fixed or not. For Text and TTY windows it indicates how text is wrapped.

wattr[12] indicates how the window is refreshed. *wattr*[13] indicates how the window can be resized.

wattr[14] and *wattr*[15] indicate the buffer coordinates at the upper left corner of Text windows. For TTY windows, *wattr*[14] indicates the line of the output log displayed at the top of the window.

WinGetAttributes

wattr[**16**], the GAUSS window handle, is the handle ID assigned by GAUSS. *wattr*[**17**], the system window handle, is the handle ID assigned by the window manager. All GAUSS functions that take a window handle expect a GAUSS window handle.

wattr[**18**] indicates whether this window is the active window (1 = true, 0 = false).

wattr[**22**] is the cursor type number as specified in the **csrtype** command.

See **WinOpenPQG**, **WinOpenText**, and **WinOpenTTY** for details on window attributes.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinGetColorCells

Purpose	Get normalized RGB values from a window colormap.	
Format	<i>rgb</i> = WinGetColorCells (<i>whandle</i> , <i>cellidx</i>);	
Input	<i>whandle</i>	scalar, window handle.
	<i>cellidx</i>	Nx1 vector, color cell indices.
Output	<i>rgb</i>	Nx3 matrix, RGB settings of color cells.
Portability	UNIX only	
Remarks	Each row of <i>rgb</i> contains the RGB values for the color cell specified in the corresponding row of <i>cellidx</i> . The RGB values are normalized to range from 0.0 (black) to 1.0 (full intensity).	
	The columns are laid out as follows:	
	[. , 1]	red values
	[. , 2]	green values
	[. , 3]	blue values
	If any row of <i>cellidx</i> contains an invalid color cell index, the corresponding row of <i>rgb</i> will contain -1 in each column.	

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinGetCursor

WinGetCursor

Purpose Get the cursor position for a Text or TTY window.

Format { *r*, *c* } = **WinGetCursor**(*whandle*);

Input *whandle* scalar, window handle.

Output *r* scalar, cursor row position.
c scalar, cursor column position.

Portability UNIX only

Remarks If *whandle* is a Text window, (*r*,*c*) specifies the location of the cursor in the window buffer. If *whandle* is a TTY window, *r* specifies the current output line (i.e., the number of lines logged + 1), *c* specifies the cursor location on that line. If *whandle* is a PQG window, *r* and *c* are meaningless.

For a Text window, the upper left corner of the buffer is (1,1). For a TTY window, the first column of the line is column 1.

WinMove

Purpose Move a window.

Format `winMove(whandle,x,y)`;

Input

<i>whandle</i>	scalar, window handle.
<i>x</i>	scalar, X window position in pixels.
<i>y</i>	scalar, Y window position in pixels.

Portability UNIX only

Remarks **WinMove** moves the window specified by *whandle* to the position (*x*,*y*). (*x*,*y*) specifies the position on the display of the upper left corner of the window border (i.e., the border supplied by the window manager). The upper left corner of the display is (0,0). (*x*,*y*) can specify an off-screen location.

To use the current value of *x* or *y*, set that argument to a missing value.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinOpenPQG

WinOpenPQG

Purpose Open a PQG graphics window.

Format *whandle* = **WinOpenPQG**(*wattr*, *wtile*, *ititle*) ;

Input

<i>wattr</i>	13x1 vector, window attributes. [1] X window position in pixels. [2] Y window position in pixels. [3] X window size in pixels. [4] Y window size in pixels. [5] unused. [6] unused. [7] unused. [8] colormap. [9] foreground color. [10] background color. [11] aspect ratio attribute. [12] unused. [13] resizing attribute.
<i>wtile</i>	string, window title.
<i>ititle</i>	string, icon title.

Output *whandle* scalar, window handle.

Portability UNIX only

Remarks *wattr*[1] and *wattr*[2] specify the position on the display of the upper left corner of the window border (i.e., the border supplied by the window manager). The upper left corner of the display is (0,0).

wattr[8] specifies the colormap for the window. See **WinSetColormap** for the list of valid colormap values.

wattr[11], the aspect ratio attribute, determines whether the aspect ratio of the window is maintained during resize and zoom actions.

0	aspect ratio can change
1	aspect ratio is fixed

WinOpenPQG

The aspect ratio attribute cannot be changed.

*wattr***[13]**, the resizing attribute, controls the resizing of the window as follows:

- 0 the window cannot be resized
- 1 the window can be resized only through a **WinResize** call
- 2 the window can be resized through a **WinResize** call or external event (e.g., resizing with the mouse)

The resizing attribute cannot be changed.

Default values are defined for the elements of *wattr* as follows:

[1]	0	window position: (0,0)
[2]	0	
[3]	640	window size: 640x480
[4]	480	
[5]	0	unused
[6]	0	unused
[7]	1	unused
[8]	6	colormap: 16 color
[9]	15	foreground color: white
[10]	0	background color: black
[11]	1	fixed aspect ratio
[12]	0	unused
[13]	2	resize by function or mouse

To use the default value for any attribute, set that element to a missing value.

WinOpenPQG returns a -1 if it fails to open the window.

Note that opening a window does not make it the active window.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinOpenText

WinOpenText

Purpose	Open a Text window.		
Format	<i>whandle</i> = WinOpenText (<i>wattr</i> , <i>wtile</i> , <i>itile</i>) ;		
Input	<i>wattr</i>	13x1 vector, window attributes. [1] X window position in pixels. [2] Y window position in pixels. [3] window rows. [4] window columns. [5] buffer rows. [6] buffer columns. [7] font (handle). [8] colormap. [9] foreground color. [10] background color. [11] text wrapping mode. [12] automatic refresh mode. [13] resizing attribute.	
	<i>wtile</i>	string, window title.	
	<i>itile</i>	string, icon title.	
Output	<i>whandle</i>	scalar, window handle.	
Portability	UNIX only		
Remarks	<i>wattr</i> [2] and <i>wattr</i> [3] specify the position on the left corner of the window border (i.e., the border supplied by the window manager). The upper left corner of the display is (0,0). <i>wattr</i> [8] specifies the colormap for the window. See WinSetColormap for the list of valid colormap values. <i>wattr</i> [11], the text wrapping mode, controls how text is wrapped in the window buffer as follows: 0 text is word-wrapped 1 text is character-wrapped		

2 text is clipped

The text wrapping mode can be changed with **WinSetTextWrap**.

wattr[**12**], the automatic refresh mode, controls how the window display is refreshed as follows:

- 0 the display is refreshed whenever a newline (\ n) is output
- 1 the display is refreshed whenever anything is output
- 2 the display is never automatically refreshed; you must explicitly refresh it

The refresh mode can be changed with **WinSetRefresh**.

wattr[**13**], the resizing attribute, controls the resizing of the window as follows:

- 0 the window cannot be resized
- 1 the window can be resized only through a **WinResize** call
- 2 the window can be resized through a **WinResize** call or external event (e.g., resizing with the mouse)

The resizing attribute cannot be changed.

Default values are defined for the elements of *wattr* as follows:

[1]	0	window position: (0,0)
[2]	0	
[3]	25	window size: 25x80
[4]	80	
[5]	25	buffer size: 25x80
[6]	80	
[7]	1	system default font
[8]	0	system default colormap
[9]	1	system foreground color
[10]	0	system background color
[11]	1	character wrap
[12]	0	refresh on \ n
[13]	2	resize by function or mouse

To use the default value for any attribute, set that element to a missing value.

WinOpenText returns a -1 if it fails to open the window.

Note that opening a window does not make it the active window.

WinOpenTTY

WinOpenTTY

Purpose Open a TTY window.

Format *whandle* = **WinOpenTTY**(*wattr*,*wtitle*,*ititle*);

Input

<i>wattr</i>	13x1 vector, window attributes. [1] X window position in pixels. [2] Y window position in pixels. [3] window rows. [4] window columns. [5] number of lines to log. [6] unused. [7] font (handle). [8] colormap. [9] foreground color. [10] background color. [11] text wrapping mode. [12] automatic refresh mode. [13] resizing attribute.
<i>wtitle</i>	string, window title.
<i>ititle</i>	string, icon title.

Output *whandle* scalar, window handle.

Portability UNIX only

Remarks *wattr*[1] and *wattr*[2] specify the position on the display of the upper left corner of the window border. The upper left corner of the display is (0,0).

wattr[8] specifies the colormap for the window. See **WinSetColormap** for the list of valid colormap values.

wattr[11], the text wrapping mode, controls how text is wrapped in the window as follows:

0	text is word-wrapped
1	text is character-wrapped
2	text is clipped

The text wrapping mode can be changed with **WinSetTextWrap**.

*wattr***[12]**, the automatic refresh mode, controls how the window display is refreshed as follows:

- 0 the display is refreshed whenever a newline (\ n) is output
- 1 the display is refreshed whenever anything is output
- 2 the display is never automatically refreshed; you must explicitly refresh it

The refresh mode can be changed with **WinSetRefresh**.

*wattr***[13]**, the resizing attribute, controls the resizing of the window as follows:

- 0 the window cannot be resized
- 1 the window can be resized only through a **WinResize** call
- 2 the window can be resized through a **WinResize** call or external event (e.g. resizing with the mouse)

The resizing attribute cannot be changed.

Default values are defined for the elements of *wattr* as follows:

[1]	0	window position: (0,0)
[2]	0	
[3]	25	window size: 25x80
[4]	80	
[5]	500	log 500 lines
[6]	0	unused
[7]	1	system default font
[8]	0	system default colormap
[9]	1	system foreground color
[10]	0	system background color
[11]	1	character wrap
[12]	0	refresh on \ n
[13]	2	resize by function or mouse

To use the default value for any attribute, set that element to a missing value.

WinOpenTTY returns a -1 if it fails to open the window.

Note that opening a window does not make it the active window.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinPan

WinPan

Purpose Pan through a Text or TTY window.

Format `WinPan(whandle, r, c)`;

Input

<i>whandle</i>	scalar, window handle.
<i>r</i>	scalar, number of rows to pan.
<i>c</i>	scalar, number of columns to pan.

Portability UNIX only

Remarks **WinPan** pans through the contents of the window specified by *whandle*. To pan up and left, use negative *r* and *c*. To pan down and right, use positive *r* and *c*.

WinPan is not supported for PQG windows.

WinPrint

Purpose	Print to a specified window.
Format	WinPrint <code>[[/flush] [/typ] [/fmted] [/mf]</code> <code>[[/jnt] whandle [expr1 expr2...] ; [;]</code>
Input	<i>whandle</i> scalar, window handle.
Portability	UNIX only
Remarks	WinPrint and the print statement have exactly the same syntax and behavior, except that WinPrint prints to a specified window, and print prints to the active window. WinPrint is not supported for PQG windows.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinPrintPQG

WinPrintPQG

Purpose Prints a PQG graph to a file or the printer.

Format `WinPrintPQG(whandle,prtopts)`;

Input

whandle scalar, window handle.

prtopts string, print options.

WinPrintPQG prints the graphics data for *whandle* to the system printer, or to a file, if one is specified in *prtopts*.

prtopts consists of a series of option flags and their settings. Separate items in *prtopts* with spaces.

The following flags and settings are defined:

-f output file name, any valid filename. File must be writeable.

-o print orientation.

L landscape

P portrait

-t printer/file type.

PS PostScript

EPS Encapsulated PostScript

HPLJET3 HP LaserJet III, IIID

HPGL Hewlett-Packard Graphics Language

PIC Lotus PIC

The file {prndev.tbl} in the \$GAUSSHOME directory lists additional printer/file types that are available but not tested. You can try any of the types listed there except the type 4 entries (i.e., those with a 4 in the second field), which are not supported.

-r print resolution.

1 low resolution

2 medium resolution

3 high resolution

4 maximum resolution

-m print margins, specified in inches.

left, right, top, bottom

WinPrintPQG

This is a list of four decimal values. Separate margin values with commas. Do not put any white space between values.

Example:

```
-m
.5, .5, .
5, .75
```

```
-c    print color mode.
      BW        black/white
      GRAY      gray scale
      RGB       16 color

-a    print aspect ratio.
      F         fixed
      V         variable

-g    print placement.
      TL        top left
      T         top
      TR        top right
      L         left
      C         center
      R         right
      BL        bottom left
      B         bottom
      BR        bottom right
```

Flags and settings are case-sensitive.

Print options not specified in *prtopts* are set to their values in *.gaussrc*. *prtopts* can be a null string.

WinPrintPQG is supported only for PQG windows.

Portability UNIX only

```
Example      WinPrintPQG(wh, "-f /tmp/graph.out -o P -t
               EPS");
               WinPrintPQG(wh, "-m 1,1,1,1.5 -a V -r 1");
               WinPrintPQG(wh, "-a F -g TL -m 1,1,1,4");
```

WinRefresh

WinRefresh

Purpose Refresh a window.

Format `WinRefresh(whandle)`;

Input *whandle* scalar, window handle.

Portability UNIX only

Remarks **WinRefresh** redraws the window specified by *whandle* using the current contents of the window's graphics data, text buffer or output log, depending upon the window type.

This is the only way to update windows with an automatic refresh mode = [*no refresh*].

WinRefreshArea

Purpose Refresh a rectangular region of a Text window.

Format `WinRefreshArea(whandle, row, col, nrows, ncols)`;

Input

<i>whandle</i>	scalar, window handle.
<i>row</i>	scalar, top row of area in buffer to refresh.
<i>col</i>	scalar, left column of area in buffer to refresh.
<i>nrows</i>	scalar, number of rows to refresh.
<i>ncols</i>	scalar, number of cols to refresh.

Portability UNIX only

Remarks `WinRefreshArea` redraws a rectangular region of the Text window using the current contents of the text buffer. The region has its top-left corner at (*row,col*), and is *nrows*×*ncols* in size.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinResize

WinResize

Purpose Resize a window.

Format **WinResize**(*whandle*, *height*, *width*);

Input

<i>whandle</i>	scalar, window handle.
<i>height</i>	scalar, window height.
<i>width</i>	scalar, window width.

Portability UNIX only

Remarks **WinResize** changes the size of the window specified by *whandle*. The position of the upper left corner of the window remains fixed.

For PQG windows, *height* and *width* are specified in pixels. The final size of the window may differ from the requested size. If the window was opened with fixed aspect ratio, the *width* will be used to calculate the *height* required to maintain the correct ratio.

For Text and TTY windows, *height* and *width* are specified in characters. (Thus, the actual requested size of the window is dependent upon the window font.) For Text windows, the size of the window cannot be larger than the text buffer.

height and *width* can be nonintegral, but will be truncated to integers.

To use the current value of *height* or *width*, set that argument to a missing value.

WinSetActive

Purpose	Set the active window.
Format	<i>whout</i> = WinSetActive (<i>whin</i>);
Input	<i>whin</i> scalar, window handle or -1.
Output	<i>whout</i> scalar, window handle.
Portability	UNIX only
Remarks	WinSetActive sets the active window to <i>whin</i> and returns the handle of the previous active window in <i>whout</i>. If <i>whin</i> is -1, the active window is deactivated. If there was no previous active window, WinSetActive returns a -1.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinSetBackground

WinSetBackground

Purpose Set the background color for the specified window.

Format *oldback* = **WinSetBackground**(*whandle*, *color*);

Input

<i>whandle</i>	scalar, window handle.
<i>color</i>	scalar, color index or -1.

Output

<i>oldback</i>	scalar, previous background color index.

Portability UNIX only

Remarks **WinSetBackground** sets the background color index for *whandle* to *color* and returns the previous background color index. If *color* is -1, the current background color index is returned.

WinSetColorCells

Purpose	Set one or more cells in a window's colormap.						
Format	<i>ret</i> = WinSetColorCells (<i>whandle</i> , <i>cellidx</i> , <i>colorspec</i>);						
Input	<i>whandle</i> scalar, window handle. <i>cellidx</i> Nx1 vector, color cell indices. <i>colorspec</i> Nx3 matrix, color specifications.						
Output	<i>ret</i> scalar, 0 if call is successful, -1 if one or more colors can't be found.						
Portability	UNIX only						
Remarks	The columns of <i>colorspec</i> are interpreted as follows: <table> <tr> <td>[. , 1]</td><td>red values</td></tr> <tr> <td>[. , 2]</td><td>green values</td></tr> <tr> <td>[. , 3]</td><td>blue values</td></tr> </table> Color values can range from 0.0 (black) to 1.0 (full intensity). Values < 0.0 will be treated as 0.0, values > 1.0 will be treated as 1.0. Because of the need to handle many different types of displays, the X color handling protocol specifies that the contents of <i>colorspec</i> be regarded as color ``suggestions" rather than color commands. Thus, WinSetColorCells will give you colors as close to the settings of <i>colorspec</i> as it can, depending on your display's hardware limitations, the current state of the system colormap, and the <i>color_tolerance</i> field in <i>gauss.cfg</i>. Due to the X11 color handling protocol, WinSetColorCells has to regard the contents of <i>colorspec</i> as color ``suggestions" rather than color commands, but it will always give you colors as close to the settings of <i>colorspec</i> as it can. WinSetColorCells attempts to change the RGB values for the specified color cells. If it succeeds in setting all of them, it returns a 0. If it fails to set all of them — specifically, if <i>cellidx</i> or <i>colorspec</i> contain any invalid values, or the system fails to reset the color cell — WinSetColorCells returns a -1.	[. , 1]	red values	[. , 2]	green values	[. , 3]	blue values
[. , 1]	red values						
[. , 2]	green values						
[. , 3]	blue values						

WinSetColormap

WinSetColormap

Purpose Set the colormap for a window.

Format *mapinfo* = **WinSetColormap**(*whandle*, *cmap*);

Input

whandle scalar, window handle.

cmap scalar, colormap number or -1.

The supported colormaps are:

- 0** System Color
- 1** System BW
- 2** True BW
- 3** 4 Gray
- 4** 4 Color
- 5** 16 Gray
- 6** 16 Color
- 7** 64 Gray
- 8** 256 Color

Output

mapinfo 3x1 vector, colormap information.

mapinfo[**1**] number of colors in colormap

mapinfo[**2**] number of bits per pixel

mapinfo[**3**] current colormap number

Portability UNIX only

Remarks **WinSetColormap** sets the colormap for the window *whandle* to *cmap*, and returns information about the selected colormap. If *cmap* = -1, no changes are made, but the information for the current colormap is returned.

On error, scalar 0 is returned.

WinSetCursor

Purpose Set the cursor position for a Text or TTY window.

Format `WinSetCursor ( whandle, r, c );`

Input

<i>whandle</i>	scalar, window handle.
<i>r</i>	scalar, cursor row.
<i>c</i>	scalar, cursor column.

Portability UNIX only

Remarks If *whandle* is a Text window, (*r*,*c*) specifies where to locate the cursor in the window buffer. If *whandle* is 1 or a TTY window, *r* is ignored, and *c* specifies where to locate the cursor on the current line.

For a Text window, the upper left corner of the buffer is (1,1). For a TTY window, the first column of a line is column 1.

(*r*,*c*) can specify a location that is outside the buffer. You can think of the buffer as covering a small region in "text coordinate space"; text written within the buffer bounds is remembered, text written outside it is lost.

Note that there is no right-hand boundary in a TTY window. So, if you locate to column 254328 and begin printing, GAUSS will attempt to allocate enough memory to store a 254K line of output.

r and *c* are truncated if they are not integers.

WinSetCursor is not supported for PQG windows.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinSetForeground

WinSetForeground

Purpose	Set the foreground color for the specified window.
Format	<i>oldfore</i> = WinSetForeground (<i>whandle</i> , <i>color</i>);
Input	<i>whandle</i> scalar, window handle. <i>color</i> scalar, color index or -1.
Output	<i>oldfore</i> scalar, previous foreground color index.
Portability	UNIX only
Remarks	WinSetForeground sets the foreground color index for <i>whandle</i> to <i>color</i> and returns the previous foreground color index. If <i>color</i> is -1, the current foreground color index is returned.

WinSetRefresh

Purpose Set the refresh mode for a window.

Format *oldref* = **WinSetRefresh**(*whandle*, *refresh*);

Input

<i>whandle</i>	scalar, window handle.
<i>refresh</i>	scalar, refresh mode.
0	refresh on newline (\n); this is the system default
1	refresh on any output
2	refresh only on WinRefresh call

Output *oldref* scalar, previous refresh mode.

Portability UNIX only

Remarks To get the current refresh mode, call **WinGetAttributes**.
WinSetRefresh is not supported for window 1 or PQG windows.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

WinSetTextWrap

WinSetTextWrap

Purpose Set the text wrapping mode for a window.

Format *oldtwrap* = **WinSetTextWrap**(*whandle*, *textwrap*);

Input

<i>whandle</i>	scalar, window handle.
<i>textwrap</i>	scalar, text wrapping mode.
	0 text is word-wrapped
	1 text is character-wrapped
	2 text is clipped

Output *oldtwrap* scalar, previous text wrapping mode.

Portability UNIX only

Remarks For Text windows, the text wrapping mode specifies how lines are stored in the window buffer. For TTY windows, the text wrapping mode specifies how lines are displayed in the window. Thus, for Text windows, clipped output is actually lost, as is output that scrolls off the top of the buffer; for TTY windows, clipped output is merely displayed differently than wrapped output.

To get the current text wrap mode, call **WinGetAttributes**.

WinSetTextWrap is not supported for PQG windows.

WinZoomPQG

Purpose Display a zoomed image in a PQG window.

Format `WinZoomPQG(whandle, x, y, xsize, ysize) ;`

Input

<i>whandle</i>	scalar, window handle.
<i>x</i>	scalar, x position of zoom rectangle in pixels.
<i>y</i>	scalar, y position of zoom rectangle in pixels.
<i>xsize</i>	scalar, x size of zoom rectangle in pixels.
<i>ysize</i>	scalar, y size of zoom rectangle in pixels.

Portability UNIX only

Remarks **WinZoomPQG** displays a zoomed image of the area specified. If the PQG window has the aspect ratio attribute set to fixed, the *y* size of the rectangle will be adjusted as needed to maintain the aspect ratio.

To zoom out (that is, to redisplay the entire graph in the PQG window), use:

```
WinZoomPQG(whandle, 0, 0, 0, 0) ;
```

WinZoomPQG provides the same capabilities as the “Zoom” pop-up menu available in the PQG windows.

For more details, consult the PQG graphics section of the manual.

WinZoomPQG is supported only for PQG windows.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

writer

writer

Purpose Writes a matrix to a GAUSS data set.

Format `y = writer(fh,x);`

Input

<i>fh</i>	handle of the file that data is to be written to.
<i>x</i>	NxK matrix.

Output

<i>y</i>	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 to nearest 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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

xlabel

Purpose	Sets a label for the X axis.
Library	pgraph
Format	<code>xlabel(<i>str</i>);</code>
Input	<i>str</i> string, the label for the X axis.
Source	pgraph.src
See also	title, ylabel, zlabel

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

xpnd

xpnd

Purpose	Expands a column vector into a symmetric matrix.		
Format	$x = \text{xpnd}(v);$		
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.	
		$M = ((-1 + \text{sqrt}(1+8*K))/2)$	

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);
```

	1
	2
	3
	4
$x =$	5
	6
	7
	8
	9
	10

$$y = \begin{pmatrix} 1 & 2 & 4 & 7 \\ 2 & 3 & 5 & 8 \\ 4 & 5 & 6 & 9 \\ 7 & 8 & 9 & 10 \end{pmatrix}$$

See also `vech`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

xtics

xtics

Purpose Sets and fixes scaling, axes numbering and tick marks for the X axis.

Library pgraph

Format **xtics**(*min,max,step,minordiv*);

Input

<i>min</i>	scalar, the minimum value.
<i>max</i>	scalar, the maximum value.
<i>step</i>	scalar, the value between major tick marks.
<i>minordiv</i>	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.

Library `pgraph`

Format `xy(x,y) ;`

Input

<i>x</i>	Nx1 or NxM matrix. Each column contains the X values for a particular line.
<i>y</i>	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`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

xyz

xyz

Purpose	Graphs X vs. Y vs. Z using Cartesian coordinates.		
Library	pgraph		
Format	xyz(<i>x,y,z</i>) ;		
Input	<i>x</i>	Nx1 or NxK matrix. Each column contains the X values for a particular line.	
	<i>y</i>	Nx1 or NxK matrix. Each column contains the Y values for a particular line.	
	<i>z</i>	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		
See also	xy, surface, volume, view		

ylabel

Purpose	Sets a label for the Y axis.
Library	pgraph
Format	<code>ylabel(<i>str</i>);</code>
Input	<i>str</i> string, the label for the Y axis.
Source	pgraph.src
See also	<code>title</code> , <code>xlabel</code> , <code>zlabel</code>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ytics

ytics

Purpose Sets and fixes scaling, axes numbering and tick marks for the Y axis.

Library **pgraph**

Format **ytics**(*min,max,step,minordiv*);

Input

<i>min</i>	scalar, the minimum value.
<i>max</i>	scalar, the maximum value.
<i>step</i>	scalar, the value between major tick marks.
<i>minordiv</i>	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**

zeros

Purpose	Creates a matrix of zeros.
Format	$y = \text{zeros}(r,c);$
Input	r scalar, the number of rows. c scalar, the number of columns.
Output	y RxC matrix of zeros.
Remarks	This is faster than ones . Noninteger arguments will be truncated to an integer.
Example	$y = \text{zeros}(3,2);$ 0.000000 0.000000 $y =$ 0.000000 0.000000 0.000000 0.000000
See also	ones, eye

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

zlabel

zlabel

Purpose	Sets a label for the Z axis.
Library	pgraph
Format	<code>zlabel(<i>str</i>);</code>
Input	<i>str</i> string, the label for the Z axis.
Source	pgraph.src
See also	<code>title</code> , <code>xlabel</code> , <code>ylabel</code>

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

ztics

Purpose Sets and fixes scaling, axes numbering and tick marks for the Z axis.

Library pgraph

Format `ztics(min,max,step,minordiv);`

Input

<i>min</i>	scalar, the minimum value.
<i>max</i>	scalar, the maximum value.
<i>step</i>	scalar, the value between major tick marks.
<i>minordiv</i>	scalar, the number of minor subdivisions. If this function is used with contour , contour labels will be placed every <i>minordiv</i> 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

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

Obsolete Commands **A**

The following commands will no longer be supported, therefore should not be used when creating new programs. These commands will be left in for use in legacy code.

disable

disable

Purpose Disables the invalid operation interrupt of the numeric processor. This affects the way missing values are handled in most calculations.

Format `disable;`

Portability **DOS only**
In all others, invalid operation exception is always disabled.

Remarks When **disable** is in effect, missing values will be allowed in most calculations. A missing value is a special floating point encoding which the numeric processor considers a NaN (Not A Number). **disable** allows missing values to pass through most calculations unchanged; i.e., a number plus a missing value is a missing value.

The default when GAUSS is started is to have the program crash when missing values are encountered or when any operation sets the numeric processor's invalid operation exception.

If **disable** is on, these operations will return a NaN, and the program will continue. This can complicate debugging for programs that do not need to handle missing values, because the program may proceed far beyond the point that NaN's are created before it actually crashes.

The opposite of **disable** is **enable**, which is the default. If **enable** is on, the program will terminate with an "Invalid floating point operation" error message.

The following operators are specially designed to handle missing values and are not affected by the **disable/enable** commands: *b/a* (matrix division when *a* is not square and neither *a* nor *b* is scalar), **counts**, **ismiss**, **maxc**, **maxindc**, **minc**, **minindc**, **miss**, **missex**, **missrv**, **moment**, **packr**, **scalmiss**, **sortc**.

ndpcntrl can be used to get and reset the numeric processor control word, so it is more flexible than **enable/disable**.

See also `enable`, `miss`, `missrv`, `ndpchkl`, `ndpcntrl`

editm

Purpose	Edits a matrix. See medit for a full-window matrix editor.
Format	$y = \text{editm}(x);$
Input	x any legal expression that returns a matrix.
Output	y edited matrix.
Portability	DOS only π and e can be entered with ALT+P and ALT+E.
Remarks	A matrix editor is invoked when the editm function is called. This editor allows you to move around the matrix you are editing and make changes. (This editor is also used by the con function.) When editm appears in a program, the following will appear in the window: $[1,1] = 1.2361434675434E+002 \ ?$ The number after the equal sign is the [1,1] element of the matrix being edited. There are two general ways to move around the matrix. First, you can simply type numbers separated by commas, spaces, or carriage returns. The editor will automatically move you left to right and down through the matrix. That is, you will first go across the first row left to right, then across the second row, and so on. When you reach the last element in the matrix, you will automatically cycle back to the first element. When this occurs, the editor's prompt will again be displayed in the window. To exit the editor, type a semicolon. If a semicolon is typed after you have entered a number, that number will be saved. The second general way to move around the matrix is to use the cursor keys. The left and right cursor keys move you back and forth along rows. The up and down cursor keys move you up and down in a column. When you come to the end of a row or column, movement is left and up, or right and down. If, for instance, you are moving left to right along a row using the right cursor key, you will move down to the beginning of the next row when you come to the end of the row you are in. GAUSS will beep at you if you try to move outside the matrix.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

editm

The question mark you see in the window is the matrix editor's prompt. If you want to change the number that is in the [1,1] position of the matrix, type in the number you want. If you do not want to change anything, you can move to a different element by pressing the appropriate cursor key. If you start to type a number and press a cursor key, the number will not be saved. The BACKSPACE key will delete characters to the left of the cursor.

If you type in a space, comma, or carriage return before you type a number, nothing will happen. GAUSS will wait until you type a number or press a cursor key to move to a new element.

Numbers can be entered in scientific notation. The syntax is: $dE+n$ or $dE-n$, where d is a number and n is an integer power of 10. Thus, 1E+10, 1.1e-4, 1100E+1 are all legal.

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. For example, 1.2+23, 8.56i, 3-2.1i, -4.2e+6i, and 1.2e-4-4.5e+3i are all legal.

An **editm** function can appear anywhere in an expression where any other function can appear. Thus, for instance, the following statements are legal:

```
y = x*editm(z);
y = sqrt(editm(x));
```

Example

```
y = editm(ones(2,2));
[1,1] = 1.0000000000000E+000 ? 2 3 4 5[space]
[1,1] = 2.0000000000000E+000 ? 6 7 8 9[;]

y = 6.000000 7.000000
    8.000000 9.000000
```

In this example, a 2x2 matrix of 1's is edited. Spaces are used to separate the numbers as they are entered. Note that after 4 numbers have been entered, the editor has cycled back to the [1,1] position in the matrix. When this occurs, the prompt appears in the window. This time, after 4 more numbers have been entered, a semicolon is entered (as indicated in the brackets). This causes the editor to stop and the edited matrix to be returned.

See also `con`, `medit`

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

enable

enable

Purpose Enables the invalid operation interrupt of the numeric processor. This affects the way missing values are handled in most calculations.

Format `enable;`

Portability **DOS only**
In all others, the invalid operation exception is always disabled.

Remarks When **enable** is in effect, missing values will not be allowed in most calculations. A missing value is a special floating point encoding which the numeric processor considers a NaN (Not A Number).

The default, when GAUSS is started, is to have the program crash when missing values are encountered or when any operation sets the numeric processor's invalid operation exception.

If **enable** is on, these operations will cause the program to terminate with an **Invalid floating point operation** error message.

The opposite of **enable** is **disable**. If **disable** is on, these operations will return a NaN and the program will continue. This can complicate debugging for programs that do not need to handle missing values, because the program may proceed far beyond the point that NaN's are created before it actually crashes.

The following operators are specially designed to handle missing values and are not affected by the **disable/enable** commands: *b/a* (matrix division when *a* is not square), **counts**, **issmiss**, **maxc**, **maxindc**, **minc**, **minindc**, **miss**, **missex**, **missrv**, **moment**, **packr**, **scalmiss**, **sortc**.

ndpcntrl can be used to get and reset the numeric processor control word, and is more flexible than **enable/disable**.

See also `ndpcntrl`, `disable`, `miss`, `missrv`, `ndpch`

files

Purpose Returns a matrix of file names that match a specification.

Format `y = files(n, a);`

Input

<i>n</i>	string, file specification. Can include path. Wildcards are allowed in the file name.
<i>a</i>	scalar, the attribute to use in the search.

Output *y* character matrix of all filenames that match or scalar 0 if none are found. The path is not included.

DOS

Nx2 matrix with the names in column 1 and the extensions (including the ".") in column 2. If a file has no extension, the 2nd column will be null.

Windows, UNIX

NxK matrix with the names broken up into character elements, one name per row. K is determined by the size of the longest filename.

OS/2

Nx2 matrix with the names in column 1 and the extensions (including the ".") in column 2. If a name or extension is longer than 8 characters, an asterisk is inserted as the 8th character.

Portability UNIX, OS/2, Windows
files was written to work under DOS, with its 8x3 filenames; use **fileinfo** or **filesa** instead.

Remarks The attribute corresponds to the file attribute for each entry in the disk directory. For normal files an attribute of 0 is used. This will return the names of all matching "normal" files. The bits in the attribute byte have the following meaning:

0	Normal files
2	Hidden files (DOS, Windows, OS/2)
4	System files (DOS, Windows, OS/2)
8	Volume Label (DOS only)

files**16** Subdirectories

The values above are added together to specify different types of files to be searched for. Therefore 6 would specify hidden+system.

If the attribute is set for hidden, system, or subdirectory entries then it will be an inclusive search. All normal entries plus all entries matching the specified attributes will be returned.

If the attribute is set for the volume label, it will be an exclusive search and only the volume label will be returned. Volume labels look like filenames and have a period before the 9th character.

Example `y = files("ch*.*",0);`

In this example all normal files listed in the current directory that begin with "ch" will be returned.

`y = files("ch*.*",0);`

In this example all normal files listed in the current directory that begin with "ch" but do not have an extension will be returned.

`y = files("ch*.*",2+4+16);`

In this example all normal, hidden, and system files and all subdirectories listed in the current directory that begin with "ch" will be returned.

```
proc exist(filename);  
    retp(not files(filename,0) $== 0);  
endp;
```

This procedure will return 1 if the file exists or 0 if not.

See also `dos`, `fileinfo`, `filesa`

lprint on, lprint off

Purpose	These commands switch the automatic line printer mode on and off.
Format	<pre>lprint on; lprint off;</pre>
Portability	DOS only
Remarks	After the lprint on command is encountered, the results of any global assignment statements will be printed on the standard printer. Assignments to local matrices or strings will not be printed. The name and dimensions of the resulting matrix will also be printed. If the result being assigned will be inserted into a submatrix of the target matrix, the name will be followed by an empty set of square brackets. The lprint off command switches this feature off. These commands are similar to the print on and print off commands, except that they control printing to the printer instead of to the window or the auxiliary output.
Example	<pre>y = rndn(1000,3); lprint on; s = stdc(y)'; m = meanc(y)'; mm = diag(y'y)'; lprint off;</pre> Produces on the printer: <pre>S[1,3]= 1.021418 1.034924 1.040514 M[1,3]= -0.047845 0.007045 -0.035395 MM[3,1]= 1045.583840 1071.118064 1083.923128</pre>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

lprint on, lprint off

In this example, a large (1000x3) matrix of random numbers is created, and some statistics are calculated and printed out.

See also **print on**

a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x y z

lpwidth

Purpose	Specifies the width of the printer.
Format	<code>lpwidth n;</code>
Portability	DOS only
Remarks	<i>n</i> is a scalar which specifies the width of the printer in columns (characters). That is, after printing <i>n</i> characters on a line, GAUSS will send a carriage return and a line feed, so that the print head will move to the beginning of the next line. 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. <i>n</i> may be any scalar-valued expression. Nonintegers will be truncated to an integer. The default is 80 columns. Note: This does not send control characters to the printer to automatically switch the mode of the printer to a different character pitch because each printer is different. This only controls the frequency of carriage return/line feed sequences.
Example	<code>lpwidth 132;</code> This statement will change the printer width to 132 columns.
See also	<code>lprint</code> , <code>lpos</code> , <code>outwidth</code>

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ndpchk

ndpchk

Purpose	Examines the status of the math coprocessor and checks whether any exceptions have been generated.	
Format	$y = \text{ndpchk}(\text{mask});$	
Input	<i>mask</i>	scalar mask value used to test various bits in the math coprocessor status word.
Output	<i>y</i>	scalar, the result of a bitwise logical AND of the status word and the mask.

Portability	DOS only
	Exceptions are masked in UNIX, OS/2, and Windows, so ndpchk will never see them.

Remarks Relevant values that can be used to check various exceptions are:

- 1 Invalid Operation
- 2 Denormalized Operand
- 4 Zero Divide
- 8 Overflow
- 16 Underflow

For example, suppose you want your program to check to see if an underflow has occurred during a critical calculation. The statement:

```
y = ndpchk(16);
```

will return 16 if the math coprocessor has detected an underflow and zero otherwise. It performs a bitwise logical AND using the argument and the math coprocessor status flags and returns the result.

This does not test the math coprocessor status word directly, but tests a system variable in GAUSS where we keep track of the flags that have been set by ORing the status word with the variable.

The values can be added together to test more than one flag in a single call. For example:

```
if ndpchk(24);
    gosub error;
```

```
endif;
```

This code would call the subroutine `error` if either overflow or underflow had occurred in a program.

GAUSS keeps track of all the exception flags in the math coprocessor as it executes. All exceptions are supported with interrupt handling code. Zero Divide and Invalid Operation are enabled by default. **ndpcntrl** is used to get and set the control word. You can also enable or disable the Invalid Operation interrupt with the **enable** and **disable** commands. For example, if you want to add two matrices together that contain the missing value codes, you will have to disable the Invalid Operation interrupt or the program will stop and print an error message. If exceptions are generated during the execution of a program, a report will be given when GAUSS drops to command level.

You can refer to the literature available from Intel Corp. for complete information on coprocessor exceptions. This is the best reference for those who want to understand how the math coprocessor works. (See the *iAPX 86/88,186/188 User's Manual*, Intel Corp., 1983, available from Intel.)

Another excellent reference is: Brady, Robert J. *Applications and Programming for the IBM PC and Other PCs*, 1983.

See also **ndpclex**

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ndpclex

ndpclex

Purpose Clears the math coprocessor exception flags.

Format `ndpclex;`

Input None

Output The flags noted above are cleared.

Portability DOS only

Remarks This will prevent the exceptions message that can occur when GAUSS drops to interactive level.

See also `error`, `scalerr`, `trapchk`, `trap`, `ndpchk`

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

ndpcntrl

Purpose Gets and sets the numeric processor control word.

Format `y = ndpcntrl(new, mask);`

Input

new scalar, integer in the range 0-65535 containing the new bit values for the control word.

mask scalar, integer in the range 0-65535. If a bit in *mask* is 1, the corresponding bit in the control word will be set to the corresponding bit in *new*. If a bit in *mask* is 0, the corresponding bit in the control word will be left as is.

Output *y* scalar, the new control word.

Portability DOS only

Remarks We assume familiarity with the 80x87 family of math coprocessors. Here are the mask values for the various bits in the control word in hexadecimal:

0x003f	Interrupt Exception Masks
0x0001	invalid
0x0002	denormal
0x0004	zero divide
0x0008	overflow
0x0010	underflow
0x0020	inexact (precision)
0x1000	Infinity Control
0x1000	affine
0x0000	projective
0x0c00	Rounding Control
0x0c00	chop
0x0800	up
0x0400	down
0x0000	near

ndpcntrl

0x0300	Precision Control
0x0000	24 bits
0x0200	53 bits
0x0300	64 bits

For those of you who are fluent in C:

```
new_control = ((old_control & ~mask) | (new
                mask))
```

The bold values above are the mask values in hexadecimal for the various functions that are controlled by the NDP control word.

The inexact result exception mask is set whenever rounding occurs. For example:

```
y = 1/3;
```

This is not a very useful exception to unmask in most GAUSS programs.

Changing these values will not have a completely global effect because some functions locally change control word values.

Example

```
oldtrol = ndpcntrl(0,0);
call ndpcntrl(0x0800,0x0c00);
.
.
.
call ndpcntrl(oldtrol,0xffff);
```

In the example above, the original control word value is retrieved in the variable **oldtrol**. Nothing in the control word can be changed if the mask value is zero. In the second line, the rounding control is set to round up. In the last line, the original control word value is restored.

ndpcntrl

```
oldtrol = ndpcntrl(0,0);  
call ndpcntrl(0x0004,0x0004);  
.  
.  
.  
call ndpcntrl(oldtrol,0xffff);
```

In the example above, the original control word value is retrieved in the variable **oldtrol**. Nothing in the control word can be changed if the mask value is zero. In the second line, the zero divide exception is masked. In the last line, the original control word value is restored.

```
call ndpcntrl(0x0001+0x0020,0x003f);
```

In this example, invalid operation and inexact result are masked. All other exceptions are unmasked.

```
call ndpcntrl(0x0004,0x0004+0x0001);
```

In this example, zero divide is masked and invalid operation is unmasked. All other exceptions are left as is because the mask value allowed only these two bits to be changed.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

setvmode

setvmode

Purpose Sets video mode.

Format $y = \text{setvmode}(\text{mode});$

Input *mode* scalar, video mode to set.

-2	get current video mode
-1	hardware default mode
0	40 x 25 text, 16 grey
1	40 x 25 text, 16/8 color
2	80 x 25 text, 16 grey
3	80 x 25 text, 16/8 color
4	320 x 200, 4 color
5	320 x 200, 4 grey
6	640 x 200, BW
7	80 x 25 text, BW
8	720 x 348, BW Hercules
13	320 x 200, 16 color
14	640 x 200, 16 color
15	640 x 350, BW
16	640 x 350, 4 or 16 color
17	640 x 480, BW
18	640 x 480, 16 color
19	320 x 200, 256 color

Output *y* returns an 8x1 vector:

[1]	number of pixels in <i>x</i> axis
[2]	number of pixels in <i>y</i> axis
[3]	number of text columns
[4]	number of text rows
[5]	number of actual colors
[6]	number of bits per pixel
[7]	number of video pages
[8]	video mode

setvmode

If the mode requested is not supported by the hardware, this function will return a scalar zero.

If the argument (*mode*) is -2, the video mode will not be changed and the returned vector will reflect the current mode.

Portability**UNIX, OS/2, Windows**

setvmode affects the active graphic panel. This command forces the active graphic panel into a DOS emulation mode, and is supported for backwards compatibility only. Graphic panel size, resolution, font and/or colormap may be changed to accomodate the requested video mode. Not supported for graphic panel 1 and TTY graphic panels.

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

x y z

Colors Appendix B

0 Black	8 Dark Grey
1 Blue	9 Light Blue
2 Green	10 Light Green
3 Cyan	11 Light Cyan
4 Red	12 Light Red
5 Magenta	13 Light Magenta
6 Brown	14 Yellow
7 Grey	15 White

