GAUSS™

Language Reference

Part Numbers: 005500, 005501
Version 8.0
Documentation Revision: 860 June 27, 2007

Contents

1 Introduction

1.1 Product Overview 1-1
1.2 Documentation Conventions 1-2

2 Getting Started

2.1 Installation Under UNIX/Linux 2-1
2.2 Installation Under Windows 2-2
2.2.1 Machine Requirements 2-2
2.2.2 Installation from Download 2-2
2.2.3 Installation from CD 2-3

3 Using the Command Line Interface

3.1 Viewing Graphics 3-2
3.2 Interactive Commands 3-2
3.2.1 quit 3-2
3.2.2 ed 3-3
3.2.3 browse 3-3
3.2.4 config 3-3
3.3 Debugging 3-4
3.3.1 General Functions 3-5
3.3.2 Listing Functions 3-5
3.3.3 Execution Functions 3-5
3.3.4 View Commands 3-6
3.3.5 Breakpoint Commands 3-7

4 Introduction to the Windows Interface

4.1 **GAUSS** Menus 4-1
4.1.1 File Menu 4-1

4.1.2 Edit Menu 4-4
4.1.3 View Menu 4-5
4.1.4 Configure Menu 4-6
4.1.5 Run Menu 4-6
4.1.6 Debug Menu 4-7
4.1.7 Tools Menu 4-8
4.1.8 Window Menu 4-9
4.1.9 Help Menu 4-10
4.1.10 **GAUSS** Toolbars 4-10
4.1.11 Main Toolbar 4-11
4.1.12 Working Directory Toolbar 4-12
4.1.13 Debug Toolbar 4-13
4.1.14 Window Toolbar 4-14
4.1.15 Status Bar 4-15
4.1.16 GAUSS Status 4-15

5 Using the Windows Interface

5.1 Using the **GAUSS** Edit Windows 5-1
5.1.1 Editing Programs 5-2
5.1.2 Using Bookmarks 5-2
5.1.3 Changing the Editor Properties 5-2
5.1.4 Using Keystroke Macros 5-2
5.1.5 Using Margin Functions 5-3
5.1.6 Editing with Split Views 5-3
5.1.7 Finding and Replacing Text 5-3
5.1.8 Running Selected Text 5-4
5.2 Using The Command Input - Output Window 5-4
5.2.1 Running Commands 5-4
5.2.2 Running Programs in Files 5-5
5.3 Using Source View 5-5
5.3.1 Source Tab 5-6
5.3.2 Symbols Tab 5-6

5.4 Using the Error Output Window 5-7
5.5 Using The Debugger 5-7
5.5.1 Starting and Stopping the Debugger 5-7
5.5.2 Using Breakpoints 5-8
5.5.3 Setting and Clearing Breakpoints 5-8
5.5.4 Stepping Through a Program 5-9
5.5.5 Viewing and Editing Variables 5-9
5.6 Customizing **GAUSS** 5-10
5.6.1 Preferences Dialog Box 5-10
5.6.2 Editor Properties 5-14
5.7 Using **GAUSS** Keyboard Assignments 5-15
5.7.1 Cursor Movement Keys 5-15
5.7.2 Edit Keys 5-16
5.7.3 Text Selection Keys 5-17
5.7.4 Command Keys 5-17
5.7.5 Function Keys 5-18
5.7.6 Menu Keys 5-19

6 Matrix Editor

6.1 Using the Matrix Editor 6-1
6.1.1 Editing Matrices 6-1
6.1.2 Viewing Variables 6-3
6.1.3 Matrix Editor Menu Bar 6-3

7 Library Tool

7.1 Using the Library Tool 7-1
7.1.1 Managing Libraries 7-1
7.1.2 Managing the Library Index 7-1
7.1.3 Managing Library Files 7-3

8 GAUSS Source Browser

8.1 Using the Source Browser in TGAUSS 8-1
8.2 Using the Source Browser in **GAUSS** 8-2
8.2.1 Opening Files From the Source Browser 8-4
8.2.2 Source Browser Keyboard Controls 8-4

9 GAUSS Help

9.1 Help Menu 9-1
9.2 Context-Sensitive Help 9-1
9.3 SHIFT+F1 Support 9-2
9.4 CTRL+F1 Support 9-2
9.5 ToolTips 9-3
9.6 Other Help 9-3

10 Language Fundamentals

10.1 Expressions 10-1
10.2 Statements 10-2
10.2.1 Executable Statements 10-3
10.2.2 Nonexecutable Statements 10-3
10.3 Programs 10-4
10.3.1 Main Section 10-4
10.3.2 Secondary Sections 10-5
10.4 Compiler Directives 10-5
10.5 Procedures 10-8
10.6 Data Types 10-9
10.6.1 Constants 10-9
10.6.2 Matrices 10-11
10.6.3 Sparse Matrices 10-18
10.6.4 N-dimensional Arrays 10-19
10.6.5 Strings 10-20
10.6.6 String Arrays 10-24

10.6.7 Character Matrices . . . 10-26
10.6.8 Date and Time Formats . . . 10-27
10.6.9 Special Data Types . . . 10-28
10.7 Operator Precedence . . . 10-30
10.8 Flow Control . . . 10-32
10.8.1 Looping . . . 10-32
10.8.2 Conditional Branching . . . 10-35
10.8.3 Unconditional Branching . . . 10-36
10.9 Functions . . . 10-37
10.10 Rules of Syntax . . . 10-38
10.10.1 Statements . . . 10-38
10.10.2 Case . . . 10-38
10.10.3 Comments . . . 10-38
10.10.4 Extraneous Spaces . . . 10-39
10.10.5 Symbol Names . . . 10-39
10.10.6 Labels . . . 10-39
10.10.7 Assignment Statements . . . 10-40
10.10.8 Function Arguments . . . 10-40
10.10.9 Indexing Matrices . . . 10-41
10.10.10 Arrays of Matrices and Strings . . . 10-42
10.10.11 Arrays of Procedures . . . 10-43

11 Operators

11.1 Element-by-Element Operators . . . 11-1
11.2 Matrix Operators . . . 11-4
11.2.1 Numeric Operators . . . 11-4
11.2.2 Other Matrix Operators . . . 11-8
11.3 Relational Operators . . . 11-9
11.4 Logical Operators . . . 11-13
11.5 Other Operators . . . 11-16
11.6 Using Dot Operators with Constants . . . 11-21
11.7 Operator Precedence . . . 11-22

12 Procedures and Keywords

12.1 Defining a Procedure . . . 12-2
12.1.1 Procedure Declaration . . . 12-3
12.1.2 Local Variable Declarations . . . 12-3
12.1.3 Body of Procedure . . . 12-4
12.1.4 Returning from the Procedure . . . 12-5
12.1.5 End of Procedure Definition . . . 12-5
12.2 Calling a Procedure . . . 12-6
12.3 Keywords . . . 12-7
12.3.1 Defining a Keyword . . . 12-7
12.3.2 Calling a Keyword . . . 12-8
12.4 Passing Procedures to Procedures . . . 12-9
12.5 Indexing Procedures . . . 12-10
12.6 Multiple Returns from Procedures . . . 12-11
12.7 Saving Compiled Procedures . . . 12-13

13 Sparse Matrices

13.1 Defining Sparse Matrices . . . 13-1
13.2 Creating and Using Sparse Matrices . . . 13-2
13.3 Sparse Support in Matrix Functions and Operators . . . 13-3
13.3.1 Return Types for Dyadic Operators . . . 13-4

14 N-Dimensional Arrays

14.1 Bracketed Indexing . . . 14-3
14.2 E×E Conformability . . . 14-5
14.3 Glossary of Terms . . . 14-5

15 Working with Arrays

15.1 Initializing Arrays . . . 15-1
15.1.1 areshape . . . 15-2

15.1.2 aconcat . . . 15-4
15.1.3 aeye . . . 15-6
15.1.4 arrayinit . . . 15-6
15.1.5 arrayalloc . . . 15-7
15.2 Assigning to Arrays . . . 15-8
15.2.1 index operator . . . 15-9
15.2.2 getArray . . . 15-12
15.2.3 getMatrix . . . 15-13
15.2.4 getMatrix4D . . . 15-13
15.2.5 getScalar3D, getScalar4D . . . 15-14
15.2.6 putArray . . . 15-15
15.2.7 setArray . . . 15-16
15.3 Looping with Arrays . . . 15-17
15.3.1 loopnextindex . . . 15-19
15.4 Miscellaneous Array Functions . . . 15-21
15.4.1 atranspose . . . 15-21
15.4.2 amult . . . 15-23
15.4.3 amean, amin, amax . . . 15-25
15.4.4 getDims . . . 15-27
15.4.5 getOrders . . . 15-27
15.4.6 arraytomat . . . 15-28
15.4.7 mattoarray . . . 15-28
15.5 Using Arrays with **GAUSS** functions . . . 15-28
15.6 A Panel Data Model . . . 15-32
15.7 Appendix . . . 15-35

16 Structures

16.1 Basic Structures . . . 16-1
16.1.1 Structure Definition . . . 16-1
16.1.2 Declaring an Instance . . . 16-2
16.1.3 Initializing an Instance . . . 16-3
16.1.4 Arrays of Structures . . . 16-4

16.1.5 Structure Indexing . . . 16-5
16.1.6 Saving an Instance to the Disk . . . 16-8
16.1.7 Loading an Instance from the Disk . . . 16-9
16.1.8 Passing Structures to Procedures . . . 16-9
16.2 Structure Pointers . . . 16-10
16.2.1 Creating and Assigning Structure Pointers . . . 16-10
16.2.2 Structure Pointer References . . . 16-11
16.2.3 Using Structure Pointers in Procedures . . . 16-13
16.3 Special Structures . . . 16-15
16.3.1 The DS Structure . . . 16-15
16.3.2 The PV Structure . . . 16-16
16.3.3 Miscellaneous PV Procedures . . . 16-20
16.3.4 Control Structures . . . 16-22
16.4 sqpSolvemt . . . 16-23
16.4.1 Input Arguments . . . 16-24
16.4.2 Output Argument . . . 16-27
16.4.3 Example . . . 16-29
16.4.4 The Command File . . . 16-30

17 Run-Time Library Structures

17.1 The PV Parameter Structure . . . 17-1
17.2 Fast Pack Functions . . . 17-6
17.3 The DS Data Structure . . . 17-7

18 Libraries

18.1 Autoloader . . . 18-1
18.1.1 Forward References . . . 18-2
18.1.2 The Autoloader Search Path . . . 18-3
18.2 Global Declaration Files . . . 18-9
18.3 Troubleshooting . . . 18-12
18.3.1 Using .dec Files . . . 18-13

19 Compiler

19.1 Compiling Programs 19-2
19.1.1 Compiling a File 19-2
19.2 Saving the Current Workspace 19-2
19.3 Debugging 19-3

20 File I/O

20.1 ASCII Files 20-3
20.1.1 Matrix Data 20-3
20.1.2 General File I/O 20-6
20.2 Data Sets 20-7
20.2.1 Layout 20-7
20.2.2 Creating Data Sets 20-8
20.2.3 Reading and Writing 20-8
20.2.4 Distinguishing Character and Numeric Data 20-9
20.3 **GAUSS** Data Archives 20-11
20.3.1 Creating and Writing Variables to GDA's 20-11
20.3.2 Reading Variables from GDA's 20-12
20.3.3 Updating Variables in GDA's 20-13
20.4 Matrix Files 20-13
20.5 File Formats 20-14
20.5.1 Small Matrix **v89** (Obsolete) 20-15
20.5.2 Extended Matrix **v89** (Obsolete) 20-16
20.5.3 Small String **v89** (Obsolete) 20-16
20.5.4 Extended String **v89** (Obsolete) 20-17
20.5.5 Small Data Set **v89** (Obsolete) 20-17
20.5.6 Extended Data Set **v89** (Obsolete) 20-19
20.5.7 Matrix **v92** (Obsolete) 20-20
20.5.8 String **v92** (Obsolete) 20-20
20.5.9 Data Set **v92** (Obsolete) 20-21
20.5.10 Matrix **v96** 20-22

20.5.11 Data Set **v96** 20-23
20.5.12 **GAUSS** Data Archive 20-24

21 Foreign Language Interface

21.1 Writing FLI Functions 21-2
21.2 Creating Dynamic Libraries 21-3

22 Data Transformations

22.1 Data Loop Statements 22-2
22.2 Using Other Statements 22-3
22.3 Debugging Data Loops 22-3
22.3.1 Translation Phase 22-3
22.3.2 Compilation Phase 22-3
22.3.3 Execution Phase 22-4
22.4 Reserved Variables 22-4

23 The GAUSS Profiler

23.1 Using the **GAUSS** Profiler 23-1
23.1.1 Collection 23-1
23.1.2 Analysis 23-2

24 Publication Quality Graphics

24.1 General Design 24-1
24.2 Using **Publication Quality Graphics** 24-2
24.2.1 Getting Started 24-2
24.2.2 Graphics Coordinate System 24-6
24.3 Graphic Panels 24-7
24.3.1 Tiled Graphic Panels 24-7
24.3.2 Overlapping Graphic Panels 24-7
24.3.3 Nontransparent Graphic Panels 24-8

24.3.4 Transparent Graphic Panels 24-8
24.3.5 Using Graphic Panel Functions 24-8
24.3.6 Inch Units in Graphic Panels 24-10
24.3.7 Saving Graphic Panel Configurations 24-10
24.4 Graphics Text Elements 24-10
24.4.1 Selecting Fonts 24-11
24.4.2 Greek and Mathematical Symbols 24-12
24.5 Colors 24-14
24.6 Global Control Variables 24-14

25 Time and Date

25.1 Time and Date Formats 25-2
25.2 Time and Date Functions 25-4
25.2.1 Timed Iterations 25-6

26 ATOG

26.1 Command Summary 26-1
26.2 Commands 26-3
26.3 Examples 26-12
26.4 Error Messages 26-15

27 Error Messages

28 Maximizing Performance

28.1 Library System 28-1
28.2 Loops 28-2
28.3 Memory Usage 28-3
28.3.1 Hard Disk Maintenance 28-4
28.3.2 CPU Cache 28-4

A Fonts

A.1 Simplex A-2
A.2 Simgrma A-3
A.3 Microb A-4
A.4 Complex A-5

B Reserved Words Appendix

C Singularity Tolerance Appendix

C.1 Reading and Setting the Tolerance C-2
C.2 Determining Singularity C-2

29 Command Reference Introduction

29.1 Documentation Conventions 29-2
29.2 Command Components 29-3
29.3 Using This Manual 29-4
29.4 Global Control Variables 29-5
29.4.1 Changing the Default Values 29-5
29.4.2 The Procedure **gausset** 29-6

30 Commands by Category

30.1 Mathematical Functions 30-1
30.2 Finance Functions 30-20
30.3 Matrix Manipulation 30-22
30.4 Sparse Matrix Handling 30-26
30.5 N-Dimensional Array Handling 30-27
30.6 Structures 30-29
30.7 Data Handling (I/0) 30-30
30.8 Compiler Control 30-39
30.9 Program Control 30-40

30.10 OS Functions and File Management 30-45
30.11 Workspace Management 30-46
30.12 Error Handling and Debugging 30-47
30.13 String Handling 30-47
30.14 Time and Date Functions 30-50
30.15 Console I/O 30-52
30.16 Output Functions 30-52
30.17 Graphics 30-54

31 Command Reference

D Obsolete Commands

E Colors

Index

List of Figures

4.1 **GAUSS** Graphical User Interface 4-2
4.2 Main Toolbar 4-11
4.3 Working Directory Toolbar 4-12
4.4 Debug Toolbar 4-13
4.5 Window Toolbar 4-14
4.6 Status Bar 4-15
6.1 Matrix Editor 6-2
7.1 Library Tool 7-2
8.1 Source Browser 8-3
16.1 Structure tree for **e1** 16-7

Command Reference Introduction 29

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

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

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

determinants, matrix inverses, decompositions, eigenvalues and eigenvectors, and condition numbers.

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

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

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

29.1 Documentation Conventions

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

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

Text Style	Use	Example

`monospace`	code example
	
	- or -
	
	filename, path, etc.
		`if scalerr(cm);`
		`  cm = inv(x);`
		`endif;`
		
		"...is located in the `examples` subdirectory..."

`monospace bold`	reference to a **GAUSS** command or other programming element within a narrative paragraph	"...as explained under **`create`**..."
SMALL CAPS	reference to section of the manual	"...see OPERATOR PRECEDENCE, Section 11.7..."

29.2 Command Components

The following list describes each of the components used in the COMMAND REFERENCE, Chapter 31.

PURPOSE — Describes what the command or function does.

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

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

FORMAT — Illustrates the syntax of the command or function.

INPUT — Describes the input parameters of the function.

GLOBAL INPUT — Describes the global variables that are referenced by the function.

OUTPUT — Describes the return values of the function.

GLOBAL OUTPUT	Describes the global variables that are updated by the function.
PORTABILITY	Describes differences under various operating systems.
REMARKS	Explanatory material pertinent to the command.
EXAMPLE	Sample code using the command or function.
SOURCE	The source file in which the function is defined, if applicable.
GLOBALS	Global variables that are accessed by the command.
SEE ALSO	Other related commands.
TECHNICAL NOTES	Technical discussion and reference source citations.
REFERENCES	Reference material citations.

29.3 Using This Manual

Users who are new to **GAUSS** should make sure they have familiarized themselves with Language Fundamentals, Chapter 10, before proceeding here. That chapter contains the basics of **GAUSS** programming.

In all, there are over 700 routines described in this GAUSS Language Reference. We suggest that new **GAUSS** users skim through Chapter 30, and then browse through Chapter 31, the main part of this manual. Here, users can familiarize themselves with the kinds of tasks that **GAUSS** can handle easily.

Chapter 30 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 31.

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 31 under the discussion of that function.

29.4 Global Control Variables

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

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

29.4.1 Changing the Default Values

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

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

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

so it reads:

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

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

```
__output = 1;
```

similarly.

29.4.2 The Procedure gausset

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

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

```
gausset;
```

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

30 Commands by Category

30.1 Mathematical Functions

Scientific Functions

abs	Returns absolute value of argument.
arccos	Computes inverse cosine.
arcsin	Computes inverse sine.
atan	Computes inverse tangent.
atan2	Computes angle given a point *x,y*.
besselj	Computes Bessel function, first kind.
bessely	Computes Bessel function, second kind.
boxcox	Computes the Box-Cox function.
cos	Computes cosine.

cosh	Computes hyperbolic cosine.
curve	Computes a one-dimensional smoothing curve.
digamma	Computes the digamma function.
exp	Computes the exponential function of x.
fmod	Computes the floating-point remainder of x/y.
gamma	Computes gamma function value.
gammai	Compute the inverse incomplete gamma function.
ln	Computes the natural log of each element.
lnfact	Computes natural log of factorial function.
log	Computes the $\log_{10}$ of each element.
mbesseli	Computes modified and exponentially scaled modified Bessels of the first kind of the n^{th} order.
nextn, nextnevn	Returns allowable matrix dimensions for computing FFT's.
optn, optnevn	Returns optimal matrix dimensions for computing FFT's.
pi	Returns π.
polar	Graphs data using polar coordinates.
sin	Computes sine.
sinh	Computes the hyperbolic sine.
spline	Computes a two-dimensional interpolatory spline.
sqrt	Computes the square root of each element.
tan	Computes tangent.
tanh	Computes hyperbolic tangent.

tocart	Converts from polar to Cartesian coordinates.
topolar	Converts from Cartesian to polar coordinates.
trigamma	Computes trigamma function.

All trigonometric functions take or return values in radian units.

Differentiation and Integration

gradMT	Computes numerical gradient.
gradMTm	Computes numerical gradient with mask.
gradp	Computes first derivative of a function.
hessMT	Computes numerical Hessian.
hessMTg	Computes numerical Hessian using gradient procedure.
hessMTgw	Computes numerical Hessian using gradient procedure with weights.
hessMTm	Computes numerical Hessian with mask.
hessMTmw	Computes numerical Hessian with mask and weights.
hessMTw	Computes numerical Hessian with weights.
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.
inthp1	Integrates a user-defined function over an infinite interval.

inthp2	Integrates a user-defined function over the $[a,+\infty)$ interval.
inthp3	Integrates a user-defined function over the $[a,+\infty)$ interval that is oscillatory.
inthp4	Integrates a user-defined function over the $[a,b]$ interval.
inthpControlCreate	Creates default **inthpControl** structure.
intquad1	Integrates a 1-dimensional function.
intquad2	Integrates a 2-dimensional function over 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.
band	Extracts bands from a symmetric banded matrix.

bandchol	Computes the Cholesky decomposition of a positive definite banded matrix.
bandcholsol	Solves the system of equations $Ax = b$ for x, given the lower triangle of the Cholesky decomposition of a positive definite banded matrix A.
bandltsol	Solves the system of equations $Ax = b$ for x, where A is a lower triangular banded matrix.
bandrv	Creates a symmetric banded matrix, given its compact form.
bandsolpd	Solves the system of equations $Ax = b$ for x, where A is a positive definite banded matrix.
chol	Computes Cholesky decomposition, $X = Y'Y$.
choldn	Performs Cholesky downdate on an upper triangular matrix.
cholsol	Solves a system of equations given the Cholesky factorization of a matrix.
cholup	Performs Cholesky update on an upper triangular matrix.
cond	Computes condition number of a matrix.
crout	Computes Crout decomposition, $X = LU$ (real matrices only).
croutp	Computes Crout decomposition with row pivoting (real matrices only).
det	Computes determinant of square matrix.
detl	Computes determinant of decomposed matrix.
hess	Computes upper Hessenberg form of a matrix (real matrices only).
inv	Inverts a matrix.
invpd	Inverts a positive definite matrix.
invswp	Computes a generalized sweep inverse.

lapeighb	Computes eigenvalues only of a real symmetric or complex Hermitian matrix selected by bounds.
lapeighi	Computes eigenvalues only of a real symmetric or complex Hermitian matrix selected by index.
lapeighvb	Computes eigenvalues and eigenvectors of a real symmetric or complex Hermitian matrix selected by bounds.
lapeighvi	Computes selected eigenvalues and eigenvectors of a real symmetric or complex Hermitian matrix.
lapgeig	Computes generalized eigenvalues for a pair of real or complex general matrices.
lapgeigh	Computes generalized eigenvalues for a pair of real symmetric or Hermitian matrices.
lapgeighv	Computes generalized eigenvalues and eigenvectors for a pair of real symmetric or Hermitian matrices.
lapgeigv	Computes generalized eigenvalues, left eigenvectors, and right eigenvectors for a pair of real or complex general matrices.
lapgschur	Computes the generalized Schur form of a pair of real or complex general matrices.
lapgsvdcst	Computes the generalized singular value decomposition of a pair of real or complex general matrices.
lapgsvds	Computes the generalized singular value decomposition of a pair of real or complex general matrices.
lapgsvdst	Computes the generalized singular value decomposition of a pair of real or complex general matrices.
lapsvdcusv	Computes the singular value decomposition a real or complex rectangular matrix, returns compact *u* and *v*.
lapsvds	Computes the singular values of a real or complex rectangular matrix.

lapsvdusv	Computes the singular value decomposition a real or complex rectangular matrix.
lu	Computes LU decomposition with row pivoting (real and complex matrices).
null	Computes orthonormal basis for right null space.
null1	Computes orthonormal basis for right null space.
orth	Computes orthonormal basis for column space *x*.
pinv	Generalized pseudo-inverse: Moore-Penrose.
pinvmt	Generalized pseudo-inverse: Moore-Penrose.
qqr	QR decomposition: returns Q_1 and R.
qqre	QR decomposition: returns Q_1, R and a permutation vector, E.
qqrep	QR decomposition with pivot control: returns Q_1, R and E.
qr	QR decomposition: returns R.
qre	QR decomposition: returns R and E.
qrep	QR decomposition with pivot control: returns R and E.
qrsol	Solves a system of equations $Rx = b$ 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.
qtyr	QR decomposition: returns $Q'Y$ and R.
qtyre	QR decomposition: returns $Q'Y$,R and E.
qtyrep	QR decomposition with pivot control: returns $Q'Y$,R and E.
qyr	QR decomposition: returns QY and R.
qyre	QR decomposition: returns QY, R and E.

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

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

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

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

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

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

Eigenvalues

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

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

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

Polynomial Operations

polychar	Computes characteristic polynomial of a square matrix.
polyeval	Evaluates polynomial with given coefficients.
polyint	Calculates 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.
ffti	Computes inverse 1- or 2-D FFT.
fftm	Computes multi-dimensional FFT.
fftmi	Computes inverse multi-dimensional FFT.
fftn	Computes 1- or 2-D FFT using prime factor algorithm.
rfft	Computes real 1- or 2-D FFT.
rffti	Computes inverse real 1- or 2-D FFT.
rfftip	Computes inverse real 1- or 2-D FFT from packed format FFT.
rfftn	Computes real 1- or 2-D FFT using prime factor algorithm.
rfftnp	Computes real 1- or 2-D FFT using prime factor algorithm, returns packed format FFT.
rfftp	Computes real 1- or 2-D FFT, returns packed format FFT.

Random Numbers

rndbeta	Computes random numbers with beta distribution.
rndcon	Changes constant of the LC random number generator.
rndgam	Computes random numbers with gamma distribution.
rndi	Returns random integers, $0 <= y < 2^{32}$.
rndKMbeta	Computes beta pseudo-random numbers.

rndKMgam	Computes gamma pseudo-random numbers.
rndKMi	Returns random integers, $0 <= y < 2^{32}$.
rndKMn	Computes standard normal pseudo-random numbers.
rndKMnb	Computes negative binomial pseudo-random numbers.
rndKMp	Computes Poisson pseudo-random numbers.
rndKMu	Computes uniform pseudo-random numbers.
rndKMvm	Computes von Mises pseudo-random numbers.
rndLCbeta	Computes beta pseudo-random numbers.
rndLCgam	Computes gamma pseudo-random numbers.
rndLCi	Returns random integers, $0 <= y < 2^{32}$.
rndLCn	Computes standard normal pseudo-random numbers.
rndLCnb	Computes negative binomial pseudo-random numbers.
rndLCp	Computes Poisson pseudo-random numbers.
rndLCu	Computes uniform pseudo-random numbers.
rndLCvm	Computes von Mises pseudo-random numbers.
rndmult	Changes multiplier of the LC random number generator.
rndn	Computes random numbers with Normal distribution.
rndnb	Computes random numbers with negative binomial distribution.
rndp	Computes random numbers with Poisson distribution.
rndseed	Changes seed of the LC random number generator.
rndu	Computes random numbers with uniform distribution.

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

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

Fuzzy Conditional Functions

dotfeq	Fuzzy . = =
dotfeqmt	Fuzzy . = =
dotfge	Fuzzy . >=
dotfgemt	Fuzzy . >=
dotfgt	Fuzzy . >
dotfgtmt	Fuzzy . >
dotfle	Fuzzy . <=
dotflemt	Fuzzy . <=
dotflt	Fuzzy . <
dotfltmt	Fuzzy . <
dotfne	Fuzzy ./ =
dotfnemt	Fuzzy ./ =
feq	Fuzzy = =
feqmt	Fuzzy = =
fge	Fuzzy >=
fgemt	Fuzzy >=
fgt	Fuzzy >
fgtmt	Fuzzy >
fle	Fuzzy <=

flemt	Fuzzy <=
flt	Fuzzy <
fltmt	Fuzzy <
fne	Fuzzy / =
fnemt	Fuzzy / =

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

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

Statistical Functions

acf	Computes sample autocorrelations.
astd	Computes the standard deviation of the elements across one dimension of an N-dimensional array.
astds	Computes the 'sample'standard deviation of the elements across one dimension of an N-dimensional array.
chiBarSquare	computes probability of chi-bar-square statistic
combinate	Computes combinations of *n* things
combinate	Computes combinations of *n* things taken *k* at a time.
combinated	Writes combinations of *n* things taken *k* at a time to a **GAUSS** data set.
ConScore	computes constrained score statistic and its probability

conv	Computes convolution of two vectors.
corrm	Computes correlation matrix of a moment matrix.
corrms	Computes sample correlation matrix of a moment matrix.
corrvc	Computes correlation matrix from a variance-covariance matrix.
corrx	Computes correlation matrix.
corrxs	Computes sample correlation matrix.
crossprd	Computes cross product.
design	Creates a design matrix of 0's and 1's.
dstat	Computes descriptive statistics of a data set or matrix.
dstatmt	Computes descriptive statistics of a data set or matrix.
dstatmtControlCreate	Creates default **dstatmtControl** structure.
gdaDStat	Computes descriptive statistics on multiple N×1 variables in a GDA.
gdaDStatMat	Computes descriptive statistics on a selection of columns in a variable in a GDA.
loess	Computes coefficients of locally weighted regression.
loessmt	Computes coefficients of locally weighted regression.
loessmtControlCreate	Creates default **loessmtControl** structure.
meanc	Computes mean value of each column of a matrix.
median	Computes medians of the columns of a matrix.
moment	Computes moment matrix ($x'x$) with special handling of missing values.
momentd	Computes moment matrix from a data set.

movingave	Computes moving average of a series.
movingaveExpwgt	Computes exponentially weighted moving average of a series.
movingaveWgt	Computes weighted moving average of a series.
numCombinations	Computes number of combinations of *n* things taken *k* at a time.
ols	Computes least squares regression of data set or matrix.
olsmt	Computes least squares regression of data set or matrix.
olsmtControlCreate	Creates default **olsmtControl** structure.
olsqr	Computes OLS coefficients using QR decomposition.
olsqr2	Computes OLS coefficients, residuals, and predicted values using QR decomposition.
olsqrmt	Computes OLS coefficients using QR decomposition.
pacf	Computes sample partial autocorrelations.
princomp	Computes principal components of a data matrix.
quantile	Computes quantiles from data in a matrix, given specified probabilities.
quantiled	Computes quantiles from data in a data set, given specified probabilities.
rndvm	Computes von Mises pseudo-random numbers.
stdc	Computes standard deviation of the columns of a matrix.
stdsc	Computes the 'sample' standard deviation of the elements in each column of a matrix.
toeplitz	Computes Toeplitz matrix from column vector.
varmall	Computes the log-likelihood of a Vector ARMA model.

varmares	Computes the residuals of a Vector ARMA model.
vcm	Computes a variance-covariance matrix from a moment matrix.
vcms	Computes a sample variance-covariance matrix from a moment matrix.
vcx	Computes a variance-covariance matrix from a data matrix.
vcxs	Computes a sample variance-covariance matrix from a data matrix.

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

Optimization and Solution

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

QNewtonmt	Minimizes an arbitrary function.
QNewtonmtControlCreate	Creates default **QNewtonmtControl** structure.
QNewtonmtOutCreate	Creates default **QNewtonmtOut** structure.
QProg	Solves the quadratic programming problem.
QProgmt	Solves the quadratic programming problem.
QProgmtInCreate	Creates an instance of a structure of type **QProgmtInCreate** with the **maxit** member set to a default value.
sqpSolve	Solves the nonlinear programming problem using a sequential quadratic programming method.
sqpSolveMT	Solves the nonlinear programming problem using a sequential quadratic programming method.
sqpSolveMTControlCreate	Creates an instance of a structure of type **sqpSolveMTcontrol** set to default values.
sqpSolveMTlagrangeCreate	Creates an instance of a structure of type **sqpSolveMTlagrange** set to default values.
sqpSolveMToutCreate	Creates an instance of a structure of type **sqpSolveMTout** set to default values.
sqpSolveSet	Resets global variables used by **sqpSolve** to default values.
utrisol	Computes the solution of $Ux = b$ where U is an upper triangular matrix.

Statistical Distributions

cdfbeta	Computes integral of beta function.
cdfbvn	Computes lower tail of bivariate Normal cdf.

cdfbvn2	Returns cdfbvn of a bounded rectangle.
cdfbvn2e	Returns cdfbvn of a bounded rectangle.
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.
cdfmvnce	Computes the complement of the multivariate Normal cumulative distribution function with error management
cdfmvne	Computes multivariate Normal cumulative distribution function with error management
cdfmvn2e	Computes the multivariate Normal cumulative distribution function with error management over the range [a,b]
cdfmvtce	Computes complement of multivariate Student's t cumulative distribution function with error management
cdfmvte	Computes multivariate Student's t cumulative distribution function with error management
cdfmvt2e	Computes multivariate Student's t cumulative distribution function with error management over [a,b]
cdfn	Computes integral of Normal distribution: lower tail, or cdf.
cdfn2	Computes interval of Normal cdf.
cdfnc	Computes complement of cdf of Normal distribution (upper tail).
cdfni	Computes the inverse of the cdf of the Normal distribution.

cdftc	Computes complement of cdf of *t*-distribution.
cdftci	Computes the inverse of the complement of the Student's t cdf.
cdftnc	Computes integral of noncentral *t*-distribution.
cdftvn	Computes lower tail of trivariate Normal cdf.
erf	Computes Gaussian error function.
erfc	Computes complement of Gaussian error function.
lncdfbvn	Computes natural log of bivariate Normal cdf.
lncdfbvn2	Returns log of cdfbvn of a bounded rectangle.
lncdfmvn	Computes natural log of multivariate Normal cdf.
lncdfn	Computes natural log of Normal cdf.
lncdfn2	Computes natural log of interval of Normal cdf.
lncdfnc	Computes natural log of complement of Normal cdf.
lnpdfmvn	Computes multivariate Normal log-probabilities.
lnpdfmvt	Computes multivariate Student's *t* log-probabilities.
lnpdfn	Computes Normal log-probabilities.
lnpdft	Computes Student's *t* log-probabilities.
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 $-\infty$.
machEpsilon	Returns the smallest number such that $1+eps>1$.
round	Rounds to the nearest integer.
trunc	Converts numbers to integers by truncating the fractional portion.

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

Each matrix element in memory requires 8 bytes of memory.

30.2 Finance Functions

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

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

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

30.3 Matrix Manipulation

Creating Vectors and Matrices

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

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

Matrices can also be loaded from an ASCII file, from a **GAUSS** matrix file, or from a **GAUSS** data set. (See FILE I/0, Chapter 20, for more information.)

Loading and Storing Matrices

asciiload	Loads data from a delimited ASCII text file into an N×1 vector.
dataload	Loads matrices, N-dimensional arrays, strings and string arrays from a disk file.
datasave	Saves matrices, N-dimensional arrays, strings and string arrays to a disk file.
load, loadm	Loads matrix from ASCII or matrix file.
loadd	Loads matrix from data set.
loadf	Loads function from disk file.
loadk	Loads keyword from disk file.
save	Saves symbol to disk file.
saved	Saves matrix to data set.

Size, Ranking, and Range

cols	Returns number of columns in a matrix.
colsf	Returns number of columns in an open data set.
counts	Returns number of elements of a vector falling in specified ranges.
countwts	Returns weighted count of elements of a vector falling in specified ranges.
cumprodc	Computes cumulative products of each column of a matrix.
cumsumc	Computes cumulative sums of each column of a matrix.
indexcat	Returns indices of elements falling within a specified range.
maxc	Returns largest element in each column of a matrix.

maxindc	Returns row number of largest element in each column of a matrix.
minc	Returns smallest element in each column of a matrix.
minindc	Returns row number of smallest element in each column of a matrix.
prodc	Computes the product of each column of a matrix.
rankindx	Returns rank index of N×1 vector. (Rank order of elements in vector).
rows	Returns number of rows in a matrix.
rowsf	Returns number of rows in an open data set.
sumc	Computes the sum of each column of a matrix.
sumr	Computes the sum of each row of a matrix.

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

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

Miscellaneous Matrix Manipulation

complex	Creates a complex matrix from two real matrices.
delif	Deletes rows from a matrix using a logical expression.
diag	Extracts the diagonal of a matrix.
diagrv	Puts a column vector into the diagonal of a matrix.
exctsmpl	Creates a random subsample of a data set, with replacement.
imag	Returns the imaginary part of a complex matrix.
indcv	Checks one character vector against another and returns the indices of the elements of the first vector in the second vector.

indnv	Checks one numeric vector against another and returns the indices of the elements of the first vector in the second vector.
intrsect	Returns the intersection of two vectors.
lowmat	Returns the main diagonal and lower triangle.
lowmat1	Returns a main diagonal of 1's and the lower triangle.
putvals	Inserts values into a matrix or N-dimensional array.
real	Returns the real part of a complex matrix.
reshape	Reshapes a matrix to new dimensions.
rev	Reverses the order of rows of a matrix.
rotater	Rotates the rows of a matrix, wrapping elements as necessary.
selif	Selects rows from a matrix using a logical expression.
setdif	Returns elements of one vector that are not in another.
shiftr	Shifts rows of a matrix, filling in holes with a specified value.
submat	Extracts a submatrix from a matrix.
subvec	Extracts an N×1 vector of elements from an N×K matrix.
trimr	Trims rows from top or bottom of a matrix.
union	Returns the union of two vectors.
upmat	Returns the main diagonal and upper triangle.
upmat1	Returns a main diagonal of 1's and the upper triangle.
vec	Stacks columns of a matrix to form a single column.
vech	Reshapes the lower triangular portion of a symmetric matrix into a column vector.
vecr	Stacks rows of a matrix to form a single column.
vget	Extracts a matrix or string from a data buffer constructed with **vput**.

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

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

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

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

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

30.4 Sparse Matrix Handling

denseToSp	Converts a dense matrix to a sparse matrix.
denseToSpRE	Converts a dense matrix to a sparse matrix using a relative epsilon.
packedToSp	Creates a sparse matrix from a packed matrix of non-zero values and row and column indices.
spCreate	Creates a sparse matrix from vectors of non-zero values, row indices, and column indices.
spDenseSubmat	Returns a dense submatrix of a sparse matrix.

spDiagRvMat	Inserts submatrices along the diagonal of a sparse matrix.
spEye	Creates a sparse identity matrix.
spGetNZE	Returns the non-zero values in a sparse matrix, as well as their corresponding row and column indices.
spGetNumNZE	Returns the number of non-zero elements in a sparse matrix.
spOnes	Generates a sparse matrix containing only ones and zeros
spSubmat	Returns a sparse submatrix of sparse matrix.
spToDense	Converts a sparse matrix to a dense matrix.
spTrTDense	Multiplies a sparse matrix transposed by a dense matrix.
spTScalar	Multiplies a sparse matrix by a scalar.
spZeros	Creates a sparse matrix containing no non-zero values.

30.5 N-Dimensional Array Handling

Creating Arrays

aconcat	Concatenates conformable matrices and arrays in a user-specified dimension.
aeye	Creates an N-dimensional array in which the planes described by the two trailing dimensions of the array are equal to the identity.
areshape	Reshapes a scalar, matrix, or array into an array of user-specified size.
arrayalloc	Creates an N-dimensional array with unspecified contents.
arrayinit	Creates an N-dimensional array with a specified fill value.
mattoarray	Converts a matrix to a type array.

Size, Ranking and Range

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

Setting and Retrieving Data in an Array

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

Miscellaneous Array Functions

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

30.6 Structures

dsCreate	Creates an instance of a structure of type **DS** set to default values.
loadstruct	Loads a structure into memory from a file on the disk.
pvCreate	Returns an initialized an instance of structure of type **PV**.
pvGetIndex	Gets row indices of a matrix in a parameter vector.
pvGetParNames	Generates names for parameter vector stored in structure of type **PV**.
pvGetParVector	Retrieves parameter vector from structure of type **PV**.
pvLength	Returns the length of a parameter vector.

pvList	Retrieves names of packed matrices in structure of type **PV**.
pvPack	Packs general matrix into a structure of type **PV** with matrix name.
pvPacki	Packs general matrix or array into a **PV** instance with name and index.
pvPackm	Packs general matrix into a structure of type **PV** with a mask and matrix name.
pvPackmi	Packs general matrix or array into a **PV** instance with a mask, name, and index.
pvPacks	Packs symmetric matrix into a structure of type **PV**.
pvPacksi	Packs symmetric matrix into a **PV** instance with matrix name and index.
pvPacksm	Packs symmetric matrix into a structure of type **PV** with a mask.
pvPacksmi	Packs symmetric matrix into a **PV** instance with a mask, matrix name, and index.
pvPutParVector	Inserts parameter vector into structure of type **PV**.
pvTest	Tests an instance of structure of type **PV** to determine if it is a proper structure of type **PV**.
pvUnpack	Unpacks matrices stored in a structure of type **PV**.
savestruct	Saves a matrix of structures to a file on the disk.

30.7 Data Handling (I/0)

Spreadsheets

SpreadsheetReadM	Reads and writes Excel files.
SpreadsheetReadSA	Reads and writes Excel files.

SpreadsheetWrite	Reads and writes Excel files.
xlsGetSheetCount	Gets the number of sheets in an Excel spreadsheet.
xlsGetSheetSize	Gets the size (rows and columns) of a specified sheet in an Excel spreadsheet.
xlsGetSheetTypes	Gets the cell format types of a row in an Excel spreadsheet.
xlsMakeRange	Builds an Excel range string from a row/column pair.
xlsreadm	Reads from an Excel spreadsheet, into a **GAUSS** matrix.
xlsreadsa	Reads from an Excel spreadsheet, into a **GAUSS** string array or string.
xlsWrite	Writes a **GAUSS** matrix, string, or string array to an Excel spreadsheet.
xlswritem	Writes a **GAUSS** matrix to an Excel spreadsheet.
xlswritesa	Writes a **GAUSS** string or string array to an Excel spreadsheet.

Text Files

fcheckerr	Gets the error status of a file.
fclearerr	Gets the error status of a file, then clears it.
fflush	Flushes a file's output buffer.
fgets	Reads a line of text from a file.
fgetsa	Reads lines of text from a file into a string array.
fgetsat	Reads lines of text from a file into a string array.
fgetst	Reads a line of text from a file.

fopen	Opens a file.
fputs	Writes strings to a file.
fputst	Writes strings to a file.
fseek	Positions the file pointer in a file.
fstrerror	Returns an error message explaining the cause of the most recent file I/O error.
ftell	Gets the position of the file pointer in a file.

GAUSS Data Archives

gdaAppend	Appends data to a variable in a GDA.
gdaCreate	Creates a GDA.
gdaDStat	Computes descriptive statistics on multiple N×1 variables in a GDA.
gdaDStatMat	Computes descriptive statistics on a selection of columns in a variable in a GDA.
gdaGetIndex	Gets the index of a variable in a GDA.
gdaGetName	Gets the name of a variable in a GDA.
gdaGetNames	Gets the names of all the variables in a GDA.
gdaGetOrders	Gets the orders of a variable in a GDA.
gdaGetType	Gets the type of a variable in a GDA.
gdaGetTypes	Gets the types of all the variables in a GDA.
gdaGetVarInfo	Gets information about all of the variables in a GDA.
gdaIsCplx	Checks to see if a variable in a GDA is complex.
gdaLoad	Loads variables in a GDA into the workspace.

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

These functions all operate on **GAUSS** Data Archives (GDA's). For more information, see GAUSS Data Archives, Section 20.3.

Data Sets

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

create	Creates and opens a data set.
datacreate	Creates a **v96** real data set.
datacreatecomplex	Creates a **v96** complex data set.
datalist	Lists selected variables from a data set.
dataopen	Opens a data set.
eof	Tests for end of file.
getnr	Computes number of rows to read per iteration for a program that reads data from a disk file in a loop.
getnrmt	Computes number of rows to read per iteration for a program that reads data from a disk file in a loop.
iscplxf	Returns whether a data set is real or complex.
loadd	Loads a small data set.
open	Opens an existing data set.
readr	Reads rows from open data set.
saved	Creates small data sets.
seekr	Moves pointer to specified location in open data set.
tempname	Creates a temporary file with a unique name.
typef	Returns the element size (2, 4 or 8 bytes) of data in open data set.
writer	Writes matrix to an open data set.

These functions all operate on **GAUSS** data sets (`.dat` files). For more information, see FILE I/O, Chapter 20.

To create a **GAUSS** data set from a matrix in memory, use **saved**. To create a data set from an existing one, use **create**. To create a data set from a large ASCII file, use the ATOG utility (see ATOG, Chapter 26).

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

The data in data sets may be specified as character or numeric. (See **File I/O**, Chapter 20.) See also **create** and **vartypef**.

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

Data Set Variable Names

getname	Returns column vector of variable names in a data set.
getnamef	Returns string array of variable names in a data set.
indices	Retrieves column numbers and names from a data set.
indices2	Similar to **indices**, but matches columns with names for dependent and independent variables.
indicesf	Retrieves column numbers and names from a data set.
indicesfn	Retrieves column numbers and names from a data set.
makevars	Decomposes matrix to create column vectors.
setvars	Creates globals using the names in a data set.
vartypef	Returns column vector of variable types (numeric/character) in a data set.

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

Data Coding

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

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

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

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

Sorting and Merging

intrleav	Produces one large sorted data file from two smaller sorted files having the same keys.
intrleavsa	Interleaves the rows of two string arrays that have been sorted on a common column.
mergeby	Produces one large sorted data file from two smaller sorted files having a single key column in common.
mergevar	Accepts a list of names of global matrices, and concatenates the corresponding matrices horizontally to form a single matrix.
sortc	Quick-sorts rows of matrix based on numeric key.
sortcc	Quick-sorts rows of matrix based on character key.
sortd	Sorts data set on a key column.
sorthc	Heap-sorts rows of matrix based on numeric key.
sorthcc	Heap-sorts rows of matrix based on character key.
sortind	Returns a sorted index of a numeric vector.
sortindc	Returns a sorted index of a character vector.
sortmc	Sorts rows of matrix on the basis of multiple columns.
sortr	Sorts rows of a matrix of numeric data.
sortrc	Sorts rows of a matrix of character data.
uniqindx	Returns a sorted unique index of a vector.

uniqindxsa	Computes the sorted index of a string vector, omitting duplicate elements.
unique	Removes duplicate elements of a vector.
uniquesa	Removes duplicate elements from a string vector.

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

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

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

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

30.8 Compiler Control

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

#lineson	Compiles program with line number and file name records.
#srcfile	Inserts source file name record at this point (currently used when doing data loop translation).
#srcline	Inserts source file line number record at this point (currently used when doing data loop translation).
#undef	Undefines a text-replacement or flag variable.

These commands are compiler directives. That is, they do not generate **GAUSS** program instructions; rather, they are instructions that tell **GAUSS** how to process a program during compilation. They determine what the final compiled form of a program will be. They are not executable statements and have no effect at run-time. (See COMPILER DIRECTIVES, Section 10.4, for more information.)

30.9 Program Control

Execution Control

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

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

Branching

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

```
        if iter > itlim;
            goto errout("Iteration limit exceeded");
        elseif iter =\,= 1;
            j = setup(x,y);
        else;
            j = iterate(x,y);
        endif;
         .
         .
         .
    errout:
        pop errmsg;
        print errmsg;
        end;
```

Looping

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

```
iter = 0;
do while dif > tol;
    { x,x0 } = eval(x,x0);
    dif = abs(x-x0);
    iter = iter + 1;
    if iter > maxits;
         break;
    endif;
    if not prtiter;
        continue;
    endif;
    format /rdn 1,0;
    print "Iteration: " iter;;
    format /re 16,8;
    print ",  Error:     " maxc(dif);
endo;

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

Subroutines

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

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

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

Procedures, Keywords, and Functions

endp	Terminates a procedure definition.
fn	Allows user to create one-line functions.
keyword	Begins the definition of a keyword procedure. Keywords are user-defined functions with local or global variables.
local	Declares variables local to a procedure.
proc	Begins definition of multi-line procedure.
retp	Returns from a procedure.

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

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

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

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

See PROCEDURES AND KEYWORDS, Chapter 12, and LIBRARIES, Chapter 18, for details.

Libraries

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

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

Compiling

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

GAUSS procedures and programs may be compiled to disk files. By then using this compiled code, the time necessary to compile programs from scratch is eliminated.

Use **compile** to compile a command file. All procedures, matrices and strings referenced by that program will be compiled as well.

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

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

Miscellaneous Program Control

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

30.10 OS Functions and File Management

cdir	Returns current directory.
ChangeDir	Changes directory in program.
chdir	Changes directory interactively.
DeleteFile	Deletes files.
dlibrary	Dynamically links and unlinks shared libraries.
dllcall	Calls functions located in dynamic libraries.
dos	Provides access to the operating system from within **GAUSS**.
envget	Gets an environment string.
exec	Executes an executable program file.

execbg	Provides access to the operating system from within **GAUSS**.
fileinfo	Takes a file specification, returns names and information of files that match.
filesa	Takes a file specification, returns names of files that match.
getpath	Returns an expanded filename including the drive and path.
searchsourcepath	Searches the source path and (if specified) the `src` subdirectory of the **GAUSS** installation directory for a specified file.
shell	Shells to OS.

30.11 Workspace Management

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

typecv	Returns types of symbols (argument contains the names of the symbols to be checked).

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

30.12 Error Handling and Debugging

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

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

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

30.13 String Handling

chrs	Converts ASCII values to a string.
convertsatostr	Converts a 1×1 string array to a string.

convertstrtosa	Converts a string to a 1×1 string array.
cvtos	Converts a character vector to a string.
ftocv	Converts an N×K matrix to a character matrix.
ftos	Converts a floating point scalar to string.
ftostrC	Converts a matrix to a string array using a C language format specification.
getf	Loads ASCII or binary file into string.
indsav	Checks one string array against another and returns
intrsectsa	Returns the intersection of two string vectors, with duplicates removed. the indices of the first string array in the second string array.
loads	Loads a string file (`.fst` file).
lower	Converts a string to lowercase.
parse	Parses a string, returning a character vector of tokens.
putf	Writes a string to disk file.
stocv	Converts a string to a character vector.
stof	Converts a string to floating point numbers.
strcombine	Converts an N×M string array to an N×1 string vector by combining each element in a column separated by a user-defined delimiter string.
strindx	Finds starting location of one string in another string.
strlen	Returns length of a string.
strput	Lays a substring over a string.
strrindx	Finds starting location of one string in another string, searching from the end to the start of the string.

strsect	Extracts a substring of a string.
strsplit	Splits an N×1 string vector into an N×K string array of the individual tokens.
strsplitPad	Splits an N×1 string vector into an N×K string array of the individual tokens. Pads on the right with null strings.
strtof	Converts a string array to a numeric matrix.
strtofcplx	Converts a string array to a complex numeric matrix.
strtriml	Strips all whitespace characters from the left side of each element in a string array.
strtrimr	Strips all whitespace characters from the right side of each element in a string array.
strtrunc	Truncates all elements of a string array to not longer than the specified number of characters.
strtruncl	Truncates the left side of all elements of a string array by a user-specified number of characters.
strtruncpad	Truncates all elements of a string array to the specified number of characters, adding spaces on the end as needed to achieve the exact length.
strtruncr	Truncates the right side of all elements of a string array by a user-specified number of characters.
token	Extracts the leading token from a string.
upper	Changes a string to uppercase.
vals	Converts a string to ASCII values.
varget	Accesses the global variable named by a string.
vargetl	Accesses the local variable named by a string.
varput	Assigns a global variable named by a string.

varputl	Assigns a local variable named by a string.

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

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

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

30.14 Time and Date Functions

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

dtvtodt	Converts DTV vector format to DT scalar format.
dtvtoutc	Converts DTV vector format to UTC scalar format.
etdays	Difference between two times in days.
ethsec	Difference between two times in hundredths of a second.
etstr	Converts elapsed time to string.
hsec	Returns elapsed time since midnight in hundredths of a second.
strtodt	Converts a string array of dates to a matrix in DT scalar format.
time	Returns current system time.
timedt	Returns system date and time in DT scalar format.
timestr	Formats time as "**hh:mm:ss**".
timeutc	Returns the number of seconds since January 1, 1970 Greenwich Mean Time.
todaydt	Returns system date in DT scalar format. The time returned is always midnight (00:00:00), the beginning of the returned day.
utctodt	Converts UTC scalar format to DT scalar format.
utctodtv	Converts UTC scalar format to DTV vector format.

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

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

will time the **GAUSS** multiplication operator.

30.15 Console I/O

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

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

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

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

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

30.16 Output Functions

Text Output

cls	Clears the window.

comlog	Controls interactive command logging.
csrcol	Gets column position of cursor on window.
csrlin	Gets row position of cursor on window.
ed	Accesses an alternate editor.
edit	Edits a file with the **GAUSS** editor.
format	Defines format of matrix printing.
formatcv	Sets the character data format used by **printfmt**.
formatnv	Sets the numeric data format used by **printfmt**.
header	Prints a header for a report.
headermt	Prints a header for a report.
locate	Positions the cursor on the window.
output	Redirects **print** statements to auxiliary output.
outwidth	Sets line width of auxiliary output.
print	Prints to window.
printdos	Prints a string for special handling by the OS.
printfm	Prints matrices using a different format for each column.
printfmt	Prints character, numeric, or mixed matrix using a default format controlled by the functions **formatcv** and **formatnv**.
satostrC	Copies from one string array to another using a C language format specifier string for each element.
screen [[**on**|**off**]]	Directs/suppresses **print** statements to window.
tab	Positions the cursor on the current line.

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

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

DOS Compatibility Windows

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

30.17 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 PUBLICATION QUALITY GRAPHICS, Chapter 24.

Graph Types

bar	Generates bar graph.
box	Graphs data using the box graph percentile method.
contour	Graphs contour data.
draw	Supplies additional graphic elements to graphs.
hist	Computes and graphs frequency histogram.
histf	Graphs a histogram given a vector of frequency counts.
histp	Graphs a percent frequency histogram of a vector.
loglog	Graphs X,Y using logarithmic X and Y axes.
logx	Graphs X,Y using logarithmic X axis.
logy	Graphs X,Y using logarithmic Y axis.
surface	Graphs a 3-D surface.

xy	Graphs X,Y using Cartesian coordinate system.
xyz	Graphs X,Y,Z using 3-D Cartesian coordinate system.

Axes Control and Scaling

_paxes	Turns axes on or off.
_pcross	Controls where axes intersect.
_pgrid	Controls major and minor grid lines.
_pticout	Controls direction of tick marks on axes.
_pxpmax	Controls precision of numbers on X axis.
_pxsci	Controls use of scientific notation on X axis.
_pypmax	Controls precision of numbers on Y axis.
_pysci	Controls use of scientific notation on Y axis.
_pzpmax	Controls precision of numbers on Z axis.
_pzsci	Controls use of scientific notation on Z axis.
scale	Scales X,Y axes for 2-D plots.
scale3d	Scales X,Y, and Z axes for 3-D plots.
xtics	Scales X axis and controls tick marks.
ytics	Scales Y axis and controls tick marks.
ztics	Scales Z axis and controls tick marks.

Text, Labels, Titles, and Fonts

_paxht	Controls size of axes labels.
_pdate	Controls date string contents.
_plegctl	Sets location and size of plot legend.
_plegstr	Specifies legend text entries.
_pmsgctl	Controls message position.
_pmsgstr	Specifies message text.
_pnum	Axes numeric label control and orientation.
_pnumht	Controls size of axes numeric labels.
_ptitlht	Controls main title size.
asclabel	Defines character labels for tick marks.
fonts	Loads fonts for labels, titles, messages, and legend.
title	Specifies main title for graph.
xlabel	Specifies X axis label.
ylabel	Specifies Y axis label.
zlabel	Specifies Z axis label.

Main Curve Lines and Symbols

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

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

Extra Lines and Symbols

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

Graphic Panel, Page, and Plot Control

_pageshf	Shifts the graph for printer output.
_pagesiz	Controls size of graph for printer output.
_plotshf	Controls plot area position.
_plotsiz	Controls plot area size.
_protate	Rotates the graph 90 degrees.
axmargin	Controls axes margins and plot size.
begwind	Graphic panel initialization procedure.
endwind	Ends graphic panel manipulation; displays graphs.

getwind	Gets current graphic panel number.
loadwind	Loads a graphic panel configuration from a file.
makewind	Creates graphic panel with specified size and position.
margin	Controls graph margins.
nextwind	Sets to next available graphic panel number.
savewind	Saves graphic panel configuration to a file.
setwind	Sets to specified graphic panel number.
window	Creates tiled graphic panels of equal size.

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

Output Options

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

Miscellaneous

_pbox	Draws a border around graphic panel/window.
_pcrop	Controls cropping of graphics data outside axes area.
_pframe	Draws a frame around 2-D, 3-D plots.
_pmcolor	Controls colors to be used for axes, title, *x* and *y* labels, date, box, and background.
graphset	Resets all PQG globals to default values.
rerun	Displays most recently created graph.
view	Sets 3-D observer position in workbox units.
viewxyz	Sets 3-D observer position in plot coordinates.
volume	Sets length, width, and height ratios of 3-D workbox.

Command Reference 31

abs

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

FORMAT *y* = **abs(***x***);**

INPUT *x* N×K matrix or sparse matrix or N-dimensional array.

OUTPUT *y* N×K matrix or sparse matrix or N-dimensional array containing absolute values of *x*.

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

```
x =  0.675243   1.053485
    -0.190746  -1.229539
```

```
y =  0.675243   1.053485
     0.190746   1.229539
```

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

acf

PURPOSE Computes sample autocorrelations.

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

INPUT

y	N×1 vector, data.
k	scalar, maximum number of autocorrelations to compute.
d	scalar, order of differencing.

OUTPUT

rk	K×1 vector, sample autocorrelations.

EXAMPLE

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

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

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

SOURCE tsutil.src

aconcat

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

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

INPUT

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

OUTPUT

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

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

EXAMPLE

```
a = arrayinit(2|3|4,0);
b = 3*ones(3,4);
y = aconcat(a,b,3);
```

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

```
a = reshape(seqa(1,1,20),4,5);
```

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

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

```
a = arrayinit(2|3|4,0);
b = seqa(1,1,24);
b = areshape(b,2|3|4);
y = aconcat(a,b,5);
```

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

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

y will be a 2×3×5 array, such that:

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

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

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

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

SEE ALSO **areshape**

aeye

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

FORMAT *a* **= aeye(***o***);**

INPUT *o* N×1 vector of orders, the sizes of the dimensions of *a*.

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

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

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

EXAMPLE
```
o = { 2,3,4 };
a = aeye(o);
```

a will be a 2×3×4 array, such that:

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

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

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

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{matrix}$$

SEE ALSO **eye**

amax

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

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

INPUT

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

OUTPUT

y	N-dimensional array.

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

EXAMPLE
```
x = round(10*rndn(24,1));
x = areshape(x,2|3|4);
dim = 2;
y = amax(x,dim);
```

x is a 2×3×4 array, such that:

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

$$\begin{matrix} -14 & 3 & 3 & -9 \\ -7 & 21 & -4 & 21 \\ 7 & -5 & 20 & -2 \end{matrix}$$

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

$$\begin{matrix} 10 & -12 & -9 & -4 \\ 1 & -6 & -10 & 0 \\ -8 & 9 & 8 & -6 \end{matrix}$$

y will be a 2×1×4 array, such that:

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

$$\begin{matrix} 7 & 21 & 20 & 21 \end{matrix}$$

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

$$\begin{matrix} 10 & 9 & 8 & 0 \end{matrix}$$

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

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

y will be a 2×3×1 array, such that:

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

3
21
20

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

10
1
9

SEE ALSO **amin**, **maxc**

amean

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

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

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

OUTPUT
y [N-1]-dimensional array.

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

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

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

x is a 2×3×4 array, such that:

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

```
 1   2   3   4
 5   6   7   8
 9  10  11  12
```

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

```
13  14  15  16
17  18  19  20
21  22  23  24
```

y will be a 1×3×4 array, such that:

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

```
 7   8   9  10
11  12  13  14
15  16  17  18
```

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

Using the same array **x** as the above example, this example computes the mean across the first dimension. **y** will be a 2×3×1 array, such that:

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

2.5
6.5
10.5

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

14.5
18.5
22.5

SEE ALSO **asum**

AmericanBinomCall

PURPOSE Prices American call options using binomial method.

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

INPUT	*S0*	scalar, current price.
	K	M×1 vector, strike prices.
	r	scalar, risk free rate.
	div	continuous dividend yield.
	tau	scalar, elapsed time to exercise in annualized days of trading.
	sigma	scalar, volatility.
	N	number of time segments.
OUTPUT	*c*	M×1 vector, call premiums.

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

EXAMPLE
```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
c = AmericanBinomCall(S0,K,r,0,tau,sigma,60);
print c;

   17.344044
   15.058486
   12.817427
```

SOURCE finprocs.src

AmericanBinomCall_Greeks

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

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.

	div	continuous dividend yield.
	tau	scalar, elapsed time to exercise in annualized days of trading.
	sigma	scalar, volatility.
	N	number of time segments.

GLOBAL INPUT

_fin_thetaType scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.

_fin_epsilon scalar, finite difference stepsize. Default = 1e-8.

OUTPUT

	d	M×1 vector, delta.
	g	M×1 vector, gamma.
	t	M×1 vector, theta.
	v	M×1 vector, vega.
	rh	M×1 vector, rho.

REMARKS

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

EXAMPLE

```
S0 = 305;
K = 300;
r = .08;
sigma = .25;
tau = .33;
div = 0;
print AmericanBinomcall_Greeks (S0,K,r,0,tau,sigma,30);

      0.70631204
   0.00076381912
      -17.400851
       68.703851
       76.691829
```

SOURCE

finprocs.src

SEE ALSO **AmericanBinomCall_Impvol**, **AmericanBinomCall**, **AmericanBinomPut_Greeks**, **AmericanBSCall_Greeks**

AmericanBinomCall_ImpVol

PURPOSE Computes implied volatilities for American call options using binomial method.

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

INPUT

c	M×1 vector, call premiums
S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
N	number of time segments.

OUTPUT

sigma	M×1 vector, volatility.

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

EXAMPLE

```
c = { 13.70, 11.90, 9.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
div = 0;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
```

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

  0.19629517
  0.16991943
  0.12874756
```

SOURCE finprocs.src

AmericanBinomPut

PURPOSE Prices American put options using binomial method.

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

OUTPUT

c	M×1 vector, put premiums.

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

EXAMPLE

```
S0 = 718.46;
```

```
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
c = AmericanBinomPut(S0,K,r,0,tau,sigma,60);
print c;

   16.986117
   19.729923
   22.548538
```

SOURCE finprocs.src

AmericanBinomPut_Greeks

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

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

GLOBAL INPUT **_fin_thetaType** scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.

_fin_epsilon scalar, finite difference stepsize. Default = 1e-8.

OUTPUT

d M×1 vector, delta.

g M×1 vector, gamma.

t M×1 vector, theta.

v M×1 vector, vega.

rh M×1 vector, rho.

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

EXAMPLE

```
S0 = 305;
K = 300;
r = .08;
div = 0;
sigma = .25;
tau = .33;
print AmericanBinomPut_Greeks(S0,K,r,0,tau,sigma,60);

     -0.38324908
   0.00076381912
       8.1336630
       68.337294
      -27.585043
```

SOURCE finprocs.src

SEE ALSO **AmericanBinomPut_Impvol**, **AmericanBinomPut**, **AmericanBinomCall_Greeks**, **AmericanBSPut_Greeks**

AmericanBinomPut_ImpVol

PURPOSE Computes implied volatilities for American put options using binomial method.

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

INPUT

c	M×1 vector, put premiums
S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
N	number of time segments.

OUTPUT

sigma	M×1 vector, volatility.

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

EXAMPLE

```
p = { 14.60, 17.10, 20.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
div = 0;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
sigma = AmericanBinomPut_ImpVol(p,S0,K,r,0,tau,30);
print sigma;
```

```
0.12466064
0.16583252
0.21203735
```

SOURCE finprocs.src

AmericanBSCall

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

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

OUTPUT

c	M×1 vector, call premiums.

EXAMPLE

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
c = AmericanBSCall(S0,K,r,0,tau,sigma);
print c;

  32.005720
```

```
31.083232
30.367548
```

SOURCE finprocs.src

AmericanBSCall_Greeks

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

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

GLOBAL INPUT

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

OUTPUT

d	M×1 vector, delta.
g	M×1 vector, gamma.
t	M×1 vector, theta.
v	M×1 vector, vega.
rh	M×1 vector, rho.

EXAMPLE

```
S0 = 305;
```

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

  0.40034039
 0.016804021
  -55.731079
   115.36906
   46.374528
```

SOURCE finprocs.src

SEE ALSO **AmericanBSCall_Impvol**, **AmericanBSCall**, **AmericanBSPut_Greeks**, **AmericanBinomCall_Greeks**

AmericanBSCall_ImpVol

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

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

INPUT

c	M×1 vector, call premiums
S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.

OUTPUT

sigma	M×1 vector, volatility.

EXAMPLE

```
c = { 13.70, 11.90, 9.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
sigma = AmericanBSCall_ImpVol(c,S0,K,r,0,tau);
print sigma;

  0.10259888
 0.088370361
 0.066270752
```

SOURCE finprocs.src

AmericanBSPut

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

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

OUTPUT

c	M×1 vector, put premiums.

EXAMPLE

```
S0 = 718.46;
```

```
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
c = AmericanBSPut(S0,K,r,0,tau,sigma);
print c;

   16.870783
   19.536842
   22.435487
```

SOURCE finprocs.src

AmericanBSPut_Greeks

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

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

GLOBAL INPUT

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

OUTPUT

d	M×1 vector, delta.
g	M×1 vector, gamma.
t	M×1 vector, theta.
v	M×1 vector, vega.
rh	M×1 vector, rho.

EXAMPLE

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

     -0.33296721
    0.0091658294
      -17.556118
       77.614237
      -40.575963
```

SOURCE finprocs.src

SEE ALSO **AmericanBSPut_Impvol**, **AmericanBSPut**, **AmericanBSCall_Greeks**, **AmericanBinomPut_Greeks**

AmericanBSPut_ImpVol

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

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

INPUT

c	M×1 vector, put premiums

	S0	scalar, current price.
	K	M×1 vector, strike prices.
	r	scalar, risk free rate.
	div	continuous dividend yield.
	tau	scalar, elapsed time to exercise in annualized days of trading.
OUTPUT	*sigma*	M×1 vector, volatility.

EXAMPLE

```
p = { 14.60, 17.10, 20.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
sigma = AmericanBSPut_ImpVol(p,S0,K,r,0,tau);
print sigma;

  0.12753662
  0.16780029
  0.21396729
```

SOURCE finprocs.src

amin

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

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

INPUT	*x*	N-dimensional array.

dim scalar, number of dimension across which to find the minimum value.

OUTPUT *y* N-dimensional array.

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

EXAMPLE
```
x = round(10*rndn(24,1));
x = areshape(x,2|3|4);
dim = 2;
y = amin(x,dim);
```

x is a 2×3×4 array, such that:

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

−14	3	3	−9
−7	21	−4	21
7	−5	20	−2

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

10	−12	−9	−4
1	−6	−10	0
−8	9	8	−6

y will be a 2×1×4 array, such that:

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

−14	−5	−4	−9

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

−8 −12 −10 −6

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

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

y will be a 2×3×1 array, such that:

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

−14
−7
−5

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

−12
−10
−8

SEE ALSO **amax**, **minc**

amult

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

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

INPUT *a* N-dimensional array.

b N-dimensional array.

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

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

EXAMPLE
```
a = areshape(seqa(1,1,12),2|3|2);
b = areshape(seqa(1,1,16),2|2|4);
y = amult(a,b);
```

a is a 2×3×2 array, such that:

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

```
1  2
3  4
5  6
```

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

```
 7   8
 9  10
11  12
```

b is a 2×2×4 array, such that:

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

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

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

```
 9   10   11   12
13   14   15   16
```

y will be a 2×3×4 array, such that:

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

```
11   14   17   20
23   30   37   44
35   46   57   68
```

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

```
167   182   197   212
211   230   249   268
255   278   301   324
```

annualTradingDays

PURPOSE Compute number of trading days in a given year.

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

INPUT *a* scalar, year.

OUTPUT *n* number of trading days in year

REMARKS A trading day is a weekday that is not a holiday as defined by the New York Stock Exchange from 1888 through 2006. Holidays are defined in `holidays.asc`. You may edit that file to modify or add holidays.

SOURCE `finutils.src`

GLOBALS **`_fin_annualTradingDays`**, **`_fin_holidays`**

SEE ALSO **`elapsedTradingDays`**, **`getNextTradingDay`**, **`getPreviousTradingDay`**, **`getNextWeekDay`**, **`getPreviousWeekDay`**

arccos

PURPOSE Computes the inverse cosine.

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

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array containing the angle in radians whose cosine is *x*.

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

EXAMPLE
```
x = { -1,  -0.5,  0,  0.5,  1 };
y = 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* **= arcsin(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

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

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

EXAMPLE
```
x = { -1,  -0.5,  0,  0.5,  1 };
y = 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

areshape

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

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

INPUT
- *x* scalar, matrix, or N-dimensional array.
- *o* M×1 vector of orders, the sizes of the dimensions of the new array.

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

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

EXAMPLE
```
x = 3;
orders = { 2,3,4 };
```

```
y = areshape(x,orders);
```

y will be a 2×3×4 array of threes.

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

y will be a 2×3×4×5 array. Since **y** contains 120 elements and **x** contains only 90, the first 90 elements of **y** will be set to the sequence of integers from 1 to 90 that are contained in **x**, and the last 30 elements of **y** will be set to the sequence of integers from 1 to 30 contained in the first 30 elements of **x**.

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

y will be a 3×2×4 array. Since **y** contains 24 elements, and **x** contains 60, the elements of **y** will be set to the sequence of integers from 1 to 24 contained in the first 24 elements of **x**.

SEE ALSO **aconcat**

arrayalloc

PURPOSE Creates an N-dimensional array with unspecified contents.

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

INPUT

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

OUTPUT *y* N-dimensional array.

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

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

y will be a complex 2×3×4 array with unspecified contents.

SEE ALSO **arrayinit**, **setarray**

arrayindex

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

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

INPUT *si* scalar, index into vector or 1-dimensional array.

o N×1 vector of orders of an N-dimensional array.

OUTPUT *i* N×1 vector of indices, index of corresponding element in N-dimensional array.

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

EXAMPLE
```
orders = { 2,3,4,5 };
v = rndu(prodc(orders),1);
a = areshape(v,orders);
vi = 50;
ai = arrayindex(vi,orders);
```

```
print vi;
print ai;
print v[vi];
print getarray(a,ai);
```

```
vi =  50

      1
      3
ai =  2
      5

v[vi] =  0.13220899

getarray(a,ai) =  0.13220899
```

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

SEE ALSO **singleindex**

arrayinit

PURPOSE Creates an N-dimensional array with a specified fill value.

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

INPUT
o N×1 vector of orders, the sizes of the dimensions of the array.
v scalar, value to initialize. If *v* is complex the result will be complex.

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

EXAMPLE
```
orders = { 2,3,4 };
y = arrayinit(orders, 0);
```

y will be a 2×3×4 array of zeros.

SEE ALSO **arrayalloc**

arraytomat

PURPOSE Converts an array to type matrix.

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

INPUT *a* N-dimensional array.

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

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

EXAMPLE
```
a = arrayinit(3|4,2);
y = arraytomat(a);
```

```
      2  2  2  2
y =   2  2  2  2
      2  2  2  2
```

SEE ALSO **mattoarray**

asciiload

PURPOSE Loads data from a delimited ASCII text file into an N×1 vector.

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

INPUT *filename* string, name of data file.

OUTPUT *y* N×1 vector.

REMARKS The file extension must be included in the file name.

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

This command loads as many elements as possible from the file into an N×1 vector. This allows you to verify if the load was successful by calling **rows(***y***)** after **asciiload** to see how many elements were actually loaded. You may then **reshape** the N×1 vector to the desired form. You could, for instance, put the number of rows and columns of the matrix right in the file as the first and second elements and **reshape** the remainder of the vector to the desired form using those values.

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

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

you may use the command:

```
y = asciiload("myfile.asc");
```

$$y = \begin{matrix} 2.805 \\ 16.568 \\ -4.871 \\ 3.399 \\ 17.361 \\ -12.725 \end{matrix}$$

SEE ALSO **load**, **dataload**

asclabel

PURPOSE To set up character labels for the X and Y axes.

LIBRARY pgraph

FORMAT **asclabel(***xl*,*yl***);**

INPUT

xl string or N×1 character vector, labels for the tick marks on the X axis. Set to 0 if no character labels for this axis are desired.

yl string or M×1 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.

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

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

SOURCE pgraph.src

SEE ALSO **xtics**, **ytics**, **scale**, **scale3d**, **fonts**

astd

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

FORMAT *y* **= astd(***x*,*dim***);**

INPUT

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

OUTPUT

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

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

This function essentially computes:

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

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

EXAMPLE

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

If **a** =

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

```
−38.357528   2.0560337
−21.331064   32.500431
```

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

```
−6.7540544   16.374989
−15.245137   21.824196
```

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

```
18.636931    35.264181
−21.137995   −33.715808
```

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

```
23.008600   −15.933576
54.852400   −7.8360916
```

then **y** =

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

```
28.229091   21.705020
37.152755   29.944103
```

In this example, 16 standard Normal random variables are generated. They are multiplied by 25 and **areshape**'d into a 4×2×2 array, and the standard deviation is computed across the third dimension of the array.

SEE ALSO **astds**, **stdc**

astds

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

FORMAT *y* = **astds**(*x*,*dim*);

INPUT

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

OUTPUT

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

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

This function essentially computes:

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

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

EXAMPLE

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

If **a** =

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

```
−38.357528   2.0560337
−21.331064   32.500431
```

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

```
−6.7540544   16.374989
−15.245137   21.824196
```

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

```
18.636931     35.264181
−21.137995   −33.715808
```

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

```
23.008600   −15.933576
54.852400   −7.8360916
```

then **y** =

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

```
24.447110   18.797099
32.175230   25.932354
```

In this example, 24 standard Normal random variables are generated. They are multiplied by 10 and **areshape**'d into a 4×3×2 array, and the standard deviation is computed across the third dimension of the array.

SEE ALSO **astd**, **stdsc**

asum

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

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

INPUT

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

OUTPUT

y	N-dimensional array.

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

EXAMPLE

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

x is a 2×3×4 array, such that:

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

```
 1   2   3   4
 5   6   7   8
 9  10  11  12
```

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

```
13  14  15  16
17  18  19  20
21  22  23  24
```

y will be a 1×3×4 array, such that:

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

```
14  16  18  20
22  24  26  28
30  32  34  36
```

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

Using the same array **x** as the above example, this example computes the sum across the first dimension. **y** will be a 2×3×1 array, such that:

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

```
10
26
42
```

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

58
74
90

SEE ALSO **amean**

atan

PURPOSE Returns the arctangent of its argument.

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

INPUT *x* N×K matrix or N-dimensional array.

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

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

For real *x*, the arctangent of *x* is the angle whose tangent is *x*. The result is a value in radians in the range $\frac{-\pi}{2}$ 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 = 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* **= atan2(***y***,***x***);**

INPUT

y	N×K matrix or P-dimensional array where the last two dimensions are N×K, the *Y* coordinate.
x	L×M matrix or P-dimensional array where the last two dimensions are L×M, E×E conformable with *y*, the *X* coordinate.

OUTPUT

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

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

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

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);
```

```
z =  0.785398
     1.107149
     1.249046
     1.325818
```

SEE ALSO **atan**, **sin**, **cos**, **pi**, **tan**, **arcsin**, **arccos**

atranspose

PURPOSE Transposes an N-dimensional array.

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

INPUT

x	N-dimensional array.
nd	N×1 vector of dimension indices, the new order of dimensions.

OUTPUT

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

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

EXAMPLE

```
x = seqa(1,1,24);
x = areshape(x,2|3|4);
nd = { 2,1,3 };
y = atranspose(x,nd);
```

This example transposes the dimensions of **x** that correspond to the first and second elements of the vector of orders. **x** is a 2×3×4 array, such that:

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

```
 1   2   3   4
 5   6   7   8
 9  10  11  12
```

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

```
13  14  15  16
17  18  19  20
21  22  23  24
```

y will be a 3×2×4 array such that:

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

```
 1   2   3   4
13  14  15  16
```

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

```
 5   6   7   8
17  18  19  20
```

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

```
 9  10  11  12
21  22  23  24
```

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

Using the same array **x** as the example above, this example transposes all three dimensions of **x**, returning a 3×4×2 array **y**, such that:

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

```
 1  13
 2  14
 3  15
 4  16
```

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

```
 5  17
 6  18
 7  19
 8  20
```

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

```
 9  21
10  22
11  23
12  24
```

SEE ALSO **areshape**

axmargin

PURPOSE Sets 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 9×6.855 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	K×K matrix or N-dimensional array where the last two dimensions are K×K .

OUTPUT

b	K×K matrix or N-dimensional array where the last two dimenstions are K×K, balanced matrix.
z	K×K matrix or N-dimensional array where the last two dimensions are K×K, 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

PURPOSE Extracts bands from a symmetric banded matrix.

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

INPUT

	y	K×K symmetric banded matrix.
	n	scalar, number of subdiagonals.

OUTPUT

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

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

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

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

EXAMPLE

```
x = { 1 2 0 0,
```

```
        2 8 1 0,
        0 1 5 2,
        0 0 2 3 };

bx = band(x,1);
```

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

SEE ALSO **bandchol**, **bandcholsol**, **bandltsol**, **bandrv**, **bandsolpd**

bandchol

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

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

INPUT *a* K×N compact form matrix.

OUTPUT *l* K×N 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);
```

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

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

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

SEE ALSO **band**, **bandcholsol**, **bandltsol**, **bandrv**, **bandsolpd**

bandcholsol

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

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

INPUT

b	K×M matrix.
l	K×N compact form matrix.

OUTPUT

x	K×M 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 $A = L \times L'$. 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

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

cx =

0.0000000	1.0000000
2.0000000	2.0000000
0.50000000	2.1794495
0.91766294	1.4689774

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

```
xi =  2.0731707   −0.53658537   0.14634146   −0.097560976
     −0.53658537   0.26829268  −0.073170732   0.048780488
      0.14634146  −0.073170732  0.29268293   −0.19512195
     −0.097560976  0.048780488  −0.19512195   0.46341463
```

SEE ALSO **band**, **bandchol**, **bandltsol**, **bandrv**, **bandsolpd**

bandltsol

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

FORMAT x **= bandltsol(**b**,**A**);**

INPUT
b	K×M matrix.
A	K×N compact form matrix.

OUTPUT
x	K×M 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
```
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
```

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

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

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

```
xci =  1.0000000  0.00000000  0.00000000  0.00000000
      −1.0000000  0.50000000  0.00000000  0.00000000
      0.22941573 −0.11470787  0.45883147  0.00000000
     −0.14331487  0.071657436 −0.28662975  0.68074565
```

SEE ALSO **band**, **bandchol**, **bandcholsol**, **bandrv**, **bandsolpd**

bandrv

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

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

INPUT *a* K×N compact form matrix.

OUTPUT *y* K×K 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 K×K matrix with N-1 subdiagonals.

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
```

```
x = bandrv(bx);
```

```
x =  1.0000000   2.0000000  0.00000000  0.00000000
     2.0000000   8.0000000  1.0000000   0.00000000
0.00000000       1.0000000  5.0000000   2.0000000
0.00000000  0.00000000      2.0000000   3.0000000
```

SEE ALSO **band**, **bandchol**, **bandcholsol**, **bandltsol**, **bandsolpd**

bandsolpd

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

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

INPUT
b	K×M matrix.
A	K×N compact form matrix.

OUTPUT
x	K×M matrix.

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

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

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

SEE ALSO **band**, **bandchol**, **bandcholsol**, **bandltsol**, **bandrv**

bar

PURPOSE Generates a bar graph.

LIBRARY pgraph

FORMAT **bar(**val**,**ht**);**

INPUT

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

ht N×K 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 which 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 scalar, width and type of bars in bar graphs and histograms. The valid range is 0-1. If this is 0, the bars will be a single pixel wide. If this is 1, the bars will touch each other.

If this value is positive, the bars will overlap. If negative, the bars will be plotted side-by-side. The default is 0.5.

_pbartyp K×2 matrix.

The first column controls the bar shading:

0	no shading.
1	dots.
2	vertical cross-hatch.
3	diagonal lines with positive slope.
4	diagonal lines with negative slope.
5	diagonal cross-hatch.
6	solid.

The second column controls the bar color.

REMARKS

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

EXAMPLE

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

```
library pgraph;
graphset;
```

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

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

base10

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

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

INPUT *x* scalar, number to break down.

OUTPUT *M* scalar, in the range $-10 < M < 10$.

P scalar, integer power such that:

$$M * 10^P = x$$

EXAMPLE

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

b = 4.5000000

```
e = 3.0000000
```

SOURCE base10.src

begwind

PURPOSE Initializes 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**

besselj

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

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

INPUT

n	N×K matrix or P-dimensional array where the last two dimensions are N×K, the order of the Bessel function. Nonintegers will be truncated to an integer.
x	L×M matrix or P-dimensional array where the last two dimensions are L×M, E×E conformable with n.

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

EXAMPLE
```
n = { 0, 1 };
x = { 0.1 1.2, 2.3 3.4 };
y = besselj(n,x);
```

```
y =  0.99750156   0.67113274
     0.53987253   0.17922585
```

SEE ALSO **bessely**, **mbesseli**

bessely

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

FORMAT *y* **= bessely(***n*,*x***);**

INPUT *n* N×K matrix or P-dimensional array where the last two dimensions are N×K, the order of the Bessel function. Nonintegers will be truncated to an integer.

x L×M matrix or P-dimensional array where the last two dimensions are L×M, E×E conformable with *n*.

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

EXAMPLE
```
n = { 0, 1 };
x = { 0.1 1.2, 2.3 3.4 };
y = bessely(n,x);
```

$$y = \begin{matrix} -1.5342387 & 0.22808351 \\ 0.052277316 & 0.40101529 \end{matrix}$$

SEE ALSO **besselj**, **mbesseli**

box

PURPOSE Graphs data using the box graph percentile method.

LIBRARY pgraph

FORMAT **box(***grp***,***y***);**

INPUT

grp 1×M vector. This contains the group numbers corresponding to each column of *y* data. If scalar 0, a sequence from 1 to **cols(***y***)** will be generated automatically for the X axis.

y N×M matrix. Each column represents the set of *y* values for an individual percentiles box symbol.

GLOBAL INPUT

_pboxctl 5×1 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 50^{th} 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:

$$\begin{aligned} intqrange &= 75^{th} - 25^{th} \\ min &= 25^{th} - 1.5 intqrange \\ max &= 75^{th} + 1.5 intqrange \end{aligned}$$

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 1×M vector or scalar as follows:

0 Plot boxes only, no symbols.

1 Plot boxes and plot symbols which lie outside the *min* and *max* box values.

2 Plot boxes and all symbols.

-1 Plot symbols only, no boxes.

These capabilities are in addition to the usual line control capabilities of **_plctrl**.

_pcolor 1×M vector or scalar for symbol colors. If scalar, all symbols will be one color.

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

SOURCE pbox.src

boxcox

PURPOSE Computes the Box-Cox function.

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

INPUT

x	M×N matrix or P-dimensional array where the last two dimensions are M×N.
lambda	K×L matrix or P-dimensional array where the last two dimensions are K×L, E×E conformable to *x*.

OUTPUT

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

REMARKS Allowable range for *x* is:

$$x > 0$$

The **boxcox** function computes

$$\texttt{boxcox}(x) = \frac{x^{\lambda} - 1}{\lambda}$$

EXAMPLE

```
x = { .2 .3, 1.5 2.5 };
lambda = { .4, 2 };
y = boxcox(x,lambda)
```

```
y =  -1.1867361   -0.95549787
      0.62500000    2.62500000
```

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;
```

$$x = \begin{matrix} 1.000 & 0.326 & -2.682 & -0.594 \\ 0.000 & 1.000 & -0.879 & 0.056 \\ 0.000 & 0.000 & 1.000 & -0.688 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{matrix}$$

REMARKS This command works just like in C.

SEE ALSO **continue**, **do**, **for**

call

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

FORMAT **call** *function_name*(*argument_list*);

call *function_name*;

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

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

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

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

SEE ALSO **proc**

cdfbeta

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

FORMAT *y* = **cdfbeta**(*x*,*a*,*b*);

INPUT
x	N×K matrix.
a	L×M matrix, E×E conformable with *x*.
b	P×Q matrix, E×E 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:

$$\begin{aligned} 0 &\leq x \leq 1 \\ a &> 0 \\ b &> 0 \end{aligned}$$

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);
```

$$\mathtt{y} = \begin{matrix} 0.142285 \\ 0.206629 \\ 0.260575 \\ 0.310875 \end{matrix}$$

SEE ALSO

cdfchic, **cdffc**, **cdfn**, **cdfnc**, **cdftc**, **gamma**

TECHNICAL NOTES

cdfbeta has the following approximate accuracy:

		max(a,b)	≤	500	absolute error is approx. ±5e-13
500	<	max(a,b)	≤	10,000	absolute error is approx. ±5e-11
10,000	<	max(a,b)	≤	200,000	absolute error is approx. ±1e-9
200,000	<	max(a,b)			Normal approximations are used; absolute error is approx. ±2e-9

REFERENCES

1. Bol'shev, L.N."Asymptotically Perason's Transformations." Teor. Veroyat. Primen. *Theory of Probability and its Applications*. Vol. 8, No. 2, 1963,

129-55.

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

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

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

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

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

cdfbvn

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

FORMAT *c* = **cdfbvn**(*h*,*k*,*r*);

INPUT

h	N×K matrix, the upper limits of integration for variable 1.
k	L×M matrix, E×E conformable with *h*, the upper limits of integration for variable 2.
r	P×Q matrix, E×E 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:

$$\begin{aligned} -\infty &< h &< +\infty \\ -\infty &< k &< +\infty \\ -1 &\leq r &\leq 1 \end{aligned}$$

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:

$$\begin{aligned} h &= (ht - ux)./sx; \\ k &= (kt - uy)./sy; \end{aligned}$$

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

SEE ALSO **cdfn**, **cdftvn**

TECHNICAL NOTES The absolute error for **cdfbvn** is approximately ±5.0e-9 for the entire range of arguments.

REFERENCES 1. Daley, D.J. "Computation of Bi- and Tri-variate Normal Integral." *Appl. Statist.* Vol. 23, No. 3, 1974, 435-38.

2. Owen, D.B. "A Table of Normal Integrals." *Commun. Statist.-Simula. Computa.*, B9(4). 1980, 389-419.

cdfbvn2

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

FORMAT *y* = **cdfbvn2**(*h*,*dh*,*k*,*dk*,*r*);

INPUT

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

OUTPUT

y	N×1 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 N×1 vectors.

cdfbvn2 computes:

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

cdfbvn2 computes an error estimate for each set of inputs. The size of the error depends on the input arguments. If **trap 2** is set, a warning message is displayed when the error reaches 0.01***abs(***y***)**. For an estimate of the actual error, see **cdfbvn2e**.

EXAMPLE Example 1

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

   1.4105101488974692e-001
```

Example 2

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

   4.9303806576313238e-32
```

Example 3

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

  0.0000000000000000e+000
```

Example 4

```
trap 2,2;
print cdfbvn2(1,-1e-45,1,1e-45,0.5);

  WARNING: Dubious accuracy from cdfbvn2:
  0.000e+000 +/- 2.8e-060
  0.0000000000000000e+000
```

SOURCE lncdfn.src

SEE ALSO **cdfbvn2e**, **lncdfbvn2**

cdfbvn2e

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

FORMAT { *y*,*e* } = **cdfbvn2e**(*h*,*dh*,*k*,*dk*,*r*);

INPUT

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

OUTPUT

y	N×1 vector, the integral over the rectangle bounded by *h*, *h* + *dh*, *k*, and *k* + *dk* of the standardized bivariate Normal distribution.
e	N×1 vector, an error estimate.

REMARKS Scalar input arguments are okay; they will be expanded to N×1 vectors. **cdfbvn2e** computes:

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

The real answer is $y \pm e$. The size of the error depends on the input arguments.

EXAMPLE Example 1

```
print cdfbvn2e(1,-1,1,-1,0.5);

    1.4105101488974692e-001
    1.9927918166193113e-014
```

Example 2

```
print cdfbvn2e(1,-1e-15,1,-1e-15,0.5);

    7.3955709864469857e-032
    2.8306169312687801e-030
```

Example 3

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

    0.0000000000000000e+000
    2.8306169312687770e-060
```

SEE ALSO **cdfbvn2**, **lncdfbvn2**

cdfchic

PURPOSE Computes the complement of the cdf of the chi-square distribution.

FORMAT *y* = **cdfchic**(*x*,*n*)

INPUT
x N×K matrix.

n L×M matrix, E×E 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:

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

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

This equals $1 - \chi^2_n(x)$, Thus, to get the chi-squared cdf, subtract **cdfchic(*x*,*n*)** from 1. The complement of the cdf is computed because this is what is most commonly needed in statistical applications, and because it can be computed with fewer problems of roundoff error.

EXAMPLE

```
x = { .1, .2, .3, .4 };
n = 3;
y = cdfchic(x,n);
```

```
y =
0.991837
0.977589
0.960028
0.940242
```

SEE ALSO

cdfbeta, **cdffc**, **cdfn**, **cdfnc**, **cdftc**, **gamma**

TECHNICAL NOTES

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

For higher accuracy when $n > 1000$, use: 1 - **cdfgam**(0.5**x*,0.5**n*);

REFERENCES 1. Bhattacharjee, G.P. "Algorithm AS 32, the Incomplete Gamma Integral." *Applied Statistics*. Vol. 19, 1970, 285-87.

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

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

cdfchii

PURPOSE Compute chi-square abscissae values given probability and degrees of freedom.

FORMAT *c* = **cdfchii**(*p*,*n*);

INPUT
p M×N matrix, probabilities.
n L×K matrix, E×E conformable with *p*, degrees of freedom.

OUTPUT
c max(M,L) by max(N,K) matrix, abscissae values for chi-squared distribution.

EXAMPLE The following generates a 3×3 matrix of pseudo-random numbers with a chi-squared distribution with expected value of 4:

```
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
```

SOURCE cdfchii.src

SEE ALSO **gammaii**

cdfchinc

PURPOSE Computes the cumulative distribution function for the noncentral chi-square distribution.

FORMAT *y* = **cdfchinc**(*x*,*v*,*d*);

INPUT

x	N×1 vector, values of upper limits of integrals, must be greater than 0.
v	scalar, degrees of freedom, $v > 0$.
d	scalar, noncentrality parameter, $d > 0$. This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. (See Scheffe, *The Analysis of Variance*, App. IV, 1959.)

OUTPUT

y	N×1 vector.

REMARKS *y* is the integral from 0 to *x* of the noncentral chi-square distribution with *v* degrees of freedom and noncentrality *d*.

cdfchinc can return a vector of values, but the degrees of freedom and noncentrality parameter must be the same for all values of *x*.

- Relation to cdfchic:
  cdfchic(*x*,*v*) **= 1 - cdfchinc**(*x*,*v*,**0**);
- The formula used is taken from Abramowitz and Stegun, *Handbook of Mathematical Functions*, 1970, 942, formula 26.4.25.

EXAMPLE

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

```
0.0042086234
 0.016608592
  0.30954232
  0.99441140
```

SOURCE cdfnonc.src

SEE ALSO **cdffnc**, **cdftnc**

cdffc

PURPOSE Computes the complement of the cumulative distribution function of the *F* distribution.

FORMAT *y* = **cdffc(***x*,*n1*,*n2*);

INPUT

	x	N×K matrix.
	n1	L×M matrix, E×E conformable with *x*.
	n2	P×Q matrix, E×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.

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 **cdffc(***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.

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.

For max(n1,n2) ≤ 1000, the absolute error is approx. ±5e-13. For max(n1,n2) > 1000, Normal approximations are used and the absolute error is approx. ±2e-6.

For higher accuracy when max(n1,n2) > 1000, use **cdfbeta**(*n2/(n2+n1*x), n2/2, n1/2);*

EXAMPLE

```
x = { .1, .2, .3, .4 };
n1 = 0.5;
n2 = 0.3;
print cdffc(x,n1,n2);

   0.751772
   0.708152
   0.680365
   0.659816
```

SEE ALSO **cdfbeta**, **cdfchic**, **cdfn**, **cdfnc**, **cdftc**, **gamma**

REFERENCES

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

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

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

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

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

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

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

cdffnc

PURPOSE

Computes the cumulative distribution function of the noncentral *F* distribution.

FORMAT

y = **cdffnc**(*x*,*n1*,*n2*,*d*);

INPUT

x	N×1 vector, values of upper limits of integrals, $x > 0$.
v1	scalar, degrees of freedom of numerator, $n1 > 0$.
v2	scalar, degrees of freedom of denominator, $n2 > 0$.
d	scalar, noncentrality parameter, $d > 0$. This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. (See Scheffe, *The Analysis of Variance*, App. IV, 1959.)

OUTPUT

y	N×1 vector.

TECHNICAL NOTES

- Relation to cdffc:
  cdffc(*x*,*n1*,*n2*) = 1 - **cdffnc**(*x*,*n1*,*n2*,0);
- The formula used is taken from Abramowitz and Stegun, *Handbook of Mathematical Functions*, 1970, 947, formula 26.6.20.

SOURCE

cdfnonc.src

SEE ALSO **cdftnc**, **cdfchinc**

cdfgam

PURPOSE Computes the incomplete gamma function.

FORMAT *g* **= cdfgam(***x*,*intlim***);**

INPUT

x	N×K matrix of data.
intlim	L×M matrix, E×E 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   10.0000

intlim =  0.000000
          0.200000
          0.400000
          0.600000
          0.800000
          1.000000

g =  0.000000   0.000000   0.000000   0.000000
     0.472911   0.181269   0.00114848 2.35307E-014
     0.628907   0.329680   0.00792633 2.00981E-011
     0.726678   0.451188   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. ±6e-13
500	$\leq$	x	$\leq$	10,000	the absolute error is approx. ±3e-11
10,000	$<$	x			a Normal approximation is used and the absolute error is approx. ±3e-10

REFERENCES

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

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

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

cdfmvn

PURPOSE Computes multivariate Normal cumulative distribution function.

FORMAT *y* = **cdfmvn**(*x*,*r*);

INPUT
x K×L matrix, abscissae.
r K×K matrix, correlation matrix.

OUTPUT
y L×1 vector, $Pr(X < x|r)$.

SEE ALSO **cdfbvn**, **cdfn**, **lncdfmvn**

SOURCE lncdfn.src

cdfmvnce

PURPOSE Computes the complement of the multivariate Normal cumulative distribution function with error management

FORMAT {*y*,*err*,*retcode*} = **cdfmvnce**(*ctl*,*x*,*r*,*m*);

INPUT
ctl instance of a **cdfmControl** structure with members
ctl.maxEvaluations scalar, maximum number of evaluations
ctl.absErrorTolerance scalar absolute error tolerance

ctl.relative error tolerance

x N × K matrix, abscissae.

r K × K matrix, correlation matrix.

m K × 1 vector, means

OUTPUT

y L ×1 vector, $Pr(X > x|r, m)$.

err L × 1 vector, estimates of absolute error

retcode L × 1 vector, return codes,

0 normal completion with *err* < *ctl*.absErrorTolerance.

1 *err* > *ctl*.absErrorTolerance and *ctl*.maxEvaluations exceeded; increase *ctl*.maxEvaluations to decrease error.

2 K > 100 or K < 1.

3 *R* not positive semi-definite.

missing *R* not properly defined.

REMARKS **cdfmvne** evaluates the following integral

$$\Phi(x_i, R, m) = \frac{1}{\sqrt{|R|(2\pi)^m}} \int_{x_{i1}}^{\infty} \int_{x_{i2}}^{\infty} \cdots \int_{x_{iK}}^{\infty} e^{-\frac{1}{2}(z-m')'R^{-1}(z-m')} dz$$

SEE ALSO **cdfmvn2e**, **cdfmvnce**, **cdfmvte**

SOURCE cdfm.src

REFERENCES

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts", *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.

Genz, A., "Numerical computation of multivariate normal probabilities", *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfmvne

PURPOSE Computes multivariate Normal cumulative distribution function with error management

FORMAT {*y*,*err*,*retcode*} = **cdfmvne**(*ctl*,*x*,*r*,*m*);

INPUT

ctl instance of a **cdfmControl** structure with members

- *ctl*.maxEvaluations scalar, maximum number of evaluations
- *ctl*.absErrorTolerance scalar absolute error tolerance
- *ctl*.relative error tolerance

x N × K matrix, abscissae.

r K × K matrix, correlation matrix.

m K × 1 vector, means

OUTPUT

y L ×1 vector, $Pr(X < x|r, m)$.

err L × 1 vector, estimates of absolute error

retcode L × 1 vector, return codes,

0 normal completion with *err* < *ctl*.absErrorTolerance.

1 *err* > *ctl*.absErrorTolerance and *ctl*.maxEvaluations exceeded; increase *ctl*.maxEvaluations to decrease error.

2 K > 100 or K < 1.

3 *R* not positive semi-definite.

missing *R* not properly defined.

REMARKS **cdfmvne** evaluates the following integral

$$\Phi(x_i, R, m) = \frac{1}{\sqrt{|R|(2\pi)^m}} \int_{-\infty}^{x_{i1}} \int_{-\infty}^{x_{i2}} \cdots \int_{-\infty}^{x_{iK}} e^{-\frac{1}{2}(z-m')'R^{-1}(z-m')} dz$$

SEE ALSO **cdfmvne**, **cdfmvn2e**, **cdfmvtce**

SOURCE cdfm.src

REFERENCES

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts", *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.

Genz, A., "Numerical computation of multivariate normal probabilities", *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfmvn2e

PURPOSE Computes the multivariate Normal cumulative distribution function with error management over the range [a,b]

FORMAT {*y*,*err*,*retcode*} = **cdfmvn2e**(*ctl*,*a*,*b*,*r*,*m*);

INPUT

ctl	instance of a **cdfmControl** structure with members	
	ctl.maxEvaluations	scalar, maximum number of evaluations
	ctl.absErrorTolerance	scalar absolute error tolerance
	ctl.relative	error tolerance
a	N × K matrix, lower limits.	
b	N × K matrix, upper limits.	
r	K × K matrix, correlation matrix.	
m	K × 1 vector, means	

OUTPUT

y	L ×1 vector, $Pr(X > a \quad \text{and} \quad X < b \mid r, m)$.
err	L × 1 vector, estimates of absolute error
retcode	L × 1 vector, return codes,
0	normal completion with *err* < *ctl*.absErrorTolerance.
1	*err* > *ctl*.absErrorTolerance and *ctl*.maxEvaluations exceeded; increase *ctl*.maxEvaluations to decrease error.
2	K > 100 or K < 1.

3 *R* not positive semi-definite.
missing *R* not properly defined.

REMARKS **cdfmvne** evaluates the following integral

$$\Phi(a_i, b_i, R, m) = \frac{1}{\sqrt{|R|(2\pi)^m}} \int_{a_{i1}}^{b_{i1}} \int_{a_{i2}}^{b_{i2}} \cdots \int_{a_{iK}}^{b_{iK}} e^{-\frac{1}{2}(z-m')'R^{-1}(z-m')} dz$$

SEE ALSO **cdfmvne**, **cdfmvnce**, **cdfmvt2e**

SOURCE cdfm.src

REFERENCES 1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts", *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.

Genz, A., "Numerical computation of multivariate normal probabilities", *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfmvtce

PURPOSE Computes complement of multivariate Student's t cumulative distribution function with error management

FORMAT {*y*,*err*,*retcode*} = **cdfmvtce**(*ctl*,*x*,*R*,*m*);

INPUT *ctl* instance of a **cdfmControl** structure with members

ctl.maxEvaluations scalar, maximum number of evaluations
ctl.absErrorTolerance scalar absolute error tolerance
ctl.relative error tolerance

x N × K matrix, abscissae.

R K × K matrix, correlation matrix.

	m	K × 1 vector, noncentralities
	n	scalar, degrees of freedom
OUTPUT	*y*	L ×1 vector, $Pr(X > x\|r, m)$.
	err	L × 1 vector, estimates of absolute error
	retcode	L × 1 vector, return codes,

0 normal completion with *err* < *ctl*.absErrorTolerance.

1 *err* > *ctl*.absErrorTolerance and *ctl*.maxEvaluations exceeded; increase *ctl*.maxEvaluations to decrease error.

2 K > 100 or K < 1.

3 *R* not positive semi-definite.

missing *R* not properly defined.

REMARKS The central multivariate Student's t cdf for the i-th row of *x* is defined by

$$\begin{aligned} T(x_i, R, n) &= \frac{\Gamma(\frac{n+K}{2})}{\Gamma(\frac{n}{2})\sqrt{|R|(n\pi)^K}} \int_{x_{i1}}^{\infty}\int_{x_{i2}}^{\infty}\cdots\int_{x_iK}^{\infty}\left(1+\frac{z'\Sigma^{-1}z}{n}\right)^{-\frac{n+K}{2}} dz \\ &\equiv \frac{2^{1-\frac{n}{2}}}{\Gamma(\frac{n}{2})}\int_0^{\infty} s^{n-1}e^{-\frac{s^2}{2}}\Phi\left(-\infty, \frac{sx_i}{\sqrt{n}}, R\right) ds \end{aligned}$$

where

$$\Phi(x_i, R) = \frac{1}{\sqrt{|R|(2\pi)^m}}\int_{x_{i1}}^{\infty}\int_{x_{i2}}^{\infty}\cdots\int_{x_{iK}}^{\infty} e^{-\frac{1}{2}z'R^{-1}z} dz$$

For the noncentral cdf we have

$$T(x_i, R, n, m) = \frac{2^{1-\frac{n}{2}}}{\Gamma(\frac{n}{2})}\int_0^{\infty} s^{n-1}e^{-\frac{s^2}{2}}\Phi\left(\frac{sx_i}{\sqrt{n}} - m', \infty, R\right) ds$$

SEE ALSO **cdfmvt2e**, **cdfmvtce**, **cdfmvne**

SOURCE cdfm.src

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts", *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.

Genz, A., "Numerical computation of multivariate normal probabilities", *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfmvte

PURPOSE Computes multivariate Student's t cumulative distribution function with error management

FORMAT {*y*,*err*,*retcode*} = **cdfmvte**(*ctl*,*x*,*R*,*m*);

INPUT

ctl	instance of a **cdfmControl** structure with members
	ctl.maxEvaluations scalar, maximum number of evaluations
	ctl.absErrorTolerance scalar absolute error tolerance
	ctl.relative error tolerance
x	N × K matrix, abscissae.
R	K × K matrix, correlation matrix.
m	K × 1 vector, noncentralities
n	scalar, degrees of freedom

OUTPUT

y	L ×1 vector, $Pr(X < x\|r, m)$.
err	L × 1 vector, estimates of absolute error
retcode	L × 1 vector, return codes,
	0 normal completion with *err* < *ctl*.absErrorTolerance.

1 *err* > *ctl*.absErrorTolerance and *ctl*.maxEvaluations exceeded; increase *ctl*.maxEvaluations to decrease error.

2 K > 100 or K < 1.

3 *R* not positive semi-definite.

missing *R* not properly defined.

REMARKS The central multivariate Student's t cdf for the i-th row of *x* is defined by

$$\begin{aligned} T(x_i,R,n) &= \frac{\Gamma(\frac{n+K}{2})}{\Gamma(\frac{n}{2})\sqrt{|R|(n\pi)^K}} \int_{-\infty}^{x_{i1}} \int_{-\infty}^{x_{i2}} \cdots \int_{-\infty}^{x_iK} \left(1+\frac{z'\Sigma^{-1}z}{n}\right)^{-\frac{n+K}{2}} dz \\ &\equiv \frac{2^{1-\frac{n}{2}}}{\Gamma(\frac{n}{2})} \int_0^\infty s^{n-1} e^{-\frac{s^2}{2}} \Phi\left(-\infty, \frac{sx_i}{\sqrt{n}}, R\right) ds \end{aligned}$$

where

$$\Phi(x_i,R) = \frac{1}{\sqrt{|R|(2\pi)^m}} \int_{-\infty}^{x_{i1}} \int_{-\infty}^{x_{i2}} \cdots \int_{-\infty}^{x_{iK}} e^{-\frac{1}{2}z'R^{-1}z} dz$$

For the noncentral cdf we have

$$T(x_i,R,n,m) = \frac{2^{1-\frac{n}{2}}}{\Gamma(\frac{n}{2})} \int_0^\infty s^{n-1} e^{-\frac{s^2}{2}} \Phi\left(-\infty, \frac{sx_i}{\sqrt{n}} - m', R\right) ds$$

SEE ALSO **cdfmvte**, **cdfmvt2e**, **cdfmvnce**

SOURCE cdfm.src

1. Genz, A. and F. Bretz,"Numerical computation of multivariate

t-probabilities with application to power calculation of multiple contrasts", *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.

Genz, A., "Numerical computation of multivariate normal probabilities", *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfmvt2e

PURPOSE Computes multivariate Student's t cumulative distribution function with error management over [a,b]

FORMAT {*y*,*err*,*retcode*} = **cdfmvt2e**(*ctl*,*a*,*b*,*R*,*m*);

INPUT

ctl instance of a **cdfmControl** structure with members

- *ctl*.maxEvaluations scalar, maximum number of evaluations
- *ctl*.absErrorTolerance scalar absolute error tolerance
- *ctl*.relative error tolerance

a N × K matrix, lower limits.

b N × K matrix, upper limits.

R K × K matrix, correlation matrix.

m K × 1 vector, noncentralities

n scalar, degrees of freedom

OUTPUT

y L ×1 vector, $Pr(X > a \quad \text{and} \quad X < b|r, m)$.

err L × 1 vector, estimates of absolute error

retcode L × 1 vector, return codes,

- **0** normal completion with *err* < *ctl*.absErrorTolerance.
- **1** *err* > *ctl*.absErrorTolerance and *ctl*.maxEvaluations exceeded; increase *ctl*.maxEvaluations to decrease error.
- **2** K > 100 or K < 1.
- **3** *R* not positive semi-definite.

missing R not properly defined.

REMARKS The central multivariate Student's t cdf for the i-th row of x is defined by

$$\begin{aligned} T(x_i, R, n) &= \frac{\Gamma(\frac{n+K}{2})}{\Gamma(\frac{n}{2})\sqrt{|R|(n\pi)^K}} \int_{a_{i1}}^{b_{i1}} \int_{a_{i2}}^{b_{i2}} \cdots \int_{a_iK}^{b_iK} \left(1 + \frac{z'\Sigma^{-1}z}{n}\right)^{-\frac{n+K}{2}} dz \\ &\equiv \frac{2^{1-\frac{n}{2}}}{\Gamma(\frac{n}{2})} \int_0^\infty s^{n-1} e^{-\frac{s^2}{2}} \Phi\left(-\infty, \frac{sx_i}{\sqrt{n}}, R\right) ds \end{aligned}$$

where

$$\Phi(x_i, R) = \frac{1}{\sqrt{|R|(2\pi)^m}} \int_{a_{i1}}^{b_{i1}} \int_{a_{i2}}^{b_{i2}} \cdots \int_{a_iK}^{b_iK} e^{-\frac{1}{2}z'R^{-1}z} dz$$

For the noncentral cdf we have

$$T(x_i, R, n, m) = \frac{2^{1-\frac{n}{2}}}{\Gamma(\frac{n}{2})} \int_0^\infty s^{n-1} e^{-\frac{s^2}{2}} \Phi\left(\frac{sa_i}{\sqrt{n}} - m', \frac{sb_i}{\sqrt{n}} - m', R\right) ds$$

SEE ALSO **cdfmvte**, **cdfmvtce**, **cdfmvn2e**

SOURCE cdfm.src

1. Genz, A. and F. Bretz,"Numerical computation of multivariate t-probabilities with application to power calculation of multiple contrasts", *Journal of Statistical Computation and Simulation*, 63:361-378, 1999.

Genz, A., "Numerical computation of multivariate normal probabilities", *Journal of Computational and Graphical Statistics*, 1:141-149, 1992.

cdfn, cdfnc

PURPOSE **cdfn** computes the cumulative distribution function (cdf) of the Normal distribution. **cdfnc** computes 1 minus the cdf of the Normal distribution.

FORMAT *n* = **cdfn**(*x*);

nc = **cdfnc**(*x*);

INPUT *x* N×K matrix.

OUTPUT *n* N×K matrix.

nc N×K 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: **cdfn**(*x*) + **cdfnc**(*x*) = **1**. However, many applications expect **cdfn**(*x*) to approach 1, but never actually reach it. Because of this, we have capped the return value of **cdfn** at 1 - machine epsilon, or approximately 1 - 1.11e-16. As the relative error of **cdfn** is about ±5e-15 for **cdfn**(*x*) around 1, this does not invalidate the result. What it does mean is that for **abs**(*x*) > (approx.) 8.2924, the identity does not hold true. If you have a need for the uncapped value of **cdfn**, the following code will return it:

```
n = cdfn(x);
if n >= 1-eps;
    n = 1;
endif;
```

where the value of machine epsilon is obtained as follows:

```
x = 1;
do while 1-x /= 1;
    eps = x;
    x = x/2;
endo;
```

Note that this is an alternate definition of machine epsilon. Machine epsilon is usually defined as the smallest number such that 1 + machine epsilon > 1, which is about 2.23e-16. This defines machine epsilon as the smallest number such that 1 - machine epsilon < 1, or about 1.11e-16.

The **erf** and **erfc** functions are also provided, and may sometimes be more useful than **cdfn** and **cdfnc**.

EXAMPLE

```
x = { -2 -1 0 1 2 };
n = cdfn(x);
nc = cdfnc(x);
```

x	=	−2.00000	−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. ±5e-12
-10	$<$	x	$<$	0	**cdfn** has a relative error of approx. ±1e-13
0	$<$	x			**cdfn** has a relative error of approx. ±5e-15

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

REFERENCES

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

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

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

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

cdfn2

PURPOSE Computes interval of Normal cumulative distribution function.

FORMAT y **= cdfn2(**x,dx**);**

INPUT
- x — M×N matrix, abscissae.
- dx — K×L matrix, E×E conformable to x, intervals.

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

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 ±1e-14 implies that the answer is accurate to better than ±1 in the 14^{th} 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
```

SEE ALSO **lncdfn2**

SOURCE lncdfn.src

cdfni

PURPOSE Computes the inverse of the cdf of the Normal distribution.

FORMAT *x* **= cdfni(***p***);**

INPUT *p* N×K real matrix, Normal probability levels, $0 \leq p \leq 1$.

OUTPUT *x* N×K real matrix, Normal deviates, such that **cdfn**(*x*) = *p*

REMARKS **cdfn**(**cdfni**(*p*)) = *p* to within the errors given below:

		p	$\leq$	4.6e-308	-37.5 is returned
4.6e-308	$<$	p	$<$	5e-24	accurate to ±5 in 12^{th} digit
5e-24	$<$	p	$<$	0.5	accurate to ±1 in 13^{th} digit
0.5	$<$	p	$<$	1 - 2.22045e-16	accurate to ±5 in 15^{th} digit
		p	$\geq$	1 - 2.22045e-16	8.12589. . . is returned

SEE ALSO **cdfn**

cdftc

PURPOSE Computes the complement of the cdf of the Student's *t* distribution.

FORMAT *y* **= cdftc(***x***,***n***);**

INPUT
x N×K matrix.
n L×M matrix, E×E 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:

$$\begin{aligned} -\infty &< x &< +\infty \\ n &> 0 \end{aligned}$$

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 = cdftc(x,n);
```

```
y =
0.473165
0.447100
0.422428
0.399555
```

SEE ALSO **cdftci**

TECHNICAL NOTES

For results greater than 0.5e-30, the absolute error is approx. ±1e-14 and the relative error is approx. ±1e-12. If you multiply the relative error by the result, then take the minimum of that and the absolute error, you have the maximum actual error for any result. Thus, the actual error is approx. ±1e-14 for results greater than 0.01. For results less than 0.01, the actual error will be less. For example, for a result of 0.5e-30, the actual error is only ±0.5e-42.

REFERENCES

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

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

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

cdftci

PURPOSE Computes the inverse of the complement of the Student's *t* cdf.

FORMAT *x* = **cdftci**(*p*,*n*);

INPUT
p N×K real matrix, complementary Student's *t* probability levels, $0 \leq p \leq 1$.

n L×M real matrix, degrees of freedom, $n \geq 1$, *n* need not be integral. E×E conformable with *p*.

OUTPUT
x max(N,L) by max(K,M) real matrix, Student's *t* deviates, such that **cdftc**(*x*,*n*) = *p*.

REMARKS **cdftc(cdftci(***p*,*n***)) =** *p* to within the errors given below:

0.5e-30	<	*p*	<	0.01	accurate to ±1 in 12th digit
0.01	<	*p*			accurate to ±1e-14

Extreme values of arguments can give rise to underflows, but no overflows are generated.

SEE ALSO **cdftc**

cdftnc

PURPOSE The integral under noncentral Student's *t* distribution, from $-\infty$ 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* = **cdftnc**(*x*,*v*,*d*);

INPUT
x N×1 vector, values of upper limits of integrals.

v scalar, degrees of freedom, v > 0.

	d	scalar, noncentrality parameter. This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. (See Scheffe, *The Analysis of Variance*, App. IV, 1959.)
OUTPUT	*y*	N×1 vector, integrals from $-\infty$ to *x* of noncentral *t*.

REMARKS **cdftc(*x*,*v*) = 1 - cdftnc(*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.

SOURCE cdfnonc.src

SEE ALSO **cdffnc**, **cdfchinc**

cdftvn

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

FORMAT *c* = **cdftvn**(*x1*,*x2*,*x3*,*rho21*,*rho31*,*rho32*);

INPUT	*x1*	N×1 vector of upper limits of integration for variable 1.
	x2	N×1 vector of upper limits of integration for variable 2.
	x3	N×1 vector of upper limits of integration for variable 3.
	rho21	scalar or N×1 vector of correlation coefficients between the two variables *x1* and *x2*.

rho31 scalar or N×1 vector of correlation coefficients between the two variables *x2* and *x3*.

rho32 scalar or N×1 vector of correlation coefficients between the two variables *x1* and *x3*.

OUTPUT

c N×1 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.

REMARKS Allowable ranges for the arguments are:

$$\begin{array}{lll} -\infty & < x1 & < +\infty \\ -\infty & < x2 & < +\infty \\ -\infty & < x3 & < +\infty \\ -1 & < rho21 & < 1 \\ -1 & < rho31 & < 1 \\ -1 & < rho32 & < 1 \end{array}$$

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 **(***rho21***,***rho31***,***rho32***)** must also be the same length, and this length must be N or 1. If it is 1, then these values will be expanded to apply to all values of *x1*,*x2*,*x3*. All inputs must be column vectors.

To find the integral under a general trivariate density, with *x1*, *x2*, and *x3* having nonzero means and any positive standard deviations, transform by subtracting the mean and dividing by the standard deviation. For example:

```
x1 = ( x1 - meanc(x1) ) / stdc(x1);
```

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

SEE ALSO **cdfn**, **cdfbvn**

REFERENCES 1. Daley, D.J. "Computation of Bi- and Tri-variate Normal Integral." *Appl. Statist.* Vol. 23, No. 3, 1974, 435-38.

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

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:\
```


ceil

PURPOSE Round up toward $+\infty$.

FORMAT *y* = **ceil**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* N×K matrix.

REMARKS This rounds every element in the matrix *x* to an integer. The elements are rounded up toward $+\infty$.

EXAMPLE
```
x = 100*rndn(2,2);
y = ceil(x);
```

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

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

SEE ALSO **floor**, **trunc**

ChangeDir

PURPOSE Changes the working directory.

FORMAT *d* = **ChangeDir(***s***);**

INPUT *s* string, directory to change to.

OUTPUT *d* string, new working directory, or null string if change failed.

SEE ALSO **chdir cdir**

chdir

PURPOSE Changes working directory.

FORMAT **chdir** *dirstr***;**

INPUT *dirstr* literal or ^string, directory to change to.

REMARKS This is for interactive use. Use **ChangeDir** in a program.

If the directory change fails, **chdir** prints an error message.

The working directory is listed in the status report on UNIX.

SEE ALSO **changedir cdir**

chiBarSquare

PURPOSE Compute compute the probability for a chi-bar square statistic from an hypothesis involving parameters under constraints

FORMAT *Slprob* = **chiBarSquare**(*SL*,*H*,*a*,*b*,*c*,*d*,*bounds*);

INPUT

SL	scalar, chi-bar square statistic
H	K×K matrix, positive covariance matrix
G	K×K matrix, cross-product matrix of the first derivatives by observation. If not available set to *H*.
grad	K×1 vector, gradient of loglikelihood with respect to parameters.
A	M×K matrix, linear equality constraint coefficients.
B	M×1 vector, linear equality constraint constants. These arguments specify the linear equality constraints of the following type: `A * X = B` where *X* is the K×1 parameter vector.
C	M×K matrix, linear inequality constraint coefficients.
D	M×1 vector, linear inequality constraint constants. These arguments specify the linear inequality constraints of the following type: `C * X >= D` where *X* is the K×1 parameter vector.
bounds	K×2 matrix, bounds on parameters. The first column contains the lower bounds, and the second column the upper bounds.
psi	indices of the set of parameters in the hypothesis.

OUTPUT

SLprob	scalar, probability of *SL*.

REMARKS See Silvapulle and Sen, *Constrained Statistical Inference*, page 75 for further details about this function. Let $Z_{p\times 1}$ $N(0, V)$ where V is a positive definite covariance matrix. Define

$$\bar{\chi}^2(V, C) = Z'V^{-1}Z - \min_{\theta \in C}(Z-\theta)'V^{-1}(Z-\theta)$$

C is a closed convex cone describing a set of constraints. **ChiBarSquare** computes the probability of this statistic given V and C.

EXAMPLE

```
V = {
 0.0005255598    -0.0006871606    -0.0003191342,
-0.0006871606     0.0037466205     0.0012285813,
-0.0003191342     0.0012285813     0.0009081412 };

SL = 3.860509;

Bounds = { 0 200, 0 200, 0 200 };

vi = invpd(v);

SLprob = chiBarSquare(SL,Vi,0,0,0,0,bounds);
```

```
slprob =  0.10885000
```

SOURCE hypotest.src

chol

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

FORMAT *y* = **chol**(*x*);

INPUT *x* N×N matrix.

OUTPUT *y* N×N 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;
```

```
x =    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
```

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

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

SEE ALSO **crout**, **solpd**

choldn

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

FORMAT *r* = **choldn**(*C*,*x*);

INPUT

C	K×K upper triangular matrix.
x	N×K matrix, the rows to downdate *C* with.

OUTPUT

r	K×K upper triangular matrix, the downdated matrix.

REMARKS *C* should be a Cholesky factorization.

choldn(*C*,*x*) is equivalent to **chol**(*C*′*C* - *x*′*x*), but **choldn** is numerically much more stable.

WARNING: it is possible to render a Cholesky factorization non-positive definite with **choldn**. You should keep an eye on the ratio of the largest diagonal element of *r* to the smallest—if it gets very large, *r* may no longer be positive definite. This ratio is a rough estimate of the condition number of the matrix.

EXAMPLE

```
let C[3,3] = 20.16210005  16.50544413  9.86676135
             0            11.16601462  2.97761666
             0            0            11.65496052;
```

```
let x[2,3] = 1.76644971   7.49445820   9.79114666
             6.87691156   4.41961438   4.32476921;
r = choldn(C,x);
```

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

SEE ALSO **cholup chol**

cholsol

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

FORMAT *x* **= cholsol(***b***,***C***);**

INPUT
b N×K matrix.
C N×N matrix.

OUTPUT
x N×K 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
```

SEE ALSO **chol**

cholup

PURPOSE Performs a Cholesky update of one or more rows on an upper triangular matrix.

FORMAT *r* **= cholup(***C*,*x***);**

INPUT

C	K×K upper triangular matrix.
x	N×K matrix, the rows to update *C* with.

OUTPUT

r	K×K 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 **choldn**

chrs

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

FORMAT *y* = **chrs**(*x*);

INPUT *x* N×K 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;
print chrs(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 = chrs(67~65~84);
print y;

CAT
```

SEE ALSO **vals**, **ftos**, **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**

clearg

PURPOSE Clears global symbols by setting them equal to scalar zero.

FORMAT **clearg** *a*,*b*,*c*;

OUTPUT *a,b,c* scalar global matrices containing 0.

REMARKS **clearg x;** is equivalent to **x = 0;**, where **x** is understood to be a global symbol. **clearg** can be used to initialize symbols not previously referenced. This command can be used inside of procedures to clear global matrices. It will ignore any locals by the same name.

EXAMPLE
```
x = 45;
clearg x;
```

```
x =  0.0000000
```

SEE ALSO **clear**, **delete**, **new**, **show**, **local**

close

PURPOSE Closes a **GAUSS** file.

FORMAT *y* **= close(***handle***);**

INPUT *handle* scalar, the file handle given to the file when it was opened with the **open**, **create**, or **fopen** command.

OUTPUT *y* scalar, 0 if successful, -1 if unsuccessful.

REMARKS *handle* is the scalar file handle created when the file was opened. It will contain an integer which can be used to refer to the file.

close will close the file specified by handle, and will return a 0 if successful and a -1 if not successful. The handle itself is not affected by **close** unless the return value of **close** is assigned to it.

If **f1** is a file handle and it contains the value 7, then after:

```
call close(f1);
```

the file will be closed but **f1** will still have the value 7. The best procedure is to do the following:

```
f1 = close(f1);
```

This will set **f1** to 0 upon a successful close.

It is important to set unused file handles to zero because both **open** and **create** check the value that is in a file handle before they proceed with the process of opening a file. During **open** or **create**, if the value that is in the file handle matches that of an already open file, the process will be aborted and a **File already open** error message will be given. This gives you some protection against opening a second file with the same handle as a currently open file. If this happened, you would no longer be able to access the first file.

An advantage of the **close** function is that it returns a result which can be tested to see if there were problems in closing a file. The most common reason for having a problem in closing a file is that the disk on which the file is located is no longer in the disk drive—or the handle was invalid. In both of these cases, **close** will return a -1.

Files are not automatically closed when a program terminates. This allows users to run a program that opens files, and then access the files from interactive mode after the program has been run. Files are automatically closed when **GAUSS** exits to the operating system or when a program is terminated with the **end** statement. **stop** will terminate a program but not close files.

As a rule it is good practice to make **end** the last statement in a program, unless further access to the open files is desired from interactive mode. You should close files as soon as you are done writing to them to protect against data loss in the case of abnormal termination of the program due to a power or equipment failure.

The danger in not closing files is that anything written to the files may be lost. The disk directory will not reflect changes in the size of a file until the file is closed and system buffers may not be flushed.

EXAMPLE

```
open f1 = dat1 for append;
y = writer(f1,x);
f1 = close(f1);
```

SEE ALSO **closeall**

closeall

PURPOSE Closes all currently open **GAUSS** files.

FORMAT **closeall;**

closeall *list_of_handles*;

REMARKS *list_of_handles* is a comma-delimited list of file handles.

closeall with no specified list of handles will close all files. The file handles will not be affected. The main advantage of using **closeall** is ease of use; the file handles do not have to be specified, and one statement will close all files.

When a list of handles follows **closeall**, all files are closed and the file handles listed are set to scalar 0. This is safer than **closeall** without a list of handles because the handles are cleared.

It is important to set unused file handles to zero because both **open** and **create** check the value that is in a file handle before they proceed with the process of opening a file. During **open** or **create**, if the value that is in the file handle matches that of an already open file, the process will be aborted and a **File already open** error message will be given. This gives you some protection against opening a second file with the same handle as a currently open file. If this happened, you would no longer be able to access the first file.

Files are not automatically closed when a program terminates. This allows users to run a program that opens files, and then access the files from interactive mode after the program has been run. Files are automatically closed when **GAUSS** exits to the operating system or when a program is terminated with the **end** statement. **stop** will terminate a program but not close files.

As a rule it is good practice to make **end** the last statement in a program, unless further access to the open files is desired from interactive mode. You should close files as soon as you are done writing to them to protect against data loss in the case of abnormal termination of the program due to a power or equipment failure.

The danger in not closing files is that anything written to the files may be lost. The disk directory will not reflect changes in the size of a file until the file is closed and system buffers may not be flushed.

EXAMPLE

```
open f1 = dat1 for read;
open f2 = dat1 for update;
x = readr(f1,rowsf(f1));
x = sqrt(x);
call writer(f2,x);
closeall f1,f2;
```

SEE ALSO **close**, **open**

cls

PURPOSE Clears the window.

FORMAT **cls;**

PORTABILITY **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.

REMARKS This command clears the window and locates the cursor at the upper left hand corner of the window.

SEE ALSO **locate**

code

PURPOSE Allows a new variable to be created (coded) with different values depending upon which one of a set of logical expressions is true.

FORMAT *y* **= code(***e***,***v***);**

INPUT

e N×K 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)×1 vector containing the values to be assigned to the new variable.

OUTPUT

y N×1 vector containing the new values.

REMARKS If none of the K expressions is true, the new variable is assigned the default value, which is given by the last element of *v*.

EXAMPLE

```
let x1 = 0   /* column vector of original values */
         5
         10
         15
         20;

let v =  1   /* column vector of new values */
         2
         3;  /* the last element of v is the "default" */

e1 = (0 .lt x1) .and (x1 .le 5);     /* expression 1 */
e2 = (5 .lt x1) .and (x1 .le 25);    /* expression 2 */

e = e1~e2;   /* concatenate e1 & e2 to make a 1,0 mask
             :: with one less column than the number
             :: of new values in v.
             */

y = code(e,v);
```

x1[5,1] =

0
5
10
15
20

(column vector of original values)

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

e[5,2] =

0	0
1	0
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 so on. If there are only zeros in the row, the last element of *v* is used. This is the default value.

If there is more than one 1 in any row of *e*, the function will terminate with an error message.

SOURCE `datatran.src`

SEE ALSO **recode**, **substute**

code (dataloop)

PURPOSE Creates new variables with different values based on a set of logical expressions.

FORMAT **code** [[**#**]] [[**$**]] *var* [[**default** *defval*]] **with**
val_1 **for** *expression_1*,
val_2 **for** *expression_2*,
.
.
.
val_n **for** *expression_n*;

INPUT

var	literal, the new variable name.
defval	scalar, the default value if none of the expressions are TRUE.
val	scalar, value to be used if corresponding expression is TRUE.

expression logical scalar-returning expression that returns nonzero TRUE or zero FALSE.

REMARKS If '**$**' is specified, the new variable will be considered a character variable. If '**#**' or nothing is specified, the new variable will be considered numeric.

The logical expressions must be mutually exclusive, i.e., only one may return TRUE for a given row (observation).

Any variables referenced must already exist, either as elements of the source data set, as externs, or as the result of a previous **make**, **vector**, or **code** statement.

If no default value is specified, 999 is used.

EXAMPLE
```
code agecat default 5 with
    1 for age < 21,
    2 for age >= 21 and age < 35,
    3 for age >= 35 and age < 50,
    4 for age >= 50 and age < 65;


code $ sex with
    "MALE" for gender =\,= 1,
    "FEMALE" for gender =\,= 0;
```

SEE ALSO **recode (dataloop)**

cols

PURPOSE Returns the number of columns in a matrix.

FORMAT *y* **= cols(***x***);**

INPUT *x* N×K matrix or sparse matrix.

OUTPUT *y* number of columns in *x*.

REMARKS If *x* is an empty matrix, **rows(***x***)** and **cols(***x***)** both return 0.

EXAMPLE
```
x = rndn(100,3);
y = cols(x);
```

```
y =  3.000000
```

SEE ALSO **rows**, **colsf**, **show**

colsf

PURPOSE Returns the number of columns in a **GAUSS** data (`.dat`) file or **GAUSS** matrix (`.fmt`) file.

FORMAT *yf* **= colsf(***fh***);**

INPUT *fh* file handle of an open file.

OUTPUT *yf* number of columns in the file that has the handle *fh*.

REMARKS In order to call **colsf** on a file, the file must be open.

EXAMPLE
```
create fp = myfile with x,10,4;
b = colsf(fp);
```

```
b =  10.000000
```

SEE ALSO **rowsf**, **cols**, **show**

combinate

PURPOSE Computes combinations of *N* things taken *K* at a time.

FORMAT *y* = **combinate**(*N*,*K*);

INPUT

N	scalar.
K	scalar.

OUTPUT

y	M×K matrix, where M is the number of combinations of *N* things taken *K* at a time.

REMARKS "Things" are represented by a sequence of integers from 1 to *N*, and the integers in each row of *Y* are the combinations of those integers taken *K* at a time.

EXAMPLE

```
n = 4;
k = 2;
y = combinate(n,k);

print y;

1.0000    2.0000
1.0000    3.0000
1.0000    4.0000
2.0000    3.0000
2.0000    4.0000
3.0000    4.0000
```

SEE ALSO **combinated**, **numCombinations**

combinated

PURPOSE Writes combinations of *N* things taken *K* at a time to a **GAUSS** data set.

FORMAT *ret* = **combinated**(*fname*,*vnames*,*N*,*K*);

INPUT

fname	string, file name.
vname	1×1 or K×1 string array, names of columns in data set. If 1×1 string, names will have column number appended. If null string, names will be X1, X2, ...
N	scalar.
K	scalar.

OUTPUT

ret	scalar, if data set was successfully written, *ret* = number of rows written to data set. Otherwise, one of the following:
	0 file already exists.
	-1 data set couldn't be created.
	-*n* the $(n\text{-}1)^{th}$ write to the data set failed.

REMARKS The rows of the data set in *fname* contain sequences of the integers from 1 to *N* in combinations taken *K* at a time.

EXAMPLE

```
vnames = "Jim"$|"Harry"$|"Susan"$|"Wendy";

k = 2;
m = combinated("couples",vnames,rows(vnames),k);

print m;

6.0000

open f0 = "couples";
y = readr(f0,m);
```

```
names = getnamef(f0);
f0=close(f0);

for i(1,rows(y),1);
    print names[y[i,.]]';
endfor;


Jim    Harry
Jim    Susan
Jim    Wendy
Harry  Susan
Harry  Wendy
Susan  Wendy

print y;

1.0000    2.0000
1.0000    3.0000
1.0000    4.0000
2.0000    3.0000
2.0000    4.0000
3.0000    4.0000
```

SEE ALSO **combinate**, **numCombinations**

comlog

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

FORMAT **comlog** [[**file=***filename*]] [[**on**|**off**|**reset**]];

INPUT *filename* literal or ^string.

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.

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

comlog off closes the log file and turns off command logging.

comlog reset turns on command logging to the current log file, resetting the log file by deleting any previous commands.

Interactive mode statements are always logged into the file specified in the **log_file** configuration variable, regardless of the state of **comlog**.

The command **comlog file=***filename* selects the file but does not turn on logging.

The command **comlog off** will turn off logging. The filename will remain the same. A subsequent **comlog on** will cause logging to resume. A subsequent **comlog reset** will cause the existing contents of the log file to be destroyed and a new file created.

The command **comlog** by itself will cause the name and status of the current log file to be printed in the window.

compile

PURPOSE Compiles a source file to a compiled code file. See also Chapter 19.

FORMAT **compile** *source fname*;

INPUT *source* literal or ^string, the name of the file to be compiled.

fname literal or ^string, optional, the name of the file to be created. If not given, the file will have the same filename and path as *source*. It will have a `.gcg` extension.

REMARKS The *source* file will be searched for in the **src_path** if the full path is not specified and it is not present in the current directory.

The *source* file is a regular text file containing a **GAUSS** program. There can be references to global symbols, **Run-Time Library** references, etc.

If there are **library** statements in *source*, they will be used during the compilation to locate various procedures and symbols used in the program. Since all of these library references are resolved at compile time, the **library** statements are not transferred to the compiled file. The compiled file can be run without activating any libraries.

If you do not want extraneous stuff saved in the compiled image, put a **new** at the top of the *source* file or execute a **new** in interactive mode before compiling.

The program saved in the compiled file can be run with the **run** command. If no extension is given, the **run** command will look for a file with the correct extension for the version of **GAUSS**. The **src_path** will be used to locate the file if the full path name is not given and it is not located on the current directory.

When the compiled file is **run**, all previous symbols and procedures are deleted before the program is loaded. It is therefore unnecessary to execute a **new** before **run**'ning a compiled file.

If you want line number records in the compiled file you can put a **#lineson** statement in the *source* file or turn line tracking on from the Options menu.

Don't try to include compiled files with **#include**.

EXAMPLE

```
compile qxy.e;
```

In this example, the **src_path** would be searched for `qxy.e`, which would be compiled to a file called `qxy.gcg` on the same subdirectory `qxy.e` was found.

```
compile qxy.e xy;
```

In this example, the **src_path** would be searched for `qxy.e` which would be compiled to a file called `xy.gcg` on the current subdirectory.

SEE ALSO **run**, **use**, **saveall**

complex

PURPOSE Converts a pair of real matrices to a complex matrix.

FORMAT *z* **= complex(***xr***,***xi***);**

INPUT

xr	N×K real matrix, the real elements of *z*.
xi	N×K real matrix or scalar, the imaginary elements of *z*.

OUTPUT

z	N×K complex matrix.

EXAMPLE

```
x = { 4 6,
      9 8 };

y = { 3 5,
      1 7 };

t = complex(x,y);
```

$$\texttt{t} = \begin{matrix} 4.0000000 + 3.0000000i & 6.0000000 + 5.0000000i \\ 9.0000000 + 1.0000000i & 8.0000000 + 7.0000000i \end{matrix}$$

SEE ALSO **imag**, **real**

con

PURPOSE Requests input from the keyboard (console), and returns it in a matrix.

FORMAT *x* = **con**(*r*,*c*);

INPUT
r scalar, row dimension of matrix.

c scalar, column dimension of matrix.

OUTPUT
x *r*×*c* matrix.

REMARKS **con** gets input from the active window. **GAUSS** will not "see" any input until you press ENTER, so follow each entry with an ENTER.

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

If *r* and *c* are both set to 1, **con** will cause a question mark to appear in the window, indicating that it is waiting for a scalar input.

Otherwise, **con** will cause the following prompt to appear in the window:

```
- [1,1]
```

indicating that it is waiting for the [1,1] element of the matrix to be inputted. The - means that **con** will move horizontally through the matrix as you input the matrix elements. To change this or other options, or to move to another part of the matrix, use the following commands:

u	up one row	**U**	first row
d	down one row	**D**	last row
l	left one column	**L**	first column
r	right one column	**R**	last column
t	first element		
b	last element		
g #, #	goto element		
g #	goto element of vector		
h	move horizontally, default		
v	move vertically, default		
****	move diagonally, default		
s	show size of matrix		
n	display element as numeric, default		
c	display element as character		
e	exp(1)		
p	pi		
.	missing value		
?	show help screen		
x	exit		

If the desired matrix is 1×N or N×1, then **con** will automatically exit after the last element has been entered, allowing you to input the vector quickly.

If the desired matrix is N×K, you will need to type '**x**' to exit when you have finished entering the matrix data. If you exit before all elements have been entered, unspecified elements will be zeroed out.

Use a leading single quote for character input.

EXAMPLE

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

```
? 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**, **let**, **load**,

cond

PURPOSE Computes the condition number of a matrix using the singular value decomposition.

FORMAT *c* = **cond**(*x*);

INPUT *x* N×K matrix.

OUTPUT *c* scalar, an estimate of the condition number of *x*. This equals the ratio of the largest singular value to the smallest. If the smallest singular value is zero or not all of the singular values can be computed, the return value is 10^{300}.

EXAMPLE

```
x = { 4 2 6,
      8 5 7,
      3 8 9 };

y = cond(x);
```

```
y =  9.8436943
```

SOURCE svd.src

conj

PURPOSE Returns the complex conjugate of a matrix.

FORMAT *y* = **conj**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* N×K 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);
```

$$\mathtt{x} = \begin{matrix} 1.0000000 + 9.0000000i & 2.0000000 \\ 4.0000000 + 4.0000000i & 0.0000000 + 5.0000000i \\ 0.0000000 + 7.0000000i & 8.0000000 - 2.0000000i \end{matrix}$$

$$\mathtt{y} = \begin{matrix} 1.0000000 - 9.0000000i & 2.0000000 \\ 4.0000000 - 4.0000000i & 0.0000000 - 5.0000000i \\ 0.0000000 - 7.0000000i & 8.0000000 + 2.0000000i \end{matrix}$$

cons

PURPOSE Retrieves a character string from the keyboard.

FORMAT *x* **= cons;**

OUTPUT *x* string, the characters entered from the keyboard

REMARKS *x* is assigned the value of a character string typed in at the keyboard. The program will pause to accept keyboard input. The maximum length of the string that can be entered is 254 characters. The program will resume execution when the ENTER key is pressed.

EXAMPLE
```
x = cons;
```

At the cursor enter:

```
probability
```

x = "probability"

SEE ALSO **con**

ConScore

PURPOSE Compute local score statistic and its probability for hypotheses involving parameters under constraints

FORMAT { *SL*, *Slprob* } **= ConScore(***H*,*G*,*grad*,*a*,*b*,*c*,*d*,*bounds*,*psi***);**

INPUT

H K×K matrix, Hessian of loglikelihood with respect to parameters.

G K×K matrix, cross-product matrix of the first derivatives by observation. If not available set to *H*.

grad K×1 vector, gradient of loglikelihood with respect to parameters.

A M×K matrix, linear equality constraint coefficients.

B M×1 vector, linear equality constraint constants.

These arguments specify the linear equality constraints of the following type:

```
A * X = B
```

where *X* is the K×1 parameter vector.

C M×K matrix, linear inequality constraint coefficients.

D M×1 vector, linear inequality constraint constants.

These arguments specify the linear inequality constraints of the following type:

```
C * X >= D
```

where *X* is the K×1 parameter vector.

bounds K×2 matrix, bounds on parameters. The first column contains the lower bounds, and the second column the upper bounds.

psi indices of the set of parameters in the hypothesis.

OUTPUT

SL scalar, local score statistic of hypothesis.

SLprob scalar, probability of *SL*.

REMARKS

ConScore computes the local score statistic for the hypothesis $H(\theta) = 0$ vs. $H(\theta) \geq 0$, where θ is the vector of estimated parameters, and $H()$ is a constraint function of the parameters.

First, the model with $H(\theta) = 0$ is estimated, and the Hessian and optionally the cross-product of the derivatives is computed. Also, the gradient vector is computed.

Next, the constraint arguments are set to $H(\theta) \geq 0$.

EXAMPLE This example is from Silvapulle and Sen, *Constrained Statistical Inference*, page 181-3. It computes the local score statistic and probability for an ARCH model. It tests the null hypothesis of no arch effects against the alternative of arch effects subject to their being constrained to be positive.

The Hessian, H, cross-product matrix, G, and the gradient vector, grad, are generated by an estimation using **Sqpsolvemt** where the model is an ARCH model with the arch parameters constrained to be zero.

```
#include sqpsolvemt.sdf

/* data */

struct DS d0;
d0 = reshape(dsCreate,2,1);

load z0[] = aoi.asc;
z = packr(lagn(251*ln(trimr(z0,1,0)./trimr(z0,0,1)),0|1|2|3|4));
d0[1].dataMatrix = z[.,1];
d0[2].dataMatrix = z[.,2:5];


/* control structure */

struct sqpsolvemtControl c0;
c0 = sqpSolveMTcontrolCreate;

/*
** constraints setting arch parameter equal to zero
** for H(theta) = 0
*/

c0.A = zeros(3,6) ~ eye(3);
c0.B = zeros(3,1);

c0.covType = 2;  // causes cross-product of Jacobian
```

```
                    // to be computed which is needed for
                    // ConScore

struct PV p0;
p0 = pvPack(pvCreate,.08999,"constant");
p0 = pvPack(p0,.25167|-.12599|.09164|.07517,"phi");
p0 = pvPack(p0,3.22713,"omega");
p0 = pvPack(p0,0|0|0,"arch");


struct sqpsolvemtOut out0;
out0 = sqpsolvemt(&lpr,p0,d0,c0);


/*
** set up constraints for H(theta) >= 0
*/

bounds = { -1e256 1e256,
           -1e256 1e256,
           -1e256 1e256,
           -1e256 1e256,
           -1e256 1e256,
           -1e256 1e256,
              0 1e256,
              0 1e256,
              0 1e256 };

H = out0.hessian;
G = out0.xproduct;
grad = -out0.gradient;  // minus because -logl in log-likelihood

psi = { 7, 8, 9 };

{ sl, slprob } = ConScore(H,G,grad,0,0,0,0,bounds,psi);
```

```
sl =  3.8605086

slprob =  0.10410000
```

SOURCE hypotest.src

continue

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

FORMAT **continue;**

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

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

REMARKS This command works just as in **C**.

contour

PURPOSE Graphs a matrix of contour data.

LIBRARY pgraph

FORMAT **contour**(*x*,*y*,*z*);

INPUT

x	1×K vector, the X axis data. K must be odd.
y	N×1 vector, the Y axis data. N must be odd.
z	N×K matrix, the matrix of height data to be plotted.

GLOBAL INPUT

_plev	K×1 vector, user-defined contour levels for **contour**. Default 0.
_pzclr	N×1 or N×2 vector. This controls the Z level colors. See **surface** for a complete description of how to set this global.

REMARKS A vector of evenly spaced contour levels will be generated automatically from the *z* matrix data. Each contour level will be labeled. For unlabeled contours, use **ztics**.

To specify a vector of your own unequal contour levels, set the vector **_plev** before calling **contour**.

To specify your own evenly spaced contour levels, see **ztics**.

SOURCE pcontour.src

SEE ALSO **surface**

conv

PURPOSE Computes the convolution of two vectors.

FORMAT *c* = **conv**(*b*,*x*,*f*,*l*);

INPUT

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

OUTPUT

c	Q×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
```

```
z2 = 16
     34
     60
     61
```

SEE ALSO **polymult**

convertsatostr

PURPOSE Converts a 1×1 string array to a string.

FORMAT *str* = **convertsatostr**(*sa*);

INPUT *sa* 1×1 string array.

OUTPUT *str* string, *sa* converted to a string.

SEE ALSO **convertstrtosa**

convertstrtosa

PURPOSE Converts a string to a 1×1 string array.

FORMAT *sa* = **convertstrtosa**(*str*);

INPUT *str* string.

OUTPUT *sa* 1×1 string array, *str* converted to a string array.

EXAMPLE
```
str = "This is a string";
```

```
z = convertstrtosa(str);
```

SEE ALSO **convertsatostr**

corrm, corrvc, corrx

PURPOSE Computes a population correlation matrix.

FORMAT *cx* **= corrm(***m***);**

cx **= corrvc(***vc***);**

cx **= corrx(***x***);**

INPUT

m	K×K moment ($x'x$) matrix. A constant term MUST have been the first variable when the moment matrix was computed.
vc	K×K variance-covariance matrix (of data or parameters).
x	N×K matrix of data.

OUTPUT

cx	P×P correlation matrix. For **corrm**, P = K-1. For **corrvc** and **corrx**, P = K.

REMARKS Computes population correlation/covariance matrix, that is, it divides by N-1, rather than N. For sample correlation/covariance matrix which uses N rather than N-1, see **corrms** and **corrxs**.

SOURCE corr.src

SEE ALSO **momentd**, **corrms**, **corrxs**

corrms, corrxs

PURPOSE Computes sample correlation matrix.

FORMAT *cx* = **corrms**(*m*);

cx = **corrxs**(*x*);

INPUT
m K×K moment ($x'x$) matrix. A constant term MUST have been the first variable when the moment matrix was computed.
x N×K matrix of data.

OUTPUT
cx P×P correlation matrix. For **corrms**, P = K-1. For **corrxs**, P = K.

REMARKS Computes sample correlation/covariance matrix, that is, it divides the sample size, N, rather than N-1. For population correlation/covariance matrix which uses N-1 rather than N, see **corrm** or **corrx**.

SOURCE corrs.src

SEE ALSO **momentd**, **corrm**, **corrx**

cos

PURPOSE Returns the cosine of its argument.

FORMAT *y* = **cos**(*x*);

INPUT
x N×K matrix.

OUTPUT
y N×K matrix containing the cosines of the elements of *x*.

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

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

EXAMPLE
```
x = { 0, .5, 1, 1.5 };
y = cos(x);
```

```
y =
1.00000000
0.87758256
0.54030231
0.07073720
```

SEE ALSO **atan**, **atan2**, **pi**

cosh

PURPOSE Computes the hyperbolic cosine.

FORMAT y **= cosh(**x**);**

INPUT x N×K matrix.

OUTPUT y N×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 Counts the numbers of elements of a vector that fall into specified ranges.

FORMAT *c* = **counts**(*x*,*v*);

INPUT *x* N×1 vector containing the numbers to be counted.

v P×1 vector containing breakpoints specifying the ranges within which counts are to be made. The vector *v* MUST be sorted in ascending order.

OUTPUT *c* P×1 vector, the counts of the elements of *x* that fall into the regions:

$$\begin{aligned} & x \leq v[1], \\ v[1] < \; & x \leq v[2], \\ & \vdots \end{aligned}$$

$$v[p-1] < \quad x \quad \leq v[p]$$

REMARKS If the maximum value of x is greater than the last element (the maximum value) of v, the sum of the elements of the result, c, will be less than N, the total number of elements in x.

If

$$x = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{matrix} \quad \text{and} \quad v = \begin{matrix} 4 \\ 5 \\ 8 \end{matrix}$$

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* **= countwts(***x*,*v*,*w***);**

INPUT

x N×1 vector, the numbers to be counted.

v P×1 vector, the breakpoints specifying the ranges within which counts are to be made. This MUST be sorted in ascending order (lowest to highest).

w N×1 vector, containing weights.

OUTPUT

c P×1 vector containing the weighted counts of the elements of *x* that fall into the regions:

$$\begin{aligned} x &\leq v[1], \\ v[1] < x &\leq v[2], \\ &\cdots \\ v[p-1] < x &\leq v[p] \end{aligned}$$

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*.

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
```

create

PURPOSE Creates and opens a **GAUSS** data set for subsequent writing.

FORMAT **create** [[*vflag*]] [[**-w32**]] [[**complex**]] *fh* **=** *filename* **with** *vnames*,*col*,*dtyp*,*vtyp*;

create [[*vflag*]] [[**-w32**]] [[**complex**]] *fh* **=** *filename* **using** *comfile*;

INPUT *vflag* literal, version flag.

-v89	obsoleted, use **-v96**.
-v92	obsoleted, use **-v96**.
-v96	supported on all platforms.

For details on the various versions, see FILE I/O, Chapter 20. The default format can be specified in `gauss.cfg` by setting the

dat_fmt_version configuration variable. The default, **v96**, should be used.

filename literal or ^string

filename is the name to be given to the file on the disk. The name can include a path if the directory to be used is not the current directory. This file will automatically be given the extension `.dat`. If an extension is specified, the `.dat` will be overridden. If the name of the file is to be taken from a string variable, the name of the string must be preceded by the ^ (caret) operator.

create... with...

vnames literal or ^string or ^character matrix.

vnames controls the names to be given to the columns of the data file. If the names are to be taken from a string or character matrix, the ^ (caret) operator must be placed before the name of the string or character matrix. The number of columns parameter, *col*, also has an effect on the way the names will be created. See below and see the examples for details on the ways names are assigned to a data file.

col scalar expression.

col is a scalar expression containing the number of columns in the data file. If *col* is 0, the number of columns will be controlled by the contents of *vnames*. If *col* is positive, the file will contain *col* columns and the names to be given each column will be created as necessary depending on the *vnames* parameter. See the examples.

dtyp scalar expression.

dtyp is the precision used to store the data. This is a scalar expression containing 2, 4, or 8, which is the number of bytes per element.

2 signed integer
4 single precision
8 double precision

Data Type	Digits		Range	
integer	4	-32768	$\leq X \leq$	32767
single	6-7	8.43×10^{-37}	$\leq \lvert X \rvert \leq$	$3.37 \times 10^{+38}$
double	15-16	4.19×10^{-307}	$\leq \lvert X \rvert \leq$	$1.67 \times 10^{+308}$

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.

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.

OUTPUT *fh* scalar.

fh is the file handle which will be used by most commands to refer to the file within **GAUSS**. This file handle is actually a scalar containing an integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the **create** (or **open**) command is executed.

REMARKS If the **complex** flag is included, the new data set will be initialized to store complex number data. Complex data is stored a row at a time, with the real and imaginary halves interleaved, element by element.

The **-w32** flag is an optimization for Windows. It is ignored on all other platforms. **GAUSS** 7.0 uses Windows system file write commands that support 64-bit file sizes. These commands are slower on Windows XP than the 32-bit file write commands that were used in **GAUSS** 6.0. If you include the **-w32** flag, successive writes to the file indicated by *fh* will use 32-bit Windows write commands, which will be faster on Windows XP. Note, however, that the **-w32** flag does not support 64-bit file sizes.

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;
```

This example uses **create... with...** to create a double precision data file called `simdat.dat` on the default drive with 5 columns. The **writer** command is used to write 10000 rows of Normal random numbers into the file. The variables (columns) will be named: **AGE**, **SEX**, **EDUCAT**, **WAGE**, **OCC**.

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

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

The names in the this example will be: **AGE**, **PAY**, **SEX**, **JOB**.

AGE and **PAY** are numeric variables, **SEX** and **JOB** are character variables.

```
create fp = mydata with ^vnames,3,2;
```

The names will be: **AGE**, **PAY**, **SEX**.

```
create fp = mydata with ^vnames,8,2;
```

The names will now be: **AGE**, **PAY**, **SEX**, **JOB1**, **JOB2**, **JOB3**, **JOB4**, **JOB5**.

If a literal is used for the *vnames* parameter, the number of columns should be explicitly given in the *col* parameter and the names will be created as follows:

```
create fp = mydata with var,4,2;
```

Giving the names: **VAR1**, **VAR2**, **VAR3**, **VAR4**.

The next example assumes a command file called `comd.gcf` containing the following lines, created using a text editor:

```
outvar age, pay, sex;
outtyp i;
```

Then the following program could be used to write 100 rows of random integers into a file called `smpl.dat` in the subdirectory called `/gauss/data`:

```
filename = "/gauss/data/smpl";
create fh = ^filename using comd;
x = rndn(100,3)*10;
if writer(fh,x) /= rows(x);
    print "Disk Full"; end;
endif;
closeall fh;
```

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

SEE ALSO **datacreate**, **datacreatecomplex**, **open**, **readr**, **writer**, **eof**, **close**, **output**, **iscplxf**

crossprd

PURPOSE Computes the cross-products (vector products) of sets of 3×1 vectors.

FORMAT *z* = **crossprd(***x*,*y***);**

INPUT
x 3×K matrix, each column is treated as a 3×1 vector.

y 3×K matrix, each column is treated as a 3×1 vector.

OUTPUT
z 3×K 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 K×1 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

crout

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

FORMAT *y* **= crout(***x***);**

INPUT *x* N×N square nonsingular matrix.

OUTPUT *y* N×N matrix containing the lower (L) and upper (U) matrices of the Crout decomposition of *x*. The main diagonal of *y* is the main diagonal of the lower matrix L. The upper matrix has an implicit main diagonal of ones. Use **lowmat** and **upmat1** to extract the L and U matrices from *y*.

REMARKS Since it does not do row pivoting, it is intended primarily for teaching purposes. See **croutp** for a decomposition with pivoting.

EXAMPLE
```
X = { 1  2 -1,
      2  3 -2,
      1 -2  1 };

y = crout(x);
L = lowmat(y);
U = upmat1(y);
```

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

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

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

SEE ALSO **croutp**, **chol**, **lowmat**, **lowmat1**, **lu**, **upmat**, **upmat1**

croutp

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

FORMAT *y* = **croutp**(*x*);

INPUT *x* N×N square nonsingular matrix.

OUTPUT *y* (N+1)×N 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 */
```

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

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

$$\mathtt{r} = 4$$

$$\mathtt{indx} = \begin{matrix} 2 \\ 3 \\ 1 \end{matrix}$$

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

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

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

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

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

SEE ALSO **crout**, **chol**, **lowmat**, **lowmat1**, **lu**, **upmat**, **upmat1**

csrcol, csrlin

PURPOSE Returns the position of the cursor.

FORMAT *y* **= csrcol;**

y **= csrlin;**

OUTPUT *y* scalar, row or column value.

PORTABILITY **Windows** only

REMARKS *y* will contain the current column or row position of the cursor on the screen. The upper left corner is (1,1).

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

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

csrcol returns the cursor column with respect to the current output line, i.e., it

will return the same value whether the text is wrapped or not. **csrlin** returns the cursor line with respect to the top line in the window.

EXAMPLE

```
r = csrlin;
c = csrcol;
cls;
locate r,c;
```

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

SEE ALSO **cls**, **locate**

cumprodc

PURPOSE Computes the cumulative products of the columns of a matrix.

FORMAT *y* **= cumprodc(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* N×K matrix containing the cumulative products of the columns of *x*.

REMARKS This is based on the recursive series **recsercp**. **recsercp** could be called directly as follows:

```
recsercp(x,zeros(1,cols(x)))
```

to accomplish the same thing.

EXAMPLE

```
x = { 1  -3,
      2   2,
      3  -1 };
y = cumprodc(x);
```

```
          1.00   −3.00
y =       2.00   −6.00
          6.00    6.00
```

SOURCE cumprodc.src

SEE ALSO **cumsumc**, **recsercp**, **recserar**

cumsumc

PURPOSE Computes the cumulative sums of the columns of a matrix.

FORMAT *y* = **cumsumc**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* N×K matrix containing the cumulative sums of the columns of *x*.

REMARKS This is based on the recursive series function **recserar**. **recserar** could be called directly as follows:

```
recserar(x,x[1,.],ones(1,cols(x)))
```

to accomplish the same thing.

EXAMPLE

```
x = { 1  -3,
      2   2,
      3  -1 };

y = cumsumc(x);
```

$$y = \begin{matrix} 1 & -3 \\ 3 & -1 \\ 6 & -2 \end{matrix}$$

SOURCE cumsumc.src

SEE ALSO **cumprodc**, **recsercp**, **recserar**

curve

PURPOSE Computes a one-dimensional smoothing curve.

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

INPUT

x	K×1 vector, x-abscissae (x-axis values).
y	K×1 vector, y-ordinates (y-axis values).
d	K×1 vector or scalar, observation weights.
s	scalar, smoothing parameter. If *s* = 0, **curve** performs an interpolation. If *d* contains standard deviation estimates, a reasonable value for *s* is K.
sigma	scalar, tension factor.
G	scalar, grid size factor.

OUTPUT

u	(K*G)×1 vector, x-abscissae, regularly spaced.
v	(K*G)×1 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* N×1 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**

datacreate

PURPOSE Creates a **v96** real data set.

FORMAT *fh* = **datacreate**(*filename*,*vnames*,*col*,*dtyp*,*vtyp*);

INPUT

filename string, name of data file.

vnames string or N×1 string array, names of variables.

col scalar, number of variables.

dtyp scalar, data precision, one of the following:

2 2-byte, signed integer.
4 4-byte, single precision.
8 8-byte, double precision.

vtyp scalar or N×1 vector, types of variables, may contain one or both of the following:

0 character variable.
1 numeric variable.

OUTPUT *fh* scalar, file handle.

REMARKS The file handle returned by **datacreate** is a scalar containing a positive integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the **create**, **datacreate**, **datacreatecomplex**, **open** or **dataopen** commands are executed. The file handle is used to reference the file in the commands **readr** and **writer**. If **datacreate** fails, it returns a -1.

If *filename* does not include a path, then the file is placed on the current directory. The file is given a `.dat` extension if no extension is specified.

If *col* is set to 0, then the number of columns in the data set is controlled by the contents of *vnames*. If *col* is positive, then the file will contain *col* columns.

If *vnames* contains *col* elements, then each column is given the name contained in the corresponding row of *vnames*. If *col* is positive and *vnames* is a string, then the columns are given the names *vnames*1, *vnames*2, . . ., *vnames*N (or *vnames*01, *vnames*02, . . ., *vnames*N), where N = *col*. The numbers appended to *vnames* are padded on the left with zeros to the same length as N.

The *dtyp* argument allows you to specify the precision to use when storing your data. Keep in mind the following range restrictions when selecting a value for *dtyp*:

Data Type	Digits	Range
integer	4	$-32768 \leq X \leq 32767$
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 are rounded to the nearest integer as they are written to the data set. If the data to be written to the file contains character data, the precision must be 8 or the character information will be lost.

If *vtyp* is a scalar, then the value in *vtyp* controls the types of all of the columns in the data set. If it is an N×1 vector, then the type of each column is controlled by the value in the corresponding row of *vtyp*.

EXAMPLE

```
fh = datacreate("myfile.dat","V",100,8,1);
x = rndn(500,100);
r = writer(fh,x);
ret = close(fh);
```

This example creates a double precision data file called `myfile.dat`, which is placed in the current directory. The file contains 100 columns with 500 observations (rows), and the columns are given the names 'V001', 'V002', . . ., 'V100'.

SOURCE `datafile.src`

SEE ALSO **datacreatecomplex**, **create**, **dataopen**, **writer**

datacreatecomplex

PURPOSE Creates a **v96** complex data set.

FORMAT *fh* = **datacreatecomplex**(*filename*,*vnames*,*col*,*dtyp*,*vtyp*);

INPUT

filename string, name of data file.

vnames string or N×1 string array, names of variables.

col scalar, number of variables.

dtyp scalar, data precision, one of the following:

2 2-byte, signed integer.
4 4-byte, single precision.
8 8-byte, double precision.

vtyp scalar or N×1 vector, types of variables, may contain one or both of the following:

0 character variable.
1 numeric variable.

OUTPUT *fh* scalar, file handle.

REMARKS The file handle returned by **datacreatecomplex** is a scalar containing a positive integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the **create**, **datacreate**, **datacreatecomplex**, **open** or **dataopen** commands are executed. The file handle is used to reference the file in the commands **readr** and **writer**. If **datacreatecomplex** fails, it returns a -1.

Complex data is stored a row at a time, with the real and imaginary halves interleaved, element by element. For columns containing character data, the imaginary parts are zeroed out.

If *filename* does not include a path, then the file is placed on the current directory. The file is given a `.dat` extension if no extension is specified.

If *col* is set to 0, then the number of columns in the data set is controlled by the contents of *vnames*. If *col* is positive, then the file will contain *col* columns.

If *vnames* contains *col* elements, then each column is given the name contained in the corresponding row of *vnames*. If *col* is positive and *vnames* is a string, then the columns are given the names *vnames*1, *vnames*2, . . ., *vnames*N (or *vnames*01, *vnames*02, . . ., *vnames*N), where N = *col*. The numbers appended to *vnames* are padded on the left with zeros to the same length as N.

The *dtyp* argument allows you to specify the precision to use when storing your data. Keep in mind the following range restrictions when selecting a value for *dtyp*:

Data Type	Digits	Range
integer	4	$-32768 \leq X \leq 32767$
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 are rounded to the nearest integer as they are written to the data set. If the data to be written to the file contains character data, the precision must be 8 or the character information will be lost.

If *vtyp* is a scalar, then the value in *vtyp* controls the types of all of the columns in the data set. If it is an N×1 vector, then the type of each column is controlled by the value in the corresponding row of *vtyp*.

EXAMPLE

```
string vnames = { "random1", "random2" };
fh = datacreatecomplex("myfilecplx.dat",vnames,2,8,1);
x = complex(rndn(1000,2),rndn(1000,2));
r = writer(fh,x);
ret = close(fh);
```

This example creates a complex double precision data file called `myfilecplx.dat`, which is placed in the current directory. The file contains 2 columns with 1000 observations (rows), and the columns are given the names ‘random1’ and ‘random2’.

SOURCE datafile.src

SEE ALSO **datacreate**, **create**, **dataopen**, **writer**

datalist

PURPOSE List selected variables from a data set.

FORMAT **datalist** *dataset* [[*var1* [[*var2*...]]];

INPUT
dataset literal, name of the data set.
var# literal, the names of the variables to list.

GLOBAL INPUT
__range scalar, the range of rows to list. The default is all rows.

__miss scalar, controls handling of missing values.

0 display rows with missing values.
1 do not display rows with missing values.

The default is 0.

__prec scalar, the number of digits to the right of the decimal point to display. The default is 3.

REMARKS The variables are listed in an interactive mode. As many rows and columns as will fit on the screen are displayed. You can use the cursor keys to pan and scroll around in the listing.

EXAMPLE
```
datalist freq age sex pay;
```

This command will display the variables **age**, **sex**, and **pay** from the data set freq.dat.

SOURCE datalist.src

dataload

PURPOSE Loads matrices, N-dimensional arrays, strings and string arrays from a disk file.

FORMAT *y* **= dataload(***filename***);**

INPUT *filename* string, name of data file.

OUTPUT *y* matrix, array, string or string array, data retrieved from the file.

REMARKS The proper extension must be included in the file name. Valid extensions are as follows:

`.fmt`	matrix file
	array file
`.fst`	string file
	string array file

See FILE I/O, Chapter 20, for details on these file types.

EXAMPLE
```
y = dataload("myfile.fmt");
```

SEE ALSO **load**, **datasave**

dataloop (dataloop)

PURPOSE Specifies the beginning of a data loop.

FORMAT **dataloop** *infile outfile***;**

INPUT *infile* string variable or literal, the name of the source data set.

OUTPUT *outfile* string variable or literal, the name of the output data set.

REMARKS The statements between the **dataloop**... **endata** commands are assumed to be metacode to be translated at compile time. The data from *infile* is manipulated by the specified statements, and stored to the data set *outfile*. Case is not significant within the **dataloop**... **endata** section, except for within quoted strings. Comments can be used as in any **GAUSS** code.

EXAMPLE
```
src = "source";
dataloop ^src dest;
    make newvar = x1 + x2 + log(x3);
    x6 = sqrt(x4);
    keep x6, x5, newvar;
endata;
```

Here, **src** is a string variable requiring the caret (^) operator, while **dest** is a string literal.

dataopen

PURPOSE Opens a data set.

FORMAT *fh* = **dataopen(***filename***,***mode***);**

INPUT *filename* string, name of data file.

mode string containing one of the following:

read	open file for read.
append	open file for append.
update	open file for update.

OUTPUT *fh* scalar, file handle.

REMARKS The file must exist before it can be opened with the **dataopen** command (to create a new file, see **datacreate** or **datasave**).

The file handle returned by **dataopen** is a scalar containing a positive integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the **create**, **datacreate**, **datacreatecomplex**, **open** or **dataopen** commands are executed. The file handle is used to reference the file in the commands **readr** and **writer**. If **dataopen** fails, it returns a -1.

A file can be opened simultaneously under more than one handle. If the value that is in the file handle when the **dataopen** command begins to execute matches that of an already open file, the process will be aborted and a **File already open** error message will be given. This gives you some protection against opening a second file with the same handle as a currently open file. If this happens, you would no longer be able to access the first file.

It is important to set unused file handles to zero because both **dataopen** and **datacreate** check the value that is in a file handle to see if it matches that of an open file before they proceed with the process of opening a file. You may set unused file handles to zero with the **close** or **closeall** commands.

If *filename* does not have an extension, **dataopen** appends a `.dat` extension before searching for the file. If the file is an `.fmt` matrix file, the extension must be explicitly given. If no path information is included, then **dataopen** searches for the file in the current directory.

Files opened in **read** mode cannot be written to. The pointer is set to the beginning of the file and the **writer** function is disabled for files opened in this way. This is the only mode available for matrix files (`.fmt`), which are always written in one piece with the **save** command.

Files opened in **append** mode cannot be read. The pointer is set to the end of the file so that a subsequent write to the file with the **writer** function will add data to the end of the file without overwriting any of the existing data in the file. The **readr** function is disabled for files opened in this way. This mode is used to add additional rows to the end of a file.

Files opened in **update** mode can be read from and written to. The pointer is

set to the beginning of the file. This mode is used to make changes in a file.

EXAMPLE

```
fh = dataopen("myfile.dat","read");
y = readr(fh,100);
ret = close(fh);
```

This example opens the data file `myfile.dat` in the current directory and reads 100 observations (rows) from the file into the global variable **y**.

SOURCE `datafile.src`

SEE ALSO **open**, **datacreate**, **writer**, **readr**

datasave

PURPOSE Saves matrices, N-dimensional arrays, strings and string arrays to a disk file.

FORMAT *ret* **= datasave(***filename***,***x***);**

INPUT

filename	string, name of data file.
x	matrix, array, string or string array, data to write to disk.

OUTPUT

ret	scalar, return code, 0 if successful, or -1 if it is unable to write the file.

REMARKS **datasave** can be used to save matrices, N-dimensional arrays, strings and string arrays. The following extensions are given to files that are saved with **datasave**:

matrix	`.fmt`
array	`.fmt`
string	`.fst`
string array	`.fst`

See FILE I/O, Chapter 20, for details on these file types.

Use **dataload** to load a data file created with **datasave**.

EXAMPLE
```
x = rndn(1000,100);
ret = datasave("myfile.fmt",x);
```

SEE ALSO **save**, **dataload**

date

PURPOSE Returns the current date in a 4-element column vector, in the order: year, month, day, and hundredths of a second since midnight.

FORMAT *y* **= date;**

REMARKS The hundredths of a second since midnight can be accessed using **hsec**.

EXAMPLE
```
print date;

    2005.0000
    8.0000000
    31.000000
    4804392.2
```

SEE ALSO **time**, **timestr**, **ethsec**, **hsec**, **etstr**

datestr

PURPOSE Returns a date in a string.

FORMAT *str* = **datestr**(*d*);

INPUT *d* 4×1 vector, like the **date** function returns. If this is 0, the **date** function will be called for the current system date.

OUTPUT *str* 8 character string containing current date in the form: **mo/dy/yr**

EXAMPLE
```
d = { 2005, 8, 31, 0 };
y = datestr(d);
print y;

    8/31/05
```

SOURCE time.src

SEE ALSO **date**, **datestring**, **datestrymd**, **time**, **timestr**, **ethsec**

datestring

PURPOSE Returns a date in a string with a 4-digit year.

FORMAT *str* = **datestring**(*d*);

INPUT *d* 4×1 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;

    8/31/2005
```

SOURCE time.src

SEE ALSO **date**, **datestr**, **datestrymd**, **time**, **timestr**, **ethsec**

datestrymd

PURPOSE Returns a date in a string.

FORMAT *str* = **datestrymd**(*d*);

INPUT *d* 4×1 vector, like the **date** function returns. If this is 0, the **date** function will be called for the current system date.

OUTPUT *str* 8 character string containing current date in the form: **yyyymmdd**

EXAMPLE
```
d = { 2005, 8, 31, 0 };
y = datestrymd(d);
print y;

    20050831
```

SOURCE time.src

SEE ALSO **date**, **datestr**, **datestring**, **time**, **timestr**, **ethsec**

dayinyr

PURPOSE Returns day number in the year of a given date.

FORMAT *daynum* = **dayinyr**(*dt*);

INPUT *dt* 3×1 or 4×1 vector, date to check. The date should be in the form returned by **date**.

OUTPUT *daynum* scalar, the day number of that date in that year.

EXAMPLE
```
x = { 2005, 8, 31, 0 };
y = dayinyr(x);
print y;

    243.00000
```

SOURCE time.src

GLOBALS **_isleap**

dayofweek

PURPOSE Returns day of week.

FORMAT *d* **= dayofweek(***a***);**

INPUT *a* N×1 vector, dates in DT format.

OUTPUT *d* N×1 vector, integers indicating day of week of each date:

1 Sunday
2 Monday
3 Tuesday
4 Wednesday
5 Thursday
6 Friday
7 Saturday

REMARKS The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number

20050901183207

represents 18:32:07 or 6:32:07 PM on September 1, 2005.

SOURCE `time.src`

debug

PURPOSE Runs a program under the source level debugger.

FORMAT **debug** *filename*;

INPUT *filename* Literal, name of file to debug.

REMARKS See DEBUGGING, Section 3.3.

declare

PURPOSE Initializes global variables at compile time.

FORMAT **declare** [[*type*]] *symbol* [[*aop clist*]];

INPUT *type* optional literal, specifying the type of the symbol.

matrix
string
array
sparse matrix
struct *structure_type*

if *type* is not specified, **matrix** is assumed. Set *type* to **string** to initialize a string or string array variable.

symbol the name of the symbol being declared.

aop the type of assignment to be made.

- **=** if not initialized, initialize. If already initialized, reinitialize.
- **!=** if not initialized, initialize. If already initialized, reinitialize.
- **:=** if not initialized, initialize. If already initialized, redefinition error.
- **?=** if not initialized, initialize. If already initialized, leave as is.

If *aop* is specified, *clist* must be also.

clist a list of constants to assign to *symbol*.

If *aop clist* is not specified, *symbol* is initialized as a scalar 0 or a null string.

REMARKS The **declare** syntax is similar to the **let** statement.

declare generates no executable code. This is strictly for compile time initialization. The data on the right-hand side of the equal sign must be constants. No expressions or variables are allowed.

declare statements are intended for initialization of global variables that are used by procedures in a library system.

It is best to place **declare** statements in a separate file from procedure definitions. This will prevent redefinition errors when rerunning the same program without clearing your workspace.

The optional *aop* and *clist* arguments are allowed only for declaring matrices, strings, and string arrays. When you **declare** an N-dimensional array, sparse matrix, or structure, they will be initialized as follows:

Variable Type	Initializes To
N-dimensional array	1-dimensional array of 1 containing 0
sparse matrix	empty sparse matrix
structure	structure containing empty and/or zeroed out members

Complex numbers can be entered by joining the real and imaginary parts with a sign (+ or -); there should be no spaces between the numbers and the sign. Numbers with no real part can be entered by appending an 'i' to the number.

There should be only one declaration for any symbol in a program. Multiple declarations of the same symbol should be considered a programming error. When **GAUSS** is looking through the library to reconcile a reference to a matrix or a string, it will quit looking as soon as a symbol with the correct name is found. If another symbol with the same name existed in another file, it would never be found. Only the first one in the search path would be available to programs.

Here are some of the possible uses of the three forms of declaration:

!=, =

Interactive programming or any situation where a global by the same name will probably be sitting in the symbol table when the file containing the **declare** statement is compiled. The symbol will be reset.

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 because you have probably just outsmarted yourself. This will keep you out of trouble because it won't allow you to zap one symbol with another value that you didn't know was getting mixed up in your program. You probably need to rename one of them.

You need to 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 don't want them reset for each successive run even if the file containing the **declare**'s gets

recompiled. This can get you into trouble if you are not careful.

The **declare** statement warning level is a compile option. Call **config** in the command line version of **GAUSS** or select Preferences from the Configure menu in the Windows interface to edit this option. If **declare** warnings are on, you will be warned whenever a **declare** statement encounters a symbol that is already initialized. Here's what happens when you declare a symbol that is already initialized when **declare** warnings are turned on:

declare !=	Reinitialize and warn.
declare :=	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;
```

```
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 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;
```

```
      DOG
x =
      CAT
```

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

```
      dog
x =
      cat
```

```
declare array a;
```

a is a 1-dimensional array of 1 containing 0.

```
declare sparse matrix sm;
```

sm is an empty sparse matrix.

```
struct mystruct {
    matrix m;
    string s;
```

```
    string array sa;
    array a;
    sparse matrix sm;
};

declare struct mystruct ms;
```

ms is a **mystruct** structure, with its members set as follows:

ms.m	empty matrix
ms.s	null string
ms.sa	1×1 string array containing a null string
ms.a	1-dimensional array of 1 containing 0
ms.sm	empty sparse matrix

SEE ALSO **let**, **external**

delete

PURPOSE Deletes global symbols from the symbol table.

FORMAT **delete** [[-*flags*]] [[*symbol1*]] [[*symbol2*]] [[*symbol3*]]**;**

INPUT *flags* specify the type(s) of symbols to be deleted

p	procedures
k	keywords
f	**fn** functions
m	matrices
s	strings
g	only procedures with global references
l	only procedures with all local references
n	no pause for confirmation

symbol literal, name of symbol to be deleted. If symbol ends in an asterisk, all symbols matching the leading characters will be deleted.

REMARKS This completely and irrevocably deletes a symbol from **GAUSS**'s memory and workspace.

Flags must be preceded by a dash (e.g. **-pfk**). If the **n**(no pause) flag is used, you will not be asked for confirmation for each symbol.

This command is supported only from interactive level. Since the interpreter executes a compiled pseudo-code, this command would invalidate a previously compiled code image and therefore would destroy any program it was a part of. If any symbols are deleted, all procedures, keywords and functions with global references to those symbols will be deleted as well.

EXAMPLE
```
print x;
```

```
96.000000
6.0000000
14.000000
3502965.9
```

```
delete -m x;
```

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

```
show x;
```

x no longer exists.

delete (dataloop)

PURPOSE Removes specific rows in a data loop based on a logical expression.

FORMAT **delete** *logical expression*;

REMARKS Deletes only those rows for which *logical expression* is TRUE. Any variables referenced must already exist, either as elements of the source data set, as **extern**'s, or as the result of a previous **make**, **vector**, or **code** statement.

GAUSS expects *logical expression* to return a row vector of 1's and 0's. The relational and other operators (e.g. <) are already interpreted in terms of their dot equivalents (. <), but it is up to the user to make sure that function calls within *logical expression* result in a vector.

EXAMPLE
```
delete age < 40 or sex =\,= 'FEMALE';
```

SEE ALSO **select**

DeleteFile

PURPOSE Deletes files.

FORMAT *ret* = **DeleteFile**(*name*);

INPUT *name* string or N×K string array, name of file or files to delete.

OUTPUT *ret* scalar or N×K matrix, 0 if successful.

REMARKS The return value, *ret*, is scalar if *name* is a string. If *name* is an N×K string array, *ret* will be an N×K matrix reflecting the success or failure of each separate file deletion.

DeleteFile calls the C library **unlink** function for each file. If **unlink** fails it sets the C library errno value. **DeleteFile** returns the value of errno if **unlink**

fails, otherwise it returns zero. If you want detailed information about the reason for failure, consult the C library **unlink** documentation for your platform for details.

delif

PURPOSE Deletes rows from a matrix. The rows deleted are those for which there is a 1 in the corresponding row of *e*.

FORMAT *y* **= delif(***x*,*e***);**

INPUT

x	N×K data matrix.
e	N×1 logical vector (vector of 0's and 1's).

OUTPUT

y	M×K 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);
```

```
y =  0   10   20
```

All rows for which the elements in column 1 are greater than 0 and the elements in column 3 are less than 100 are deleted.

SEE ALSO **selif**

denseToSp

PURPOSE Converts a dense matrix to a sparse matrix.

FORMAT *y* = **denseToSp**(*x*,*eps*);

INPUT

x	M×N dense matrix.
eps	scalar, elements of *x* whose absolute values are less than or equal to *eps* will be treated as zero.

OUTPUT

y	M×N sparse matrix.

REMARKS A dense matrix is just a normal format matrix.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **denseToSp**.

EXAMPLE

```
sparse matrix y;
x = { 0  0  0  1,
      0  4  0  0,
      0  0  0  0,
      0  0 -2  0 };

y = denseToSp(x,0);
d = spDenseSubmat(y,0,0);
```

$$d = \begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 \end{matrix}$$

SEE ALSO **spCreate**, **spDenseSubmat**, **spToDense**

denseToSpRE

PURPOSE Converts a dense matrix to a sparse matrix, using a relative epsilon.

FORMAT *y* **= denseToSpRE(***x*,*reps*);

INPUT

x	M×N dense matrix.
reps	scalar, relative epsilon. Elements of *x* will be treated as zero if their absolute values are less than or equal to *reps* multiplied by the mean of the absolute values of the non-zero values in *x*.

OUTPUT

y	M×N sparse matrix.

REMARKS A dense matrix is just a normal format matrix.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **denseToSpRE**.

EXAMPLE

```
sparse matrix y;
x = { -9  0  0  1,
       0  4  0  0,
       5  0  0  7,
       0  0 -2  0 };
```

```
y = denseToSpRE(x,.5);
d = spToDense(y);
```

$$d = \begin{matrix} -9 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 5 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 \end{matrix}$$

SEE ALSO **denseToSp**, **spCreate**, **spToDense**

denToZero

PURPOSE Converts every denormal to a 0 in a matrix or array.

FORMAT *y* **= denToZero(***x***);**

INPUT *x* A matrix or an N-dimensional array.

OUTPUT *y* A matrix or an N-dimensional array with the same orders as the input. Every denormal in the input will be converted to 0 in the output. column.

EXAMPLE
```
x = { 1, exp(-724.5), 3 };
y = isden(x);
```

y = 1, 0, 3;

SEE ALSO **isden**

design

PURPOSE Creates a design matrix of 0's and 1's from a column vector of numbers specifying the columns in which the 1's should be placed.

FORMAT *y* **= design(***x***);**

INPUT *x* N×1 vector.

OUTPUT *y* N×K matrix, where K = **maxc(***x***)**; each row of *y* will contain a single 1, and the rest 0's. The one in the i^{th} row will be in the **round(***x***[***i***,1])** column.

REMARKS Note that *x* does not have to contain integers: it will be rounded to nearest if necessary.

EXAMPLE
```
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
```

SOURCE design.src

SEE ALSO **cumprodc**, **cumsumc**, **recserrc**

det

PURPOSE Returns the determinant of a square matrix.

FORMAT *y* = **det**(*x*);

INPUT *x* N×N square matrix or K-dimensional array where the last two dimensions are N×N .

OUTPUT *y* scalar or [K-2]-dimensional array, the determinant(s) of *x*.

REMARKS *x* may be any valid expression that returns a square matrix (number of rows equals number of columns) or a K-dimensional array where the last two dimensions are of equal size.

If *x* is a K-dimensional array, the result will be a [K-2]-dimensional array containing the determinants of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10×4×4 array, the result will be a 1-dimensional array of 10 elements containing the determinants of each of the 10 4×4 arrays contained in *x*.

det computes a LU decomposition.

detl can be much faster in many applications.

EXAMPLE
```
x = { 3  2  1,
      0  1 -2,
      1  3  4 };
y = det(x);
```

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

```
y = 25
```

SEE ALSO **detl**

detl

PURPOSE Returns the determinant of the last matrix that was passed to one of the intrinsic matrix decomposition routines.

FORMAT *y* **= detl;**

REMARKS Whenever one of the intrinsic matrix decomposition routines is executed, the determinant of the matrix is also computed and stored in a system variable. This function will return the value of that determinant and, because the value has been computed in a previous instruction, this will require no computation.

The following functions will set the system variable used by **detl**:

chol(*x***)**	
crout(*x***)**	
croutp(*x***)**	
det(*x***)**	
inv(*x***)**	
invpd(*x***)**	
solpd(*y***,***x***)**	determinant of *x*
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 10×10:

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

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

SEE ALSO **det**

dfft

PURPOSE Computes a discrete Fourier transform.

FORMAT *y* **= dfft(***x***);**

INPUT *x* N×1 vector.

OUTPUT *y* N×1 vector.

REMARKS The transform is divided by N.

This uses a second-order Goertzel algorithm. It is considerably slower than **fft**, but it may have some advantages in some circumstances. For one thing, N does not have to be an even power of 2.

SOURCE dfft.src

SEE ALSO **dffti**, **fft**, **ffti**

dffti

PURPOSE Computes inverse discrete Fourier transform.

FORMAT *y* = **dffti**(*x*);

INPUT *x* N×1 vector.

OUTPUT *y* N×1 vector.

REMARKS The transform is divided by N.

This uses a second-order Goertzel algorithm. It is considerably slower than **ffti**, but it may have some advantages in some circumstances. For one thing, N does not have to be an even power of 2.

SOURCE dffti.src

SEE ALSO **fft**, **dffti**, **ffti**

diag

PURPOSE Creates a column vector from the diagonal of a matrix.

FORMAT *y* = **diag**(*x*);

INPUT *x* N×K matrix or L-dimensional array where the last two dimensions are N×K.

OUTPUT *y* min(N,K)×1 vector or L-dimensional array where the last two dimensions are min(N,K)×1.

REMARKS If *x* is a matrix, it need not be square. Otherwise, if *x* is an array, the last two dimensions need not be equal.

If *x* is an array, the result will be an array containing the diagonals of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10×4×4 array, the result will be a 10×4×1 array containing the diagonals of each of the 10 4×4 arrays contained in *x*.

diagrv reverses the procedure and puts a vector into the diagonal of a matrix.

EXAMPLE
```
x = rndu(3,3);
y = diag(x);
```

```
          0.660818   0.367424   0.302208
x =       0.204800   0.077357   0.145755
          0.712284   0.353760   0.642567
```

```
          0.660818
y =       0.077357
          0.642567
```

SEE ALSO **diagrv**

diagrv

PURPOSE Inserts a vector into the diagonal of a matrix.

FORMAT *y* **= diagrv(***x*,*v***);**

INPUT
x N×K matrix.
v min(N,K)×1 vector.

OUTPUT
y N×K matrix equal to *x* with its principal diagonal elements equal to those of *v*.

REMARKS **diag** reverses the procedure and pulls the diagonal out of a matrix.

EXAMPLE
```
x = rndu(3,3);
v = ones(3,1);
y = diagrv(x,v);
```

```
      0.660818  0.367424  0.302208
x =   0.204800  0.077357  0.145755
      0.712284  0.353760  0.642567

      1.000000
v =   1.000000
      1.000000

      1.000000  0.367424  0.302208
y =   0.204800  1.000000  0.145755
      0.712284  0.353760  1.000000
```

SEE ALSO **diag**

digamma

PURPOSE Computes the digamma function.

FORMAT *y* = **digamma**(*x*);

INPUT *x* M×N matrix or N-dimensional array.

OUTPUT *y* M×N matrix or N-dimensional array, digamma.

REMARKS The digamma function is the first derivative of the log of the gamma function with respect to its argument.

dlibrary

PURPOSE Dynamically links and unlinks shared libraries.

FORMAT **dlibrary** *lib1* [[*lib2*]]**...;**

dlibrary -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 `.so` or `.dll`, depending on the platform. They are searched for in the default dynamic library directory. Arguments that include "/" path separators are assumed to be file names, and are not expanded. Relatively pathed file names are assumed to be specified relative to the current working directory, not relative to the dynamic library directory.

-a append flag, the shared libraries listed are added to the current set of shared libraries rather than replacing them. For search purposes, the new shared libraries follow the already active ones. Without the **-a** flag, any previously linked libraries are dumped.

-d dump flag, ALL shared libraries are unlinked and the functions they contain are no longer available to your programs. If you use **dllcall** to call one of your functions after executing a `dlibrary -d`, your program will terminate with an error.

REMARKS If no flags are used, the shared libraries listed are linked into **GAUSS** and any previously linked libraries are dumped. When you call **dllcall**, the shared libraries will be searched in the order listed for the specified function. The first instance of the function found will be called.

dlibrary with no arguments prints out a list of the currently linked shared libraries. The order in which they are listed is the order in which they are searched for functions.

dlibrary recognizes a default directory in which to look for dynamic libraries. You can specify this by setting the variable **dlib_path** in `gauss.cfg`. Set it to point to a single directory, not a sequence of directories. **sysstate**, case 24, may also be used to get and set this default.

GAUSS maintains its own shared libraries which are listed when you execute **dlibrary** with no arguments, and searched when you call **dllcall**. The default shared library or libraries are searched last. You can force them to be searched earlier by listing them explicitly in a **dlibrary** statement. They always active and are not unlinked when you execute **dlibrary -d**.

For more information, see FOREIGN LANGUAGE INTERFACE, Chapter 21.

SEE ALSO **dllcall**, **sysstate**–case 24

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 shared libraries into **GAUSS**; **dllcall** is used to access the functions contained in those shared libraries. **dllcall** searches the shared libraries (see **dlibrary** for an explanation of the search order) for a function named *func*, and calls the first instance it finds. The default shared libraries are searched last.

INPUT

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

arg# arguments to be passed to *func*, optional. These must be simple variable references; they cannot be expressions.

-r optional flag. If **-r** is specified, **dllcall** examines the value returned by *func*, and fails if it is nonzero.

-v optional flag. Normally, **dllcall** passes parameters to *func* in a list. If **-v** is specified, **dllcall** passes them in a vector. See below for more details.

REMARKS *func* should be written to:

1. Take 0 or more pointers to doubles as arguments.
2. Take arguments either in a list or a vector.
3. Return an integer.

In C syntax, *func* should take one of the following forms:

1. **int** *func*(**void**);
2. **int** *func*(**double ***_arg1_ [,**double ***_arg2_,...]);
3. **int** *func*(**double ***_arg_[]);

dllcall can pass a list of up to 100 arguments to *func*; if it requires more arguments than that, you MUST write it to take a vector of arguments, and you MUST specify the **-v** flag when calling it. **dllcall** can pass up to 1000 arguments in vector format. In addition, in vector format **dllcall** appends a null pointer to the vector, so you can write *func* to take a variable number of arguments and just test for the null pointer.

Arguments are passed to *func* by reference. This means you can send back more than just the return value, which is usually just a success/failure code. (It also

means that you need to be careful not to overwrite the contents of matrices or strings you want to preserve.) To return data from *func*, simply set up one or more of its arguments as return matrices (basically, by making them the size of what you intend to return), and inside *func* assign the results to them before returning.

For more information, see FOREIGN LANGUAGE INTERFACE, Chapter 21.

SEE ALSO **dlibrary**, **sysstate**–case 24

do while, do until

PURPOSE Executes a series of statements in a loop as long as a given expression is true (or false).

FORMAT
```
do while expression;
or
do until expression;
        .
        .
        .
    statements in loop
        .
        .
        .
endo;
```

REMARKS *expression* is any expression that returns a scalar. It is TRUE if it is nonzero and FALSE if it is zero.

In a **do while** loop, execution of the loop will continue as long as the expression is TRUE.

In a **do until** loop, execution of the loop will continue as long as the

expression is FALSE.

The condition is checked at the top of the loop. If execution can continue, the statements of the loop are executed until the **endo** is encountered. Then **GAUSS** returns to the top of the loop and checks the condition again.

The **do** loop does not automatically increment a counter. See the first example below.

do loops may be nested.

It is often possible to avoid using loops in **GAUSS** by using the appropriate matrix operator or function. It is almost always preferable to avoid loops when possible, since the corresponding matrix operations can be much faster.

EXAMPLE

```
format /rdn 1,0;
space = "     ";
comma = ",";
i = 1;
do while i <= 4;
    j = 1;
    do while j <= 3;
        print space i comma j;;
        j = j+1;
    endo;
    i = i+1;
    print;
endo;
```

```
 1,1   1,2   1,3
 2,1   2,2   2,3
 3,1   3,2   3,3
 4,1   4,2   4,3
```

In the example above, two nested loops are executed and the loop counter values are printed out. Note that the inner loop counter must be reset inside of

the outer loop before entering the inner loop. An empty **print** statement is used to print a carriage return/line feed sequence after the inner loop finishes.

The following are examples of simple loops that execute a predetermined number of times. These loops will both have the result shown.

First loop:

```
format /rd 1,0;
i = 1;
do while i <= 10;
    print i;;
    i = i+1;
endo;
```

produces:

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

Second loop:

```
format /rd 1,0;
i = 1;
do until i > 10;
    print i;;
    i = i+1;
endo;
```

produces:

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

SEE ALSO **continue**, **break**

dos

PURPOSE Provides access to the operating system from within **GAUSS**.

FORMAT **dos** [[*s*]]**;**

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

PORTABILITY **UNIX/Linux**

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

This function may be used in terminal mode.

Windows

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

Running programs in the background is allowed on both of the aforementioned platforms.

REMARKS This allows all operating system commands to be used from within **GAUSS**. It allows other programs to be run even though **GAUSS** is still resident in memory.

If no operating system command (for instance, **dir** or **copy**) or program name is specified, then a shell of the operating system will be entered which can be used just like the base level OS. The **exit** command must be given from the shell to get back into **GAUSS**. If a command or program name is included, the return to **GAUSS** is automatic after the OS command has been executed.

All matrices are retained in memory when the OS is accessed in this way. This command allows the use of word processing, communications, and other programs from within **GAUSS**.

Do not execute programs that terminate and remain resident because they will be left resident inside of **GAUSS**'s workspace. Some examples are programs that create RAM disks or print spoolers.

If the command is to be taken from a string variable, the ^ (caret) must precede the string.

The shorthand "**>**" can be used in place of "**dos**".

EXAMPLE

```
cmdstr = "atog mycfile";
dos ^cmdstr;
```

This will run the ATOG utility, using `mycfile.cmd` as the ATOG command file. For more information, see ATOG, Chapter 26.

```
> dir *.prg;
```

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

```
> ls *.prg
```

This will perform the same operation on UNIX/Linux.

```
dos;
```

This will cause a second level OS shell to be entered. The OS prompt will appear and OS commands or other programs can be executed. To return to **GAUSS**, type **exit**.

SEE ALSO **exec**

doswin

PURPOSE Opens the DOS compatibility window with default settings.

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

```
let attr = 50 50 7 0 7;

if not DOSWinOpen("Legacy Window", attr);
```

```
    errorlog "Failed to open DOS window, aborting";
    stop;
endif;
 .
 .
 .
DOSWinCloseall;
```


DOSWinOpen

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

FORMAT *ret* = **DOSWinOpen**(*title*,*attr*);

INPUT

title string, window title.

attr 5×1 vector or scalar missing, window attributes.

[1] window x position

[2] window y position

[3] text foreground color

[4] text background color

[5] close action bit flags

bit 0 (1's bit)	issue dialog
bit 1 (2's bit)	close window
bit 2 (4's bit)	stop program

OUTPUT *ret* scalar, success flag, 1 if successful, 0 if not.

PORTABILITY **Windows** only

REMARKS If *title* is a null string (""), the window will be titled "GAUSS-DOS".

Defaults are defined for the elements of *attr*. To use the default, set an element to a missing value. Set *attr* to a scalar missing to use all defaults. The defaults are defined as follows:

[1]	varies	use x position of previous DOS window
[2]	varies	use y position of previous DOS window
[3]	7	white foreground
[4]	0	black background
[5]	6	4+2: stop program and close window without confirming

If the DOS window is already open, the new *title* and *attr* will be applied to it. Elements of *attr* that are missing are not reset to the default values, but are left as is.

To set the close action flags value (*attr*[5]), just sum the desired bit values. For example:

stop program (4) + close window (2) + confirm close (1) = 7

The close action flags are only relevant when a user attempts to interactively close the DOS window while a program is running. If **GAUSS** is idle, the window will be closed immediately. Likewise, if a program calls **DOSWinCloseall**, the window is closed, but the program does not get terminated.

EXAMPLE

```
let attr = 50 50 7 0 7;

if not DOSWinOpen("Legacy Window", attr);
    errorlog "Failed to open DOS window, aborting";
    stop;
endif;
```

This example opens the DOS window at screen location (50,50), with white text on a black background. The close action flags are 4 + 2 + 1 (stop program + close window + issue confirm dialog) = 7. Thus, if the user attempts to close the window while a program is running, he/she will be asked for confirmation. Upon confirmation, the window will be closed and the program terminated.

dotfeq, dotfge, dotfgt, dotfle, dotflt, dotfne

PURPOSE Fuzzy comparison functions. These functions use **_fcmptol** to fuzz the comparison operations to allow for roundoff error.

FORMAT *y* = **dotfeq**(*a*,*b*);

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

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

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

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

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

INPUT

a	N×K matrix, first matrix.
b	L×M matrix, second matrix, E×E compatible with *a*.

GLOBAL INPUT

_fcmptol	scalar, comparison tolerance. The default value is 1.0e-15.

OUTPUT

y	max(N,L) by max(K,M) matrix of 1's and 0's.

REMARKS The return value is 1 if TRUE and 0 if FALSE.

The statement:

```
y = dotfeq(a,b);
```

is equivalent to:

```
y = a .eq b;
```

The calling program can reset **_fcmptol** before calling these procedures:

```
_fcmptol = 1e-12;
```

EXAMPLE

```
x = 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 =
1.0000000 1.0000000
0.0000000 0.0000000

SOURCE fcompare.src

GLOBALS **_fcmptol**

SEE ALSO **feq--fne**

dotfeqmt, dotfgemt, dotfgtmt, dotflemt, dotfltmt, dotfnemt

PURPOSE Fuzzy comparison functions. These functions use the *fcmptol* argument to fuzz the comparison operations to allow for roundoff error.

FORMAT *y* **= dotfeqmt(***a*,*b*,*fcmptol***);**

y = **dotfgemt**(*a*,*b*,*fcmptol*);

y = **dotfgtmt**(*a*,*b*,*fcmptol*);

y = **dotflemt**(*a*,*b*,*fcmptol*);

y = **dotfltmt**(*a*,*b*,*fcmptol*);

y = **dotfnemt**(*a*,*b*,*fcmptol*);

INPUT

a	N×K matrix, first matrix.
b	L×M matrix, second matrix, E×E compatible with *a*.
fcmptol	scalar, comparison tolerance.

OUTPUT

y	max(N,L) by max(K,M) matrix of 1's and 0's.

REMARKS The return value is 1 if TRUE and 0 if FALSE.

The statement:

```
y = dotfeqmt(a,b,1e-13);
```

is equivalent to:

```
y = a .eq b;
```

EXAMPLE

```
x = rndu(2,2);
y = rndu(2,2);
t = dotfge(x,y,1e-15);
```

```
x =  0.85115559  0.98914218
     0.12703276  0.43365175
```

```
y =  0.41907226  0.49648058
     0.58039125  0.98200340

t =  1.0000000  1.0000000
     0.0000000  0.0000000
```

SOURCE fcomparemt.src

SEE ALSO **feqmt--fnemt**

draw

PURPOSE Graphs lines, symbols, and text using the PQG global variables. This procedure does not require actual X, Y, or Z data since its main purpose is to manually build graphs using **_pline, _pmsgctl**, **_psym**, **_paxes**, **_parrow** and other globals.

LIBRARY pgraph

FORMAT **draw;**

REMARKS **draw** is especially useful when used in conjunction with transparent windows.

EXAMPLE
```
library pgraph;
graphset;

begwind;
makewind(9,6.855,0,0,0);  /* make full size window for plot */
makewind(3,1,3,3,0);      /* make small overlapping window
                          ** for text
                          */
setwind(1);
```

```
    x = seqa(.1,.1,100);
    y = sin(x);
    xy(x,y);                /* plot data in first window */

nextwind;
    _pbox = 15;
    _paxes = 0;
    _pnum = 0;
    _ptitlht = 1;
    margin(0,0,2,0);
    title("This is a text window.");
    draw;                   /* add a smaller text window */

endwind;                    /* create graph */
```

SOURCE pdraw.src

SEE ALSO **window**, **makewind**

PURPOSE Specifies columns to be dropped from the output data set in a data loop.

FORMAT **drop** *variable_list*;

REMARKS Commas are optional in *variable_list*.

Deletes the specified variables from the output data set. Any variables referenced must already exist, either as elements of the source data set, or as the result of a previous **make**, **vector**, or **code** statement.

If neither **keep** nor **drop** is used, the output data set will contain all variables from the source data set, as well as any defined variables. The effects of multiple **keep** and **drop** statements are cumulative.

EXAMPLE
```
drop age, pay, sex;
```

SEE ALSO **keep (dataloop)**

dsCreate

PURPOSE Creates an instance of a structure of type **DS** set to default values.

INCLUDE `ds.sdf`

FORMAT *s* **= dsCreate;**

OUTPUT *s* instance of structure of type **DS**.

SOURCE `ds.src`

dstat

PURPOSE Computes descriptive statistics.

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

INPUT *dataset* string, name of data set.

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

vars the variables.

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

K×1 character vector, names of variables.

- or -

K×1 numeric vector, indices of variables.

These can be any size subset of the variables in the data set and can be in any order. If a scalar 0 is passed, all columns of the data set will be used.

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

N×K matrix, the data on which to compute the descriptive statistics.

GLOBAL INPUT

__altnam matrix, default 0.

This can be a K×1 character vector of alternate variable names for the output.

__maxbytes scalar, the maximum number of bytes to be read per iteration of the read loop. Default = 1e9.

__maxvec scalar, the largest number of elements allowed in any one matrix. Default = 20000.

__miss scalar, default 0.

0 there are no missing values (fastest).

1 listwise deletion, drop a row if any missings occur in it.

2 pairwise deletion.

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

if 0, (default) the number of rows will be calculated using **__maxbytes** and **__maxvec**.

__output scalar, controls output, default 1.

1 print output table.

0 do not print output.

OUTPUT

vnam K×1 character vector, the names of the variables used in the statistics.

mean K×1 vector, means.

var K×1 vector, variance.

	std	K×1 vector, standard deviation.
	min	K×1 vector, minima.
	max	K×1 vector, maxima.
	valid	K×1 vector, the number of valid cases.
	mis	K×1 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`

dstatmt

PURPOSE Compute descriptive statistics.

FORMAT *dout* **= dstatmt(***dc0***,***dataset***,***vars***);**

INPUT *dc0* instance of a **dstatmtControl** structure containing the following members:

dc0`.altnames`	K×1 string array of alternate variable names to be used if a matrix in memory is analyzed (i.e., *dataset* is a null string or 0). Default = "".
dc0`.maxbytes`	scalar, the maximum number of bytes to be read per iteration of the read loop. Default = 1e9.
dc0`.maxvec`	scalar, the largest number of elements allowed in any one matrix. Default = 20000.
dc0`.miss`	scalar, default 0. **0** there are no missing values (fastest).

1 listwise deletion, drop a row if any missings occur in it.

2 pairwise deletion.

dc0.row scalar, the number of rows to read per iteration of the read loop.
If 0, (default) the number of rows will be calculated using *dc0*.**maxbytes** and *dc0*.**maxvec**.

dc0.output scalar, controls output, default 1.

1 print output table.

0 do not print output.

dataset string, name of data set.

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

vars the variables.

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

K×1 string array, names of variables.

- or -

K×1 numeric vector, indices of variables.

These can be any size subset of the variables in the data set and can be in any order. If a scalar 0 is passed, all columns of the data set will be used.

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

N×K matrix, the data on which to compute the descriptive statistics.

OUTPUT *dout* instance of a **dstatmtOut** structure containing the following members:

dout.vnames K×1 string array, the names of the variables used in the statistics.

dout.mean K×1 vector, means.

dout.var K×1 vector, variance.

dout.std K×1 vector, standard deviation.

dout.min	K×1 vector, minima.
dout.max	K×1 vector, maxima.
dout.valid	K×1 vector, the number of valid cases.
dout.missing	K×1 vector, the number of missing cases.
dout.errcode	scalar, error code, 0 if successful; otherwise, one of the following: **2** Can't open file. **7** Too many missings - no data left after packing. **9** **altnames** member of **dstatmtControl** structure wrong size. **10** **vartype** member of **dstatmtControl** structure wrong size.

REMARKS If pairwise deletion is used, the minima and maxima will be the true values for the valid data. The means and standard deviations will be computed using the correct number of valid observations for each variable.

SOURCE dstatmt.src

SEE ALSO **dstatmtControlCreate**

dstatmtControlCreate

PURPOSE Creates default **dstatmtControl** structure.

INCLUDE dstatmt.sdf

FORMAT *c* = **dstatmtControlCreate;**

OUTPUT *c* instance of **dstatmtControl** structure with members set to default values.

SOURCE dstatmt.src

SEE ALSO **dstatmt**

dtdate

PURPOSE Creates a matrix in DT scalar format.

FORMAT *dt* = **dtdate**(*year*,*month*,*day*,*hour*,*minute*,*second*);

INPUT

year	N×K matrix of years.
month	N×K matrix of months, 1-12.
day	N×K matrix of days, 1-31.
hour	N×K matrix of hours, 0-23.
minute	N×K matrix of minutes, 0-59.
second	N×K matrix of seconds, 0-59.

OUTPUT

dt	N×K matrix of DT scalar format dates.

REMARKS The arguments must be E×E conformable.

SOURCE time.src

SEE ALSO **dtday**, **dttime**, **utctodt**, **dttostr**

dtday

PURPOSE Creates a matrix in DT scalar format containing only the year, month and day. Time of day information is zeroed out.

FORMAT *dt* = **dtday**(*year*,*month*,*day*);

INPUT

year	N×K matrix of years.
month	N×K matrix of months, 1-12.
day	N×K matrix of days, 1-31.

OUTPUT

dt	N×K matrix of DT scalar format dates.

REMARKS This amounts to 00:00:00 or midnight on the given day. The arguments must be E×E conformable.

SOURCE time.src

SEE ALSO **dttime**, **dtdate**, **utctodt**, **dttostr**

dttime

PURPOSE Creates a matrix in DT scalar format containing only the hour, minute and second. The date information is zeroed out.

FORMAT *dt* = **dttime**(*hour*,*minute*,*second*);

INPUT

hour	N×K matrix of hours, 0-23.
minute	N×K matrix of minutes, 0-59.

second N×K matrix of seconds, 0-59.

OUTPUT *dt* N×K matrix of DT scalar format times.

REMARKS The arguments must be E×E conformable.

SOURCE `time.src`

SEE ALSO **dtday**, **dtdate**, **utctodt**, **dttostr**

dttodtv

PURPOSE Converts DT scalar format to DTV vector format.

FORMAT *dtv* **= dttodtv(***dt***);**

INPUT *dt* N×1 vector, DT scalar format.

OUTPUT *dtv* N×8 matrix, DTV vector format.

REMARKS In DT scalar format, 15:10:55 on July 3, 2005 is 20050703151055.

Each row of *dtv*, in DTV vector format, contains:

[N,1] Year
[N,2] Month in Year, 1-12
[N,3] Day of month, 1-31
[N,4] Hours since midnight, 0-23
[N,5] Minutes, 0-59
[N,6] Seconds, 0-59
[N,7] Day of week, 0-6, 0 = Sunday

[N,8] Days since Jan 1 of current year, 0-365

EXAMPLE

```
dt = 20010326110722;
print "dt = " dt;

 20010326110722

dtv = dttodtv(dt);
print "dtv = " dtv;

 2001    3   26   11    7   22    1   84
```

SOURCE time.src

SEE ALSO **dtvnormal**, **timeutc**, **utctodtv**, **dtvtodt**, **dttoutc**, **dtvtodt**, **strtodt**, **dttostr**

dttostr

PURPOSE Converts a matrix containing dates in DT scalar format to a string array.

FORMAT *sa* **= dttostr(***x***,***fmt***);**

INPUT

x	N×K matrix containing dates in DT scalar format.
fmt	string containing date/time format characters.

OUTPUT

sa	N×K string array.

REMARKS The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number

20050703105031

represents 10:50:31 or 10:50:31 AM on July 3, 2005. **dttostr** converts a date in DT scalar format to a character string using the format string in *fmt*.

The following formats are supported:

YYYY	4 digit year
YR	Last two digits of year
MO	Number of month, 01-12
DD	Day of month, 01-31
HH	Hour of day, 00-23
MI	Minute of hour, 00-59
SS	Second of minute, 00-59

EXAMPLE

```
s0 = dttostr(utctodt(timeutc), "YYYY-MO-DD HH:MI:SS");
print ("Date and Time are: " $+ s0);

Date and time are: 2005-09-14 11:49:10


print dttostr(utctodt(timeutc), "Today is DD-MO-YR");

Today is 14-09-05


s = dttostr(x, "YYYY-MO-DD");
```

If x =

```
20000317060424
20010427031213
20010517020437
20011117161422
20010717120448
20010817043451
20010919052320
20011017032203
20011107071418
```

then s =
2000 – 03 – 17
2001 – 04 – 27
2001 – 05 – 17
2001 – 11 – 17
2001 – 07 – 17
2001 – 08 – 17
2001 – 09 – 19
2001 – 10 – 17
2001 – 11 – 07

SEE ALSO **strtodt**, **dttoutc**, **utctodt**

dttoutc

PURPOSE Converts DT scalar format to UTC scalar format.

FORMAT *utc* **= dttoutc(***dt***);**

INPUT *dt* N×1 vector, DT scalar format.

OUTPUT *utc* N×1 vector, UTC scalar format.

REMARKS In DT scalar format, 10:50:31 on July 15, 2005 is 20050703105031. A UTC scalar gives the number of seconds since or before January 1, 1970 Greenwich Mean Time.

EXAMPLE
```
dt = 20010326085118;
tc = dttoutc(dt);

print "tc = " tc;

tc = 985633642;
```

SOURCE time.src

SEE ALSO **dtvnormal**, **timeutc**, **utctodtv**, **dttodtv**, **dtvtodt**, **dtvtoutc**, **dtvtodt**, **strtodt**, **dttostr**

dtvnormal

PURPOSE Normalizes a date and time (DTV) vector.

FORMAT *d* = **dtvnormal**(*t*);

INPUT *t* 1×8 date and time vector that has one or more elements outside the normal range.

OUTPUT *d* Normalized 1×8 date and time vector.

REMARKS The date and time vector is a 1×8 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
```
format /rd 10,2;
x = { 2005 14 21 6 21 37 0 0 };
d = dtvnormal(x);
```

```
d =  2006   2   21   6   21   37   2   51
```

SEE ALSO **date**, **ethsec**, **etstr**, **time**, **timestr**, **timeutc**, **utctodtv**

dtvtodt

PURPOSE Converts DT vector format to DT scalar format.

FORMAT *dt* **= dtvtodt(***dtv***);**

INPUT *dtv* N×8 matrix, DTV vector format.

OUTPUT *dt* N×1 vector, DT scalar format.

REMARKS In DT scalar format, 11:06:47 on March 15, 2001 is 20010315110647.

Each row of *dtv*, in DTV vector format, contains:

[N,1] Year
[N,2] Month in Year, 1-12
[N,3] Day of month, 1-31
[N,4] Hours since midnight, 0-23
[N,5] Minutes, 0-59
[N,6] Seconds, 0-59
[N,7] Day of week, 0-6, 0 = Sunday
[N,8] Days since Jan 1 of current year, 0-365

EXAMPLE
```
let dtv = { 2005 3 26 11 7 22 1 84 };
dt = dtvtodt(dtv);
```

```
dtv =  2005   3   26   11   7   22   1   84;


dt =  20050326110722
```

SOURCE time.src

SEE ALSO **dtvnormal**, **timeutc**, **utctodtv**, **dttodtv**, **dtvtodt**, **dttoutc**, **dtvtodt**, **strtodt**, **dttostr**

dtvtoutc

PURPOSE Converts DTV vector format to UTC scalar format.

FORMAT *utc* = **dtvtoutc**(*dtv*);

INPUT *dtv* N×8 matrix, DTV vector format.

OUTPUT *utc* N×1 vector, UTC scalar format.

REMARKS A UTC scalar gives the number of seconds since or before January 1, 1970 Greenwich Mean Time.

Each row of *dtv*, in DTV vector format, contains:

[N,1] Year
[N,2] Month in Year, 1-12
[N,3] Day of month, 1-31
[N,4] Hours since midnight, 0-23
[N,5] Minutes, 0-59
[N,6] Seconds, 0-59
[N,7] Day of week, 0-6, 0 = Sunday
[N,8] Days since Jan 1 of current year, 0-365

EXAMPLE
```
dtv = utctodtv(timeutc);
utc = dtvtoutc(dtv);
```

```
dtv =  2005   9   14   14   40   18   3   256
```

```
utc =  1126734018
```

SEE ALSO **dtvnormal**, **timeutc**, **utctodt**, **dttodtv**, **dttoutc**, **dtvtodt**, **dtvtoutc**, **strtodt**, **dttostr**

dummy

PURPOSE Creates a set of dummy (0/1) variables by breaking up a variable into specified categories. The highest (rightmost) category is unbounded on the right.

FORMAT *y* **= dummy(***x*,*v***);**

INPUT

x	N×1 vector of data that is to be broken up into dummy variables.
v	(K-1)×1 vector specifying the K-1 breakpoints (these must be in ascending order) that determine the K categories to be used. These categories should not overlap.

OUTPUT

y	N×K 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  ≤  1
1  <  x  ≤  5
5  <  x  ≤  7
7  <  x
```

SOURCE datatran.src

SEE ALSO **dummybr**, **dummydn**

dummybr

PURPOSE Creates a set of dummy (0/1) variables. The highest (rightmost) category is bounded on the right.

FORMAT *y* **= dummybr(***x*,*v***);**

INPUT

x	N×1 vector of data that is to be broken up into dummy variables.
v	K×1 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	N×K 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:

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

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

$$\begin{matrix} & & \mathbf{x} & \leq & 1 \\ 1 & < & \mathbf{x} & \leq & 5 \\ 5 & < & \mathbf{x} & \leq & 7 \end{matrix}$$

SOURCE datatran.src

SEE ALSO **dummydn**, **dummy**

dummydn

PURPOSE Creates a set of dummy (0/1) variables by breaking up a variable into specified categories. The highest (rightmost) category is unbounded on the right, and a specified column of dummies is dropped.

FORMAT *y* = **dummydn**(*x*,*v*,*p*);

INPUT

x	N×1 vector of data to be broken up into dummy variables.
v	K×1 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	N×(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:

$$\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{array}$$

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

$$\begin{array}{ccccc} & & \mathbf{x} & \leq & 1 \\ 1 & < & \mathbf{x} & \leq & 5 \\ 5 & < & \mathbf{x} & \leq & 7 \\ 7 & < & \mathbf{x} & & \end{array}$$

SOURCE datatran.src

SEE ALSO **dummy**, **dummybr**

ed

PURPOSE Accesses an alternate editor.

FORMAT **ed** *filename*;

INPUT *filename* literal, the name of the file to be edited.

REMARKS The default name of the editor is set in `gauss.cfg`. To change the name of the editor used 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**.

edit

PURPOSE Edits a disk file.

FORMAT **edit** *filename*;

INPUT *filename* literal, the name of the file to be edited.

PORTABILITY **Windows** and **Mac** only

This command loads a disk file in a **GAUSS** edit window. It is available only in the **GAUSS** graphical user interface.

REMARKS The edit command does not follow the `src_path` to locate files. You must specify the location in the *filename*. The default location is the current directory.

EXAMPLE

```
edit test1.e;
```

SEE ALSO **run**

eig

PURPOSE Computes the eigenvalues of a general matrix.

FORMAT *va* = **eig**(*x*);

INPUT *x* N×N matrix or K-dimensional array where the last two dimensions are N×N.

OUTPUT *va* N×1 vector or K-dimensional array where the last two dimensions are N×1, the eigenvalues of *x*.

REMARKS If *x* is an array, the result will be an array containing the eigenvalues of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10×4×4 array, the result will be a 10×4×1 array containing the eigenvalues of each of the 10 4×4 arrays contained in *x*.

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**

eigh

PURPOSE Computes the eigenvalues of a complex hermitian or real symmetric matrix.

FORMAT *va* **= eigh(***x***);**

INPUT *x* N×N matrix or K-dimensional array where the last two dimensions are N×N.

OUTPUT *va* N×1 vector or K-dimensional array where the last two dimensions are N×1, the eigenvalues of *x*.

REMARKS If *x* is an array, the result will be an array containing the eigenvalues of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10×4×4 array, the result will be a 10×4×1 array containing the eigenvalues of each of the 10 4×4 arrays contained in *x*.

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**

eighv

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

FORMAT { *va*,*ve* } **= eighv(***x***);**

INPUT
x N×N matrix or K-dimensional array where the last two dimensions are N×N.

OUTPUT
va N×1 vector or K-dimensional array where the last two dimensions are N×1, the eigenvalues of *x*.

ve N×N matrix or K-dimensional array where the last two dimensions are N×N, the eigenvectors of *x*.

REMARKS If *x* is an array, *va* will be an array containing the eigenvalues of each 2-dimensional array described by the two trailing dimensions of *x*, and *ve* will be an array containing the corresponding eigenvectors. In other words, for a 10×4×4 array, *va* will be a 10×4×1 array containing the eigenvalues and *ve* a 10×4×4 array containing the eigenvectors of each of the 10 4×4 arrays contained in *x*.

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**

eigv

PURPOSE Computes eigenvalues and eigenvectors of a general matrix.

FORMAT { *va*,*ve* } = **eigv**(*x*);

INPUT
x N×N matrix or K-dimensional array where the last two dimensions are N×N.

OUTPUT
va N×1 vector or K-dimensional array where the last two dimensions are N×1, the eigenvalues of *x*.

ve N×N matrix or K-dimensional array where the last two dimensions are N×N, the eigenvectors of *x*.

REMARKS If *x* is an array, *va* will be an array containing the eigenvalues of each 2-dimensional array described by the two trailing dimensions of *x*, and *ve* will be an array containing the corresponding eigenvectors. In other words, for a

10×4×4 array, *va* will be a 10×4×1 array containing the eigenvalues and *ve* a 10×4×4 array containing the eigenvectors of each of the 10 4×4 arrays contained in *x*.

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
          −0.019156716  −0.91339349    1.6734214
```

SEE ALSO **eig**, **eigh**, **eighv**

elapsedTradingDays

PURPOSE Computes number of trading days between two dates inclusively.

FORMAT *n* = **elapsedTradingDays**(*a*,*b*);

INPUT
a scalar, date in DT scalar format.
b scalar, date in DT scalar format.

OUTPUT
n number of trading days between dates inclusively, that is, elapsed time includes the dates *a* and *b*.

REMARKS A trading day is a weekday that is not a holiday as defined by the New York Stock Exchange from 1888 through 2006. Holidays are defined in `holidays.asc`. You may edit that file to modify or add holidays.

SOURCE `finutils.src`

GLOBALS **_fin_holidays**

SEE ALSO **elapsedTradingDays**, **getNextTradingDay**, **getPreviousTradingDay**, **getNextWeekDay**, **getPreviousWeekDay**

end

PURPOSE Terminates a program.

FORMAT **end;**

REMARKS **end** causes **GAUSS** to revert to interactive mode, and closes all open files. **end**

also closes the auxiliary output file and turns the window on. It is not necessary to put an **end** statement at the end of a program.

An **end** command can be placed above a label which begins a subroutine to make sure that a program does not enter a subroutine without a **gosub**.

stop also terminates a program but closes no files and leaves the window setting as it is.

EXAMPLE

```
output on;
screen off;
print x;
end;
```

In this example, a matrix **x** is printed to the auxiliary output. The output to the window is turned off to speed up the printing. The **end** statement is used to terminate the program, so the output file will be closed and the window turned back on.

SEE ALSO **new**, **stop**, **system**

endp

PURPOSE Closes a procedure or keyword definition.

FORMAT **endp;**

REMARKS **endp** marks the end of a procedure definition that began with a **proc** or **keyword** statement. (For details on writing and using procedures, see PROCEDURES AND KEYWORDS, Chapter 12.)

EXAMPLE

```
proc regress(y,x);
    retp(inv(x'x)*x'y);
endp;
```

```
x = { 1 3 2, 7 4 9, 1 1 6, 3 3 2 };
y = { 3, 5, 2, 7 };

b = regress(y,x);
```

```
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

y =  3.00000000
     5.00000000
     2.00000000
     7.00000000

b =   0.15456890
      1.50276345
     −0.12840825
```

SEE ALSO **proc**, **keyword**, **retp**

endwind

PURPOSE Ends graphic panel manipulation; displays 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**

envget

PURPOSE Searches the environment table for a defined name.

FORMAT *y* **= envget(***s***);**

INPUT *s* string, the name to be searched for.

OUTPUT *y* string, the string that corresponds to that name in the environment table or a null string if it is not found.

EXAMPLE
```
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 that will open a data file using a path stored in an environment string called DPATH. The procedure returns the file handle and is called as follows:

```
fp = dopen(''myfile'');
```

SEE ALSO **cdir**

eof

PURPOSE Tests if the end of a file has been reached.

FORMAT *y* **= eof(***fh***);**

INPUT *fh* scalar, file handle.

OUTPUT *y* scalar, 1 if end of file has been reached, else 0.

REMARKS This function is used with the **readr** and **fgets***xxx* commands to test for the end of a file.

The **seekr** function can be used to set the pointer to a specific row position in a data set; the **fseek** function can be used to set the pointer to a specific byte offset in a file opened with **fopen**.

EXAMPLE
```
open f1 = dat1;
xx = 0;
do until eof(f1);
    xx = xx+moment(readr(f1,100),0);
endo;
```

In this example, the data file `dat1.dat` is opened and given the handle **f1**. Then the data are read from this data set and are used to create the moment matrix (**x′x**) of the data. On each iteration of the loop, 100 additional rows of data are read in, and the moment matrix for this set of rows is computed and added to the matrix **xx**. When all the data have been read, **xx** will contain the entire moment matrix for the data set.

GAUSS will keep reading until **eof(f1)** returns the value 1, which it will when the end of the data set has been reached. On the last iteration of the loop, all remaining observations are read in if there are 100 or fewer left.

SEE ALSO **open**, **readr**, **seekr**

eqSolve

PURPOSE Solves a system of nonlinear equations

FORMAT { *x*,*retcode* } = **eqSolve(&***F*,*start*);

INPUT

start K×1 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 K×1 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 K×1 vector of the typical magnitude of x, used for scaling. This becomes important when the magnitudes of the components of x are expected to be very different. By default, variable values are not scaled.

_eqs_IterInfo scalar, if nonzero, iteration information is printed. Default = 0.

The following are set by **gausset**:

__Tol scalar, the tolerance of the scalar function $f = 0.5 * ||F(x)||^2$ required to terminate the algorithm. Default = 1e-5.

__altnam K×1 character vector of alternate names to be used by the printed output. By default, the names "X1, X2,X3..." or "X01,X02,X03..." (depending on how **__vpad** is set) will be used.

__output scalar. If non-zero, final results are printed.

__title string, a custom title to be printed at the top of the iterations report. By default, only a generic title will be printed.

__vpad scalar. If **__altnam** is not set, variable names are automatically created. Two types of names can be created:

- **0** Variable names are not padded to give them equal length. For example, X1, X2,. . .,X10,. . .
- **1** Variable names are padded with zeros to give them an equal number of characters. For example, X01,X02,. . .,X10,. . . This is useful if you want the variable names to sort properly.

OUTPUT *x* K×1 vector, solution.

retcode scalar, the return code:

- **1** Norm of the scaled function value is less than **__Tol**. *x* 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**). *x* 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 norm2($F(x)$) sufficiently; either *x* 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 norm2($F(x)$) asymptotes from above to a finite value in some direction or the maximum step length is too small.

6 x seems to be an approximate local minimizer of norm2($F(x)$) 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;

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

_eqs_JacobianProc = &fjc;

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

produces:

```
================================================================================
 EqSolve Version 6.5.0                            8/18/2005   3:33 pm
================================================================================

||F(X)|| at final solution:                                     0.93699762
--------------------------------------------------------------------------------
Termination Code = 1:

Norm of the scaled function value is less than __Tol;
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
VARIABLE          START              ROOTS               F(ROOTS)
--------------------------------------------------------------------------------
X1             -1.00000           0.54144351         4.4175402e-06
X2             12.00000            1.4085912        -6.6263102e-06
X3             -1.00000            1.1111111         4.4175402e-06
--------------------------------------------------------------------------------
```

SOURCE eqsolve.src

eqSolvemt

PURPOSE Solves a system of nonlinear equations.

INCLUDE eqsolvemt.sdf

FORMAT *out* = **eqSolvemt(&***fct*,*par*,*data*,*c*);

INPUT

&*fct* pointer to a procedure that computes the function to be minimized. This procedure must have two input arguments, an instance of a **PV** structure containing the parameters, and an instance of a **DS** structure containing data, if any. And, one output argument, a column vector containing the result of each equation.

par an instance of a **PV** structure. The *par* instance is passed to the user-provided procedure pointed to by **&***fct*. *par* is constructed using the **pvPack** functions.

data an array of instances of a **DS** structure. This array is passed to the user-provided procedure pointed to by **&***fct* to be used in the objective function. **eqSolvemt** does not look at this structure. Each instance contains the the following members which can be set in whatever way that is convenient for computing the objective function:

data1[*i*].dataMatrix N×K matrix, data matrix.
data1[*i*].dataArray N×K×L... array, data array.
data1[*i*].vnames string array, variable names (optional).
data1[*i*].dsname string, data name (optional).
data1[*i*].type scalar, type of data (optional).

c an instance of an **eqSolvemtControl** structure. Normally an instance is initialized by calling **eqSolvemtControlCreate** and members of this instance can be set to other values by the user. For an instance named *c*, the members are:

c.jacobianProc pointer to a procedure which computes the analytical Jacobian. By default, **eqSolvemt** will compute the Jacobian numerically.

c.maxIters scalar, the maximum number of iterations. Default = 100.

c.stepTolerance scalar, the step tolerance. Default = macheps$^{2/3}$.

c.typicalF K×1 vector of the typical $fct(X)$ values at a point not near a root, used for scaling. This becomes important when the magnitudes of the components of $fct(X)$ are expected to be very different. By default, function values are not scaled.

c.typicalX K×1 vector of the typical magnitude of X, used for scaling. This becomes important when the magnitudes of the components of X are expected to be very different. By default, variable values are not scaled.

c.printIters scalar, if nonzero, iteration information is printed. Default = 0.

c.tolerance scalar, the tolerance of the scalar function $f = 0.5 * ||fct(X)||^2$ required to terminate the algorithm. That is, the condition that $| f(x) |$ <= *c*.**tolerance** must be met before that algorithm can terminate successfully. Default = 1e-5.

c.altnam K×1 character vector of alternate names to be used by the printed output. By default, the names "X1,X2,X3..." will be used.

c.title string, printed as a title in output.

c.output scalar. If non-zero, final results are printed.

OUTPUT *out* an instance of an **eqsolvemtOut** structure. For an instance named *out*, the members are:

out.par an instance of a **PV** structure containing the parameter estimates.

out.fct scalar, function evaluated at *X*.

out.retcode scalar, return code:

-1 Jacobian is singular.

1 Norm of the scaled function value is less than *c*.**tolerance**. *X* given is an approximate root of *fct*(*X*) (unless *c*.**tolerance** is too large).

2 The scaled distance between the last two steps is less than the step-tolerance (*c*.**stepTolerance**). *X* may be an approximate root of *fct*(*X*), but it is also possible that the algorithm is making very slow progress and is not near a root, or the step-tolerance is too large.

3 The last global step failed to decrease norm2(*fct*(*X*)) sufficiently; either *X* is close to a root of *fct*(*X*) and no more accuracy is possible, or an incorrectly coded analytic Jacobian is being used, or the secant approximation to the Jacobian is inaccurate, or the step-tolerance is too large.

4 Iteration limit exceeded.

5 Five consecutive steps of maximum step length have been taken; either norm2(*fct*(*X*)) asymptotes from above to a finite value in some direction or the maximum step length is too small.

6 *X* seems to be an approximate local minimizer of norm2(*fct*(*X*)) that is not a root of *fct*(*X*). To find a root of *fct*(*X*), restart **eqSolvemt** from a different region.

REMARKS The equation procedure should return a column vector containing the result for each equation.

If there is no data, you can pass an empty **DS** structure in the second argument:

```
call eqSolvemt(&fct,par,dsCreate,c);
```

EXAMPLE

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

```
#include eqSolvemt.sdf;
struct eqSolvemtControl c;
c = eqSolvemtControlCreate;

c.printIters = 1;

struct PV par;
par = pvPack(pvCreate,1,"x1");
par = pvPack(par,1,"x2");

struct eqSolvemtOut out1;
out1 = eqSolvemt(&fct,par,dsCreate,c);

proc fct(struct PV p, struct DS d);
    local x1, x2, z;
    x1 = pvUnpack (p, "x1");
    x2 = pvUnpack (p, "x2");
    z = x1^2+x2^2-2 | exp(x1-1)+x2^3-2;
    retp(z);
endp;
```

SOURCE eqsolvemt.src

SEE ALSO **eqsolvemtControlCreate**, **eqsolvemtOutCreate**

eqSolvemtControlCreate

PURPOSE Creates default **eqSolvemtControl** structure.

INCLUDE `eqsolvemt.sdf`

FORMAT *c* = **eqSolvemtControlCreate;**

OUTPUT *c* instance of **eqSolvemtControl** structure with members set to default values.

SOURCE `eqsolvemt.src`

SEE ALSO **eqsolvemt**

eqSolvemtOutCreate

PURPOSE Creates default **eqSolvemtOut** structure.

FORMAT *c* = **eqSolvemtOutCreate;**

OUTPUT *c* instance of **eqSolvemtOut** structure with members set to default values.

SOURCE `eqsolvemt.src`

SEE ALSO **eqsolvemt**

eqSolveSet

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

FORMAT **eqSolveset;**

GLOBAL OUTPUT

__eqs_TypicalX	Set to 0.
__eqs_TypicalF	Set to 0.
__eqs_IterInfo	Set to 0.
__eqs_JacobianProc	Set to 0.
__eqs_MaxIters	Set to 100.
__eqs_StepTol	Set to **__macheps**$^{2/3}$

erf, erfc

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

FORMAT *y* = **erf**(*x*);

y = **erfc**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* N×K matrix.

REMARKS The allowable range for *x* is:

$$x >= 0$$

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

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

EXAMPLE

```
x = { .5 .4 .3,
      .6 .8 .3 };
y = erf(x);
```

```
y =  0.52049988  0.42839236  0.32862676
     0.60385609  0.74210096  0.32862676
```

```
x = { .5 .4 .3,
      .6 .8 .3 };
y = erfc(x);
```

```
y =  0.47950012  0.57160764  0.67137324
     0.39614391  0.25789904  0.67137324
```

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.

error

PURPOSE Allows the user to generate a user-defined error code which can be tested quickly with the **scalerr** function.

FORMAT *y* **= error(***x***);**

INPUT *x* scalar, in the range 0–65535.

OUTPUT *y* scalar error code which can be interpreted as an integer with the **scalerr** function.

REMARKS The user may assign any number in the range 0–65535 to denote particular error conditions. This number may be tested for as an error code by **scalerr**.

The **scalerr** function will return the value of the error code and so is the reverse of **error**. These user-generated error codes work in the same way as the intrinsic **GAUSS** error codes which are generated automatically when **trap 1** is on and certain **GAUSS** functions detect a numerical error such as a singular matrix.

error(0) is equal to the missing value code.

EXAMPLE
```
proc syminv(x);
    local oldtrap,y;
    if not x =\,= x';
        retp(error(99));
    endif;
    oldtrap  = trapchk(0xffff);
    trap 1;
    y = invpd(x);
    if scalerr(y);
        y  = inv(x);
    endif;
    trap oldtrap,0xffff;
    retp(y);
endp;
```

The procedure **syminv** returns error code 99 if the matrix is not symmetric. If **invpd** fails, it returns error code 20. If **inv** fails, it returns error code 50. The original trap state is restored before the procedure returns.

SEE ALSO **scalerr**, **trap**, **trapchk**

errorlog

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

FORMAT **errorlog** *str*;

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

REMARKS This command enables you to do your own error handling in your **GAUSS** programs. To print an error message to the window and error log file along with file name and line number information, use **errorlogat**.

SEE ALSO **errorlogat**

errorlogat

PURPOSE Prints an error message to the window and error log file, along with the file name and line number at which the error occurred.

FORMAT **errorlogat** *str*;

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

REMARKS This command enables you to do your own error handling in your **GAUSS** programs. To print an error message to the window and error log file without file name and line number information, use **errorlog**.

SEE ALSO **errorlog**

etdays

PURPOSE Computes the difference between two times, as generated by the **date** command, in days.

FORMAT *days* = **etdays**(*tstart*,*tend*);

INPUT

tstart 3×1 or 4×1 vector, starting date, in the order: yr, mo, day. (Only the first 3 elements are used.)

tend 3×1 or 4×1 vector, ending date, in the order: yr, mo, day. (Only the first 3 elements are used.) MUST be later than *tstart*.

OUTPUT

days scalar, elapsed time measured in days.

REMARKS This will work correctly across leap years and centuries. The assumptions are a Gregorian calendar with leap years on the years evenly divisible by 4 and not evenly divisible by 100, unless divisible by 400.

EXAMPLE

```
let date1 = 2004 1 2;
let date2 = 2005 9 14;
d = etdays(date1,date2);
```

```
d = 621
```

SOURCE time.src

SEE ALSO **dayinyr**

ethsec

PURPOSE Computes the difference between two times, as generated by the **date** command, in hundredths of a second.

FORMAT *hs* = **ethsec**(*tstart*,*tend*);

INPUT

tstart	4×1 vector, starting date, in the order: yr, mo, day, hundredths of a second.
tend	4×1 vector, ending date, in the order: yr, mo, day, hundredths of a second. MUST be later date than *tstart*.

OUTPUT

hs	scalar, elapsed time measured in hundredths of a second.

REMARKS This will work correctly across leap years and centuries. The assumptions are a Gregorian calendar with leap years on the years evenly divisible by 4 and not evenly divisible by 100, unless divisible by 400.

EXAMPLE

```
let date1 = 2004 1 2 0;
let date2 = 2005 9 14 0;
t = ethsec(date1,date2);
```

```
t = 5365440000
```

SOURCE time.src

SEE ALSO **dayinyr**

etstr

PURPOSE Formats an elapsed time measured in hundredths of a second to a string.

FORMAT *str* **= etstr(***tothsecs***);**

INPUT *tothsecs* scalar, an elapsed time measured in hundredths of a second, as given, for instance, by the **ethsec** function.

OUTPUT *str* string containing the elapsed time in the form:
days # hours # minutes #,## seconds

EXAMPLE
```
d1 = { 2004, 1, 2, 0 };
d2 = { 2004, 9, 14, 815642 };
t = ethsec(d1,d2);
str = etstr(t);
```

t = 5366255642

str = 34 days 2 hours 15 minutes 56.42 seconds

SOURCE time.src

SEE ALSO **ethsec**

EuropeanBinomCall

PURPOSE Prices European call options using binomial method.

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

OUTPUT

c	M×1 vector, call premiums.

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

EXAMPLE

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
c = EuropeanBinomCall(S0,K,r,0,tau,sigma,60);
print c;

   17.344044
   15.058486
   12.817427
```

SOURCE `finprocs.src`

EuropeanBinomCall_Greeks

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

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

GLOBAL INPUT

_fin_thetaType scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.

_fin_epsilon scalar, finite difference stepsize. Default = 1e-8.

OUTPUT

d	M×1 vector, delta.
g	M×1 vector, gamma.
t	M×1 vector, theta.
v	M×1 vector, vega.
rh	M×1 vector, rho.

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

EXAMPLE

```
S0 = 305;
K = 300;
r = .08;
sigma = .25;
tau = .33;
div = 0;
print EuropeanBinomcall_Greeks(S0,K,r,0,tau,sigma,30);

      0.70631204
   0.00076381912
      -44.616125
       68.703851
       76.691829
```

SOURCE finprocs.src

SEE ALSO **EuropeanBinomCall_Impvol**, **EuropeanBinomCall**, **EuropeanBinomPut_Greeks**, **EuropeanBSCall_Greeks**

EuropeanBinomCall_ImpVol

PURPOSE Computes implied volatilities for European call options using binomial method.

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

INPUT

c	M×1 vector, call premiums.
S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
N	number of time segments.

OUTPUT *sigma* M×1 vector, volatility.

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

EXAMPLE
```
c = { 13.70, 11.90, 9.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
div = 0;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
sigma = EuropeanBinomCall_ImpVol(c,S0,K,r,0,tau,30);
print sigma;

  0.19632223
  0.16988741
  0.12879447
```

SOURCE `finprocs.src`

EuropeanBinomPut

PURPOSE Prices European put options using binomial method.

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

INPUT
	S0	scalar, current price.
	K	M×1 vector, strike prices.
	r	scalar, risk free rate.

	div	continuous dividend yield.
	tau	scalar, elapsed time to exercise in annualized days of trading.
	sigma	scalar, volatility.
	N	number of time segments.
OUTPUT	*c*	M×1 vector, put premiums.

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

EXAMPLE
```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
c = EuropeanBinomPut(S0,K,r,0,tau,sigma,60);
print c;

   16.851815
   19.580390
   22.353464
```

SOURCE `finprocs.src`

EuropeanBinomPut_Greeks

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

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.
N	number of time segments.

GLOBAL INPUT

_fin_thetaType scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.

_fin_epsilon scalar, finite difference stepsize. Default = 1e-8.

OUTPUT

d	M×1 vector, delta.
g	M×1 vector, gamma.
t	M×1 vector, theta.
v	M×1 vector, vega.
rh	M×1 vector, rho.

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

EXAMPLE

```
S0 = 305;
K = 300;
r = .08;
div = 0;
sigma = .25;
tau = .33;
print EuropeanBinomPut_Greeks(S0,K,r,0,tau,sigma,60);
```

```
-0.36885112
0.0011457287
6.6396424
68.979259
-33.796807
```

SOURCE finprocs.src

SEE ALSO **EuropeanBinomPut_Impvol**, **EuropeanBinomPut**, **EuropeanBinomCall_Greeks**, **EuropeanBSPut_Greeks**

EuropeanBinomPut_ImpVol

PURPOSE Computes implied volatilities for European put options using binomial method.

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

INPUT

c	M×1 vector, put premiums.
S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
N	number of time segments.

OUTPUT

sigma	M×1 vector, volatility.

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

EXAMPLE

```
p = { 14.60, 17.10, 20.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
div = 0;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
sigma = EuropeanBinomPut_ImpVol(p,S0,K,r,0,tau,30);
print sigma;

  0.13006393
  0.17043648
  0.21499803
```

SOURCE finprocs.src

EuropeanBSCall

PURPOSE Prices European call options using Black, Scholes and Merton method.

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

OUTPUT

c	M×1 vector, call premiums.

EXAMPLE

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
c = EuropeanBSCall(S0,K,r,0,tau,sigma);
print c;

   17.249367
   14.908466
   12.796356
```

SOURCE finprocs.src

EuropeanBSCall_Greeks

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

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

INPUT

S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.
sigma	scalar, volatility.

GLOBAL INPUT **_fin_thetaType** scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.

_fin_epsilon scalar, finite difference stepsize. Default = 1e-8.

OUTPUT

d	M×1 vector, delta.
g	M×1 vector, gamma.
t	M×1 vector, theta.
v	M×1 vector, vega.
rh	M×1 vector, rho.

EXAMPLE

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

      0.64458005
    0.0085029307
      -38.505439
       65.256273
       56.872007
```

SOURCE finprocs.src

SEE ALSO **EuropeanBSCall_Impvol**, **EuropeanBSCall**, **EuropeanBSPut_Greeks**, **EuropeanBinomCall_Greeks**

EuropeanBSCall_ImpVol

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

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

INPUT

c	M×1 vector, call premiums.
S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.

OUTPUT

sigma	M×1 vector, volatility.

EXAMPLE

```
c = { 13.70, 11.90, 9.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
sigma = EuropeanBSCall_ImpVol(c,S0,K,r,0,tau);
print sigma;

  0.19724517
  0.17084848
  0.12978762
```

SOURCE finprocs.src

EuropeanBSPut

PURPOSE Prices European put options using Black, Scholes, and Merton method.

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

INPUT

S0	scalar, current price.

	K	M×1 vector, strike prices.
	r	scalar, risk free rate.
	div	continuous dividend yield.
	tau	scalar, elapsed time to exercise in annualized days of trading.
	sigma	scalar, volatility.
OUTPUT	*c*	M×1 vector, put premiums.

EXAMPLE

```
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
sigma = .2493;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
c = EuropeanBSPut(S0,K,r,0,tau,sigma);
print c;

   16.759909
   19.404914
   22.278710
```

SOURCE `finprocs.src`

EuropeanBSPut_Greeks

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

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

INPUT	*S0*	scalar, current price.

	K	M×1 vector, strike prices.
	r	scalar, risk free rate.
	div	continuous dividend yield.
	tau	scalar, elapsed time to exercise in annualized days of trading.
	sigma	scalar, volatility.

GLOBAL INPUT

_fin_thetaType scalar, if 1, one day look ahead, else, infinitesmal. Default = 0.

_fin_epsilon scalar, finite difference stepsize. Default = 1e-8.

OUTPUT

	d	M×1 vector, delta.
	g	M×1 vector, gamma.
	t	M×1 vector, theta.
	v	M×1 vector, vega.
	rh	M×1 vector, rho.

EXAMPLE

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

   -0.35541995
  0.0085029307
    -15.130748
     65.256273
    -39.548591
```

SOURCE finprocs.src

SEE ALSO **EuropeanBSPut_Impvol**, **EuropeanBSPut**, **EuropeanBSCall_Greeks**, **EuropeanBinomPut_Greeks**

EuropeanBSPut_ImpVol

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

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

INPUT

c	M×1 vector, put premiums
S0	scalar, current price.
K	M×1 vector, strike prices.
r	scalar, risk free rate.
div	continuous dividend yield.
tau	scalar, elapsed time to exercise in annualized days of trading.

OUTPUT

sigma	M×1 vector, volatility.

EXAMPLE

```
p = { 14.60, 17.10, 20.10 };
S0 = 718.46;
K = { 720, 725, 730 };
r = .0498;
t0 = dtday(2001, 1, 30);
t1 = dtday(2001, 2, 16);
tau = elapsedTradingDays(t0,t1) / annualTradingDays(2001);
sigma = EuropeanBSPut_ImpVol(p,S0,K,r,0,tau);
print sigma;

  0.13222999
  0.17115393
  0.21666496
```

SOURCE finprocs.src

exctsmpl

PURPOSE Computes a random subsample of a data set.

FORMAT *n* = **exctsmpl**(*infile*,*outfile*,*percent*);

INPUT

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

OUTPUT

n scalar, number of rows in output data set.

Error returns are controlled by the low bit of the trap flag:

trap 0	terminate with error message
trap 1	return scalar negative integer
	-1 can't open input file
	-2 can't open output file
	-3 disk full

REMARKS Random sampling is done with replacement. Thus, an observation may be in the resulting sample more than once. If *percent* is 100, the resulting sample will not be identical to the original sample, though it will be the same size.

EXAMPLE

```
n = exctsmpl("freqdata.dat","rout",30);
```

```
n =  120
```

`freqdata.dat` is an example data set provided with **GAUSS**. Switching to the `examples` subdirectory of your **GAUSS** installation directory will make it possible to do the above example as shown. Otherwise you will need to substitute another data set name for "`freqdata.dat`".

SOURCE exctsmpl.src

exec

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

FORMAT *y* **= exec(***program***,***comline***);**

INPUT

program string, the name of the program, including the extension, to be executed.

comline string, the arguments to be placed on the command line of the program being executed.

OUTPUT

y scalar, the exit code returned by *program*.

If **exec** can't execute *program*, the error returns will be negative:

- **-1** file not found
- **-2** the file is not an executable file
- **-3** not enough memory
- **-4** command line too long

EXAMPLE

```
y = exec("atog","comd1.cmd");
if y;
    errorlog "atog failed";
    end;
endif;
```

In this example the ATOG ASCII conversion utility is executed under the **exec** function. The name of the command file to be used, `comd1.cmd`, is passed to ATOG on its command line. The exit code **y** returned by **exec** is tested to see if ATOG was successful; if not, the program will be terminated after printing an error message. See ATOG, Chapter 26.

execbg

PURPOSE Executes an executable program in the background and returns the process id to **GAUSS**.

FORMAT *pid* **= execbg(***program***,***comline***);**

INPUT *program* string, the name of the program, including the extension, to be executed.

comline string, the arguments to be placed on the command line of the program being executed.

OUTPUT *pid* scalar, the process id of the executable returned by program.

If **execbg** cannot execute program, the error returns will be negative:

- **-1** file not found
- **-2** the file is not an executable file
- **-3** not enough memory
- **-4** command line too long

EXAMPLE
```
y = execbg("atog.exe","comd1.cmd");
if (y < 0);
    errorlog "atog failed";
    end;
endif;
```

In this example, the ATOG ASCII conversion utility is executed under the **execbg** function. The name of the command file to be used, `comd1.cmd`, is passed to ATOG on its command line. The returned value, **y**, is tested to see whether ATOG was successful. If not successful the program terminates after printing an error message. See ATOG, Chapter 26.

exp

PURPOSE Calculates the exponential function.

FORMAT *y* = **exp**(*x*);

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array containing *e*, the base of natural logs, raised to the powers given by the elements of *x*.

EXAMPLE
```
x = eye(3);
y = exp(x);
```

```
      1.000000  0.000000  0.000000
x =   0.000000  1.000000  0.000000
      0.000000  0.000000  1.000000


      2.718282  1.000000  1.000000
y =   1.000000  2.718282  1.000000
      1.000000  1.000000  2.718282
```

This example creates a 3×3 identity matrix and computes the exponential function for each one of its elements. Note that **exp(1)** returns *e*, the base of natural logs.

SEE ALSO **ln**

extern (dataloop)

PURPOSE Allows access to matrices or strings in memory from inside a data loop.

FORMAT **extern** *variable_list*;

REMARKS Commas in *variable_list* are optional.

extern tells the translator not to generate local code for the listed variables, and not to assume that they are elements of the input data set.

extern statements should be placed before any reference to the symbols listed. The specified names should not exist in the input data set, or be used in a **make** statement.

EXAMPLE This example shows how to assign the contents of an external vector to a new variable in the data set, by iteratively assigning a range of elements to the variable. The reserved variable **x_x** contains the data read from the input data set on each iteration. The external vector must have at least as many rows as the data set.

```
base = 1;       /* used to index a range of */
                /* elements from exvec */
dataloop oldata newdata;
    extern base, exvec;
    make ndvar = exvec[seqa(base,1,rows(x_x))];
    # base = base + rows(x_x);  /* execute command */
                                /* literally */
endata;
```

external

PURPOSE Lets the compiler know about symbols that are referenced above or in a separate file from their definitions.

FORMAT **external proc** *dog*,*cat*;

external keyword *dog*;

external fn *dog*;

external matrix *x*,*y*,*z*;

external string *mstr*,*cstr*;

external array *a*,*b*;

external sparse matrix *sma*,*smb*;

external struct *structure_type sta*,*stb*;

REMARKS See PROCEDURES AND KEYWORDS, Chapter 12.

You may have several procedures in different files that reference the same global variable. By placing an **external** statement at the top of each file, you can let the compiler know what the type of the symbol is. If the symbol is listed and strongly typed in an active library, no **external** statement is needed.

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

EXAMPLE Let us suppose that you created a set of procedures defined in different files, which all set a global matrix **_errcode** to some scalar error code if errors were encountered.

You could use the following code to call one of the procedures in the set and check whether it succeeded:

```
external matrix _errcode;

x = rndn(10,5);
y = myproc1(x);
```

```
if _errcode;
    print "myproc1 failed";
    end;
endif;
```

Without the **external** statement, the compiler would assume that **_errcode** was a procedure and incorrectly compile this program. The file containing the **myproc1** procedure must also contain an **external** statement that defines **_errcode** as a matrix, but this would not be encountered by the compiler until the **if** statement containing the reference to **_errcode** in the main program file had already been incorrectly compiled.

SEE ALSO **declare**

eye

PURPOSE Creates an identity matrix.

FORMAT *y* **= eye(***n***);**

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

OUTPUT *y* *n*×*n* identity matrix.

REMARKS If *n* is not an integer, it will be truncated to an integer.

The matrix created will contain 1's down the diagonal and 0's everywhere else.

EXAMPLE
```
x = eye(3);
```

```
      1.000000  0.000000  0.000000
x =   0.000000  1.000000  0.000000
      0.000000  0.000000  1.000000
```

SEE ALSO **zeros**, **ones**

fcheckerr

PURPOSE Gets the error status of a file.

FORMAT *err* = **fcheckerr**(*f*);

INPUT *f* scalar, file handle of a file opened with **fopen**.

OUTPUT *err* scalar, error status.

REMARKS If there has been a read or write error on a file, **fcheckerr** returns 1, otherwise 0.

If you pass **fcheckerr** the handle of a file opened with **open** (i.e., a data set or matrix file), your program will terminate with a fatal error.

fclearerr

PURPOSE Gets the error status of a file, then clears it.

FORMAT *err* = **fclearerr**(*f*);

INPUT *f* scalar, file handle of a file opened with **fopen**.

OUTPUT *err* scalar, error status.

REMARKS Each file has an error flag that gets set when there is an I/O error on the file. Typically, once this flag is set, you can no longer do I/O on the file, even if the

error is a recoverable one. **fclearerr** clears the file's error flag, so you can attempt to continue using it.

If there has been a read or write error on a file, **fclearerr** returns 1, otherwise 0.

If you pass **fclearerr** the handle of a file opened with **open** (i.e., a data set or matrix file), your program will terminate with a fatal error.

The flag accessed by **fclearerr** is not the same as that accessed by **fstrerror**.

feq, fge, fgt, fle, flt, fne

PURPOSE Fuzzy comparison functions. These functions use **_fcmptol** to fuzz the comparison operations to allow for roundoff error.

FORMAT *y* = **feq**(*a*,*b*);

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

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

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

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

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

INPUT

a	N×K matrix, first matrix.
b	L×M matrix, second matrix, E×E compatible with *a*.

GLOBAL INPUT

_fcmptol	scalar, comparison tolerance. The default value is 1.0e-15.

OUTPUT *y* scalar, 1 (TRUE) or 0 (FALSE).

REMARKS The return value is TRUE if every comparison is TRUE.

The statement:

```
y = feq(a,b);
```

is equivalent to:

```
y = a eq b;
```

For the sake of efficiency, these functions are not written to handle missing values. If *a* and *b* contain missing values, use **missrv** to convert the missing values to something appropriate before calling a fuzzy comparison function.

The calling program can reset **_fcmptol** before calling these procedures:

```
_fcmptol = 1e-12;
```

EXAMPLE

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

```
x =  0.038289504   0.072535275
     0.014713947    0.96863611

y =    0.25622293   0.70636474
     0.0036191244   0.35913385

t =  0.0000000
```

SOURCE fcompare.src

SEE ALSO **dotfeq--dotfne**

feqmt, fgemt, fgtmt, flemt, fltmt, fnemt

PURPOSE Fuzzy comparison functions. These functions use the *fcmptol* argument to fuzz the comparison operations to allow for roundoff error.

FORMAT *y* = **feqmt**(*a*,*b*,*fcmptol*);

y = **fgemt**(*a*,*b*,*fcmptol*);

y = **fgtmt**(*a*,*b*,*fcmptol*);

y = **flemt**(*a*,*b*,*fcmptol*);

y = **fltmt**(*a*,*b*,*fcmptol*);

y = **fnemt**(*a*,*b*,*fcmptol*);

INPUT

a	N×K matrix, first matrix.
b	L×M matrix, second matrix, E×E compatible with *a*.
fcmptol	scalar, comparison tolerance.

OUTPUT

y	scalar, 1 (TRUE) or 0 (FALSE).

REMARKS The return value is TRUE if every comparison is TRUE.

The statement:

```
y = feqmt(a,b,1e-15);
```

is equivalent to:

```
y = a eq b;
```

For the sake of efficiency, these functions are not written to handle missing values. If *a* and *b* contain missing values, use **missrv** to convert the missing values to something appropriate before calling a fuzzy comparison function.

EXAMPLE

```
x = rndu(2,2);
y = rndu(2,2);
t = fgemt(x,y,1e-14);
```

```
x =  0.038289504   0.072535275
     0.014713947    0.96863611

y =    0.25622293   0.70636474
     0.0036191244   0.35913385

t =  0.0000000
```

SOURCE fcomparemt.src

SEE ALSO **dotfeqmt--dotfnemt**

fflush

PURPOSE Flushes a file's output buffer.

FORMAT *ret* = **fflush**(*f*);

INPUT *f* scalar, file handle of a file opened with **fopen**.

OUTPUT *ret* scalar, 0 if successful, -1 if not.

REMARKS If **fflush** fails, you can call **fstrerror** to find out why.

If you pass **fflush** the handle of a file opened with **open** (i.e., a data set or matrix file), your program will terminate with a fatal error.

fft

PURPOSE Computes a 1- or 2-D Fast Fourier transform.

FORMAT *y* **= fft(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* L×M 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  −1.0000000
     −0.5000000  0.00000000
```

SEE ALSO **fft**, **rfft**, **rfftI**

ffti

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

FORMAT *y* = **ffti**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* L×M 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 = { 22 24,
      23 25 };
y = fft(x);
```

```
y =  23.500000  -1.0000000
     -0.5000000  0.00000000
```

```
fi = ffti(y);
```

```
fi =  22.000000  24.000000
      23.000000  25.000000
```

SEE ALSO **fft**, **rfft**, **rfftI**

fftm

PURPOSE Computes a multi-dimensional FFT.

FORMAT *y* = **fftm**(*x*,*dim*);

INPUT

	x	M×1 vector, data.
	dim	K×1 vector, size of each dimension.

OUTPUT

	y	L×1 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 2×2×2×2 hypercubic **x** would consist of two cubes in sequence, each cube containing two matrices in sequence, each matrix containing two rows in sequence, and each row containing two columns in sequence. Visually, **x** would look something like this:

$$X_{hyper} = X_{cube1} \mid X_{cube2}$$
$$X_{cube1} = X_{mat1} \mid X_{mat2}$$
$$X_{mat1} = X_{row1} \mid X_{row2}$$
$$X_{row1} = X_{col1} \mid X_{col2}$$

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 2×3×2×3 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 2^{nd} 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 2×3×3 three-dimensional array, i.e., two 3×3 matrices of data. Suppose that **x1** is the first 3×3 matrix and **x2** the second 3×3 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 2×3×2×3 hypercube example, **x** would be padded from 2×3×2×3 out to 2×4×2×4. The input vector would contain 36 elements, while the output vector would contain 64 elements. You may have noticed that we used a **dimi** with padded values at the end of the example to check our answer.

SOURCE `fftm.src`

SEE ALSO **fftmi**, **fft**, **ffti**, **fftn**

fftmi

PURPOSE Computes a multi-dimensional inverse FFT.

FORMAT *y* **= fftmi(***x***,***dim***);**

INPUT *x* M×1 vector, data.

dim K×1 vector, size of each dimension.

OUTPUT *y* L×1 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 2×2×2×2 hypercubic **x** would consist of two cubes in sequence, each cube containing two matrices in sequence, each matrix containing two rows in sequence, and each row containing two columns in sequence. Visually, **x** would look something like this:

$$X_{hyper} = X_{cube1} \mid X_{cube2}$$
$$X_{cube1} = X_{mat1} \mid X_{mat2}$$
$$X_{mat1} = X_{row1} \mid X_{row2}$$
$$X_{row1} = X_{col1} \mid X_{col2}$$

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 2×3×2×3 hypercube.

```
let dim = 2 3 2 3;

let x1[2,3] = 1 2 3 4 5 6;
let x2[2,3] = 6 5 4 3 2 1;
let x3[2,3] = 1 2 3 5 7 11;
xc1 = vecr(x1)|vecr(x2)|vecr(x3);    /* cube 1 */

let x1 = 1 1 2 3 5 8;
let x2 = 1 2 6 24 120 720;
let x3 = 13 17 19 23 29 31;
xc2 = x1|x2|x3;                      /* cube 2 */

xh = xc1|xc2;                        /* hypercube */
xhffti = fftmi(xh,dim);
```

We left out the **vecr** step for the 2^{nd} 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 2×3×3 three-dimensional array, i.e., two 3×3 matrices of data. Suppose that **x1** is the first 3×3 matrix and **x2** the second 3×3

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 2×3×2×3 hypercube example, **x** would be padded from 2×3×2×3 out to 2×4×2×4. The input vector would contain 36 elements, while the output vector would contain 64 elements.

SOURCE fftm.src

SEE ALSO **fftmi**, **fft**, **ffti**, **fftn**

fftn

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

FORMAT *y* **= fftn(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* L×M 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 33600×1 vector can compute as much as 20% faster than a 32768×1 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*M).

SEE ALSO **fft**, **ffti**, **fftm**, **fftmi**, **rfft**, **rffti**, **rfftip**, **rfftn**, **rfftnp**, **rfftp**

fgets

PURPOSE Reads a line of text from a file.

FORMAT *str* = **fgets**(*f*,*maxsize*);

INPUT

f	scalar, file handle of a file opened with **fopen**.
maxsize	scalar, maximum size of string to read in, including the terminating null byte.

OUTPUT

str	string.

REMARKS **fgets** reads text from a file into a string. It reads up to a newline, the end of the

file, or *maxsize*-1 characters. The result is placed in *str*, which is then terminated with a null byte. The newline, if present, is retained.

If the file is already at end-of-file when you call **fgets**, your program will terminate with an error. Use **eof** in conjunction with **fgets** to avoid this.

If the file was opened for update (see **fopen**) and you are switching from writing to reading, don't forget to call **fseek** or **fflush** first, to flush the file's buffer.

If you pass **fgets** the handle of a file opened with **open** (i.e., a data set or matrix file), your program will terminate with a fatal error.

SEE ALSO **fgetst**, **fgetsa**, **fopen**

fgetsa

PURPOSE Reads lines of text from a file into a string array.

FORMAT *sa* = **fgetsa**(*f*,*numl*);

INPUT
f scalar, file handle of a file opened with **fopen**.
numl scalar, number of lines to read.

OUTPUT
sa N×1 string array, N <= *numl*.

REMARKS **fgetsa** reads up to *numl* lines of text. If **fgetsa** reaches the end of the file before reading *numl* lines, *sa* will be shortened. Lines are read in the same manner as **fgets**, except that no limit is placed on the size of a line. Thus, **fgetsa** always returns complete lines of text. Newlines are retained. If *numl* is 1, **fgetsa** returns a string. (This is one way to read a line from a file without placing a limit on the length of the line.)

If the file is already at end-of-file when you call **fgetsa**, your program will

terminate with an error. Use **eof** in conjunction with **fgetsa** to avoid this. If the file was opened for update (see **fopen**) and you are switching from writing to reading, don't forget to call **fseek** or **fflush** first, to flush the file's buffer.

If you pass **fgetsa** the handle of a file opened with **open** (i.e., a data set or matrix file), your program will terminate with a fatal error.

SEE ALSO **fgetsat**, **fgets**, **fopen**

fgetsat

PURPOSE Reads lines of text from a file into a string array.

FORMAT *sa* = **fgetsat**(*f*,*numl*);

INPUT

f	scalar, file handle of a file opened with **fopen**.
numl	scalar, number of lines to read.

OUTPUT

sa	N×1 string array, N <= *numl*.

REMARKS **fgetsat** operates identically to **fgetsa**, except that newlines are not retained as text is read into *sa*.

In general, you don't want to use **fgetsat** on files opened in binary mode (see **fopen**). **fgetsat** drops the newlines, but it does NOT drop the carriage returns that precede them on some platforms. Printing out such a string array can produce unexpected results.

SEE ALSO **fgetsa**, **fgetst**, **fopen**

fgetst

PURPOSE Reads a line of text from a file.

FORMAT *str* = **fgetst**(*f*,*maxsize*);

INPUT
f scalar, file handle of a file opened with **fopen**.

maxsize scalar, maximum size of string to read in, including the null terminating byte.

OUTPUT
str string.

REMARKS **fgetst** operates identically to **fgets**, except that the newline is not retained in the string.

In general, you don't want to use **fgetst** on files opened in binary mode (see **fopen**). **fgetst** drops the newline, but it does NOT drop the preceding carriage return used on some platforms. Printing out such a string can produce unexpected results.

SEE ALSO **fgets**, **fgetsat**, **fopen**

fileinfo

PURPOSE Returns names and information for files that match a specification.

FORMAT { *fnames*,*finfo* } = **fileinfo**(*fspec*);

INPUT
fspec string, file specification. Can include path. Wildcards are allowed in *fspec*.

OUTPUT *fnames* N×1 string array of all file names that match, null string if none are found.

finfo N×13 matrix, information about matching files.

UNIX/Linux

[N, 1] filesystem ID
[N, 2] inode number
[N, 3] mode bit mask
[N, 4] number of links
[N, 5] user ID
[N, 6] group ID
[N, 7] device ID (char/block special files only)
[N, 8] size in bytes
[N, 9] last access time
[N,10] last data modification time
[N,11] last file status change time
[N,12] preferred I/O block size
[N,13] number of 512-byte blocks allocated

Windows

[N, 1] drive number (A = 0, B = 1, etc.)
[N, 2] n/a, 0
[N, 3] mode bit mask
[N, 4] number of links, always 1
[N, 5] n/a, 0
[N, 6] n/a, 0
[N, 7] n/a, 0
[N, 8] size in bytes
[N, 9] last access time
[N,10] last data modification time
[N,11] creation time
[N,12] n/a, 0
[N,13] n/a, 0

finfo will be a scalar zero if no matches are found.

REMARKS *fnames* will contain file names only; any path information that was passed is dropped.

The time stamp fields (*finfo*[N,9:11]) are expressed as the number of seconds since midnight, Jan. 1, 1970, Coordinated Universal Time (UTC).

SEE ALSO **filesa**

filesa

PURPOSE Returns a string array of file names.

FORMAT *y* = **filesa(***n***);**

INPUT *n* string, file specification to search for. Can include path. Wildcards are allowed in *n*.

OUTPUT *y* N×1 string array of all file names that match, or null string if none are found.

REMARKS *y* will contain file names only; any path information that was passed is dropped.

EXAMPLE
```
y = filesa("ch*");
```

In this example all files listed in the current directory that begin with "ch" will be returned.

```
proc exist(filename);
    retp(not filesa(filename) $=\,= "");
endp;
```

This procedure will return 1 if the file exists or 0 if not.

SEE ALSO **fileinfo**, **shell**

floor

PURPOSE Round down toward $-\infty$.

FORMAT *y* **= floor(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array containing the elements of *x* rounded down.

REMARKS This rounds every element in *x* down to the nearest integer.

EXAMPLE
```
x = 100*rndn(2,2);
```

```
x =  77.68  -14.10
      4.73  -158.88
```

```
f = floor(x);
```

```
f =  77.00  -15.00
      4.00  -159.00
```

SEE ALSO **ceil**, **round**, **trunc**

fmod

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

FORMAT r = **fmod**(x,y);

INPUT
- x N×K matrix.
- y L×M matrix, E×E conformable with x.

OUTPUT
- r max(N,L) by max(K,M) matrix.

REMARKS Returns the floating-point remainder r of x/y such that $x = iy + r$, where i is an integer, r has the same sign as x and $|r| < |y|$.

Compare this with %, the modulo division operator. (See Operators, Chapter 11.)

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
```

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**(*str*);

INPUT *str* string or character vector containing the names of fonts to be used in the plot. The folowing fonts are available:

Simplex	standard sans serif font.
Simgrma	Simplex greek, math.
Microb	bold and boxy.
Complex	standard font with serif.

REMARKS The first font specified will be used for the axes numbers.

If *str* is a null string, or **fonts** is not called, Simplex is loaded by default.

For more information on how to select fonts within a text string, see PUBLICATION QUALITY GRAPHICS, Chapter 24.

SOURCE pgraph.src

SEE ALSO **title**, **xlabel**, **ylabel**, **zlabel**

fopen

PURPOSE Opens a file.

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

INPUT *filename* string, name of file to open.
omode string, file I/O mode. (See Remarks, below.)

OUTPUT *f* scalar, file handle.

PORTABILITY **UNIX**

Carriage return-linefeed conversion for files opened in text mode is unnecessary, because in UNIX a newline is simply a linefeed.

REMARKS *filename* can contain a path specification.

omode is a sequence of characters that specify the mode in which to open the file. The first character must be one of:

r Open an existing file for reading. If the file does not exist, **fopen** fails.

w Open or create a file for writing. If the file already exists, its current contents will be destroyed.

a Open or create a file for appending. All output is appended to the end of the file.

To this can be appended a **+** and/or a **b**. The **+** indicates the file is to opened for reading and writing, or update, as follows:

r+ Open an existing file for update. You can read from or write to any location in the file. If the file does not exist, **fopen** fails.

w+ Open or create a file for update. You can read from or write to any location in the file. If the file already exists, its current contents will be destroyed.

a+ Open or create a file for update. You can read from any location in the file, but all output will be appended to the end of the file.

Finally, the **b** indicates whether the file is to be opened in text or binary mode. If the file is opened in binary mode, the contents of the file are read verbatim; likewise, anything output to the file is written verbatim. In text mode (the default), carriage return-linefeed sequences are converted on input to linefeeds, or newlines. Likewise on output, newlines are converted to carriage return-linefeeds. Also in text mode, if a CTRL-Z (char 26) is encountered during a read, it is interpreted as an end-of-file character, and reading ceases. In binary mode, CTRL-Z is read in uninterpreted.

The order of **+** and **b** is not significant; **rb+** and **r+b** mean the same thing.

You can both read from and write to a file opened for update. However, before switching from one to the other, you must make an **fseek** or **fflush** call, to flush the file's buffer.

If **fopen** fails, it returns a 0.

Use **close** and **closeall** to close files opened with **fopen**.

SEE ALSO **fseek**, **close**, **closeall**

for

PURPOSE Begins a **for** loop.

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

```
    .
    .
    .
endfor;
```

INPUT

i	literal, the name of the counter variable.
start	scalar expression, the initial value of the counter.
stop	scalar expression, the final value of the counter.
step	scalar expression, the increment value.

REMARKS The counter is strictly local to the loop. The expressions, *start*, *stop* and *step* are evaluated only once when the loop initializes. They are converted to integers and stored local to the loop.

The **for** loop is optimized for speed and much faster than a **do** loop.

The commands **break** and **continue** are supported. The **continue** command steps the counter and jumps to the top of the loop. The **break** command terminates the current loop.

The loop terminates when the value of *i* exceeds *stop*. If **break** is used to terminate the loop and you want the final value of the counter, you need to assign it to a variable before the **break** statement (see the third example, following).

EXAMPLE Example 1

```
x = zeros(10, 5);
for i (1, rows(x), 1);
    for j (1, cols(x), 1);
        x[i,j] = i*j;
    endfor;
endfor;
```

Example 2

```
x = rndn(3,3);
y = rndn(3,3);
for i (1, rows(x), 1);
    for j (1, cols(x), 1);
        if x[i,j] >= y[i,j];
            continue;
        endif;
        temp = x[i,j];
        x[i,j] = y[i,j];
        y[i,j] = temp;
    endfor;
endfor;
```

Example 3

```
li = 0;
x = rndn(100,1);
y = rndn(100,1);
for i (1, rows(x), 1);
    if x[i] /= y[i];
        li = i;
        break;
    endif;
endfor;
if li;
    print "Compare failed on row " li;
endif;
```

format

PURPOSE Controls the format of matrices and numbers printed out with **print** statements.

FORMAT **format** [[*/typ*]] [[*/fmted*]] [[*/mf*]] [[*/jnt*]] [[*f,p*]]

INPUT *typ* literal, symbol type flag(s). Indicate which symbol types you are setting the output format for.

/mat, /sa, /str Formatting parameters are maintained separately for matrices and arrays (**/mat**), 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**.

/fmted literal, enable formatting flag.

/on, /off Enable/disable formatting. When formatting is disabled, the contents of a variable are dumped to the screen in a "raw" format. **/off** is currently supported only for strings. "Raw" format for strings means that the entire string is printed, starting at the current cursor position. When formatting is enabled for strings, they are handled the same as string arrays. This shouldn't be too surprising, since a string is actually a 1×1 string array.

/mf literal, matrix row format flag.

/m0 no delimiters before or after rows when printing out matrices.

/m1 or **/mb1** print 1 carriage return/line feed pair before each row of a matrix with more than 1 row.

/m2 or **/mb2** print 2 carriage return/line feed pairs before each row of a matrix with more than 1 row.

/m3 or **/mb3** print "Row 1", "Row 2"... before each row of a matrix with more than one row.

/ma1 print 1 carriage return/line feed pair after each row of a matrix with more than 1 row.

/ma2 print 2 carriage return/line feed pairs after each row of a matrix with more than 1 row.

/a1	print 1 carriage return/line feed pair after each row of a matrix.
/a2	print 2 carriage return/line feed pairs after each row of a matrix.
/b1	print 1 carriage return/line feed pair before each row of a matrix.
/b2	print 2 carriage return/line feed pairs before each row of a matrix.
/b3	print "Row 1", "Row 2"... before each row of a matrix.

/jnt literal, matrix element format flag – controls justification, notation and trailing character.

Right-Justified

/rd	Signed decimal number in the form [[-]]####.####, where #### is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed.
/re	Signed number in the form [[-]]#.##E±###, where # is one decimal digit, ## is one or more decimal digits depending on the precision, and ### is three decimal digits. If precision is 0, the form will be [[-]]#E±### with no decimal point printed.
/ro	This will give a format like **/rd** or **/re** depending on which is most compact for the number being printed. A format like **/re** will be used only if the exponent value is less than -4 or greater than the precision. If a **/re** format is used, a decimal point will always appear. The precision signifies the number of significant digits displayed.
/rz	This will give a format like **/rd** or **/re** depending on which is most compact for the number being printed. A format like **/re** will be used only if the

exponent value is less than -4 or greater than the precision. If a **/re** format is used, trailing zeros will be supressed and a decimal point will appear only if one or more digits follow it. The precision signifies the number of significant digits displayed.

Left-Justified

/ld Signed decimal number in the form [[-]]####.####, where #### is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed. If the number is positive, a space character will replace the leading minus sign.

/le Signed number in the form [[-]]#.##E±###, where # is one decimal digit, ## is one or more decimal digits depending on the precision, and ### is three decimal digits. If precision is 0, the form will be [[-]]#E±### with no decimal point printed. If the number is positive, a space character will replace the leading minus sign.

/lo This will give a format like **/ld** or **/le** depending on which is most compact for the number being printed. A format like **/le** will be used only if the exponent value is less than -4 or greater than the precision. If a **/le** format is used, a decimal point will always appear. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.

/lz This will give a format like **/ld** or **/le** depending on which is most compact for the number being printed. A format like **/le** will be used only if the exponent value is less than -4 or greater than the precision. If a **/le** format is used, trailing zeros

will be supressed and a decimal point will appear only if one or more digits follow it. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.

Trailing Character

The following characters can be added to the */jnt* parameters above to control the trailing character if any:

```
format /rdn 1,3;
```

s	The number will be followed immediately by a space character. This is the default.
c	The number will be followed immediately by a comma.
t	The number will be followed immediately by a tab character.
n	No trailing character.

f scalar expression, controls the field width.

p scalar expression, controls the precision.

REMARKS If character elements are to be printed, the precision should be at least 8 or the elements will be truncated. This does not affect the string data type.

For numeric values in matrices, *p* sets the number of significant digits to be printed. For string arrays, strings, and character elements in matrices, *p* sets the number of characters to be printed. If a string is shorter than the specified precision, the entire string is printed. For string arrays and strings, *p* = -1 means print the entire string, regardless of its length. *p* = -1 is illegal for matrices; setting *p* >= 8 means the same thing for character elements.

The */xxx* slash parameters are optional. Field and precision are optional also, but if one is included, then both must be included.

Slash parameters, if present, must precede the field and precision parameters.

A **format** statement stays in effect until it is overridden by a new **format**

statement. The slash parameters may be used in a **print** statement to override the current default.

f and *p* may be any legal expressions that return scalars. Nonintegers will be truncated to integers.

The total width of field will be overridden if the number is too big to fit into the space allotted. For instance, **format /rds 1,0** can be used to print integers with a single space between them, regardless of the magnitudes of the integers.

Complex numbers are printed with the sign of the imaginary half separating them and an "i" appended to the imaginary half. Also, the field parameter refers to the width of field for each half of the number, so a complex number printed with a field of 8 will actually take (at least) 20 spaces to print. The character printed after the imaginary part can be changed (for example, to a "j") with the **sysstate** function, case 9.

The default when **GAUSS** is first started is:

```
format /mb1 /ros 16,8;
```

EXAMPLE This code:

```
x = rndn(3,3);

format /m1 /rd 16,8;
print x;
```

produces:

```
-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:

```
-1.635334648 1.613507002 -1.062951787
0.2617128159 0.2797229414 -1.389372421
0.5889111366 0.4681220206 1.088059602
```

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
```

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**, **output**

formatcv

PURPOSE Sets the character data format used by **printfmt**.

FORMAT *oldfmt* = **formatcv**(*newfmt*);

INPUT *newfmt* 1×3 vector, the new format specification.

OUTPUT *oldfmt* 1×3 vector, the old format specification.

REMARKS See **printfm** for details on the format vector.

EXAMPLE This example saves the old format, sets the format desired for printing *x*, prints *x*, then restores the old format. This code:

```
x = { A 1, B 2, C 3 };
oldfmt = formatcv("*.*s" ~ 3 ~ 3);
call printfmt(x,0~1);
call formatcv(oldfmt);
```

produces:

```
A               1
B               2
C               3
```

SOURCE gauss.src

GLOBALS **__fmtcv**

SEE ALSO **formatnv**, **printfm**, **printfmt**

formatnv

PURPOSE Sets the numeric data format used by **printfmt**.

FORMAT *oldfmt* = **formatnv(***newfmt***);**

INPUT *newfmt* 1×3 vector, the new format specification.

OUTPUT *oldfmt* 1×3 vector, the old format specification.

REMARKS See **printfm** for details on the format vector.

EXAMPLE This example saves the old format, sets the format desired for printing *x*, prints *x*, then restores the old format. This code:

```
x = { A 1, B 2, C 3 };
oldfmt = formatnv("*.*lf" ~ 8 ~ 4);
call printfmt(x,0~1);
call formatnv(oldfmt);
```

produces:

```
A   1.0000
B   2.0000
C   3.0000
```

SOURCE gauss.src

GLOBALS __fmtnv

SEE ALSO **formatcv**, **printfm**, **printfmt**

fputs

PURPOSE Writes strings to a file.

FORMAT *numl* = **fputs**(*f*,*sa*);

INPUT
- *f* scalar, file handle of a file opened with **fopen**.
- *sa* string or string array.

OUTPUT
- *numl* scalar, the number of lines written to the file.

PORTABILITY **UNIX**

Carriage return-linefeed conversion for files opened in text mode is unnecessary, because in UNIX a newline is simply a linefeed.

REMARKS **fputs** writes the contents of each string in *sa*, minus the null terminating byte, to the file specified. If the file was opened in text mode (see **fopen**), any newlines present in the strings are converted to carriage return-linefeed sequences on output. If *numl* is not equal to the number of elements in *sa*, there may have been an I/O error while writing the file. You can use **fcheckerr** or **fclearerr** to check this. If there was an error, you can call **fstrerror** to find out what it was. If the file was opened for update (see **fopen**) and you are switching from reading to writing, don't forget to call **fseek** or **fflush** first, to flush the file's buffer. If you pass **fputs** the handle of a file opened with **open** (i.e., a data set or matrix file), your program will terminate with a fatal error.

SEE ALSO **fputst**, **fopen**

fputst

PURPOSE Writes strings to a file.

FORMAT *numl* = **fputst** (*f*,*sa*);

INPUT

f	scalar, file handle of a file opened with **fopen**.
sa	string or string array.

OUTPUT

numl	scalar, the number of lines written to the file.

PORTABILITY **UNIX**

Carriage return-linefeed conversion for files opened in text mode is unnecessary, because in UNIX a newline is simply a linefeed.

REMARKS **fputst** works identically to **fputs**, except that a newline is appended to each string that is written to the file. If the file was opened in text mode (see **fopen**), these newlines are also converted to carriage return-linefeed sequences on output.

SEE ALSO **fputs**, **fopen**

fseek

PURPOSE Positions the file pointer in a file.

FORMAT *ret* = **fseek**(*f*,*offs*,*base*);

INPUT
- *f* scalar, file handle of a file opened with **fopen**.
- *offs* scalar, offset (in bytes).
- *base* scalar, base position.
  - **0** beginning of file.
  - **1** current position of file pointer.
  - **2** end of file.

OUTPUT *ret* scalar, 0 if successful, 1 if not.

PORTABILITY **UNIX**

Carriage return-linefeed conversion for files opened in text mode is unnecessary, because in UNIX a newline is simply a linefeed.

REMARKS **fseek** moves the file pointer *offs* bytes from the specified *base* position. *offs* can be positive or negative. The call may fail if the file buffer needs to be flushed (see **fflush**).

If **fseek** fails, you can call **fstrerror** to find out why.

For files opened for update (see **fopen**), the next operation can be a read or a write.

fseek is not reliable when used on files opened in text mode (see **fopen**). This has to do with the conversion of carriage return-linefeed sequences to newlines. In particular, an **fseek** that follows one of the **fget***xxx* or **fput***xxx* commands may not produce the expected result. For example:

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

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

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

If you pass **fseek** the handle of a file opened with **open** (i.e., a data set or matrix file), your program will terminate with a fatal error.

SEE ALSO **fopen**

fstrerror

PURPOSE Returns an error message explaining the cause of the most recent file I/O error.

FORMAT *s* **= fstrerror;**

OUTPUT *s* string, error message.

REMARKS Any time an I/O error occurs on a file opened with **fopen**, an internal error flag is updated. (This flag, unlike those accessed by **fcheckerr** and **fclearerr**, is not specific to a given file; rather, it is system-wide.) **fstrerror** returns an error message based on the value of this flag, clearing it in the process. If no error has occurred, a null string is returned.

Since **fstrerror** clears the error flag, if you call it twice in a row, it will always return a null string the second time.

The Windows system command called by **ftell** does not set the internal error flag accessed by **fstrerror**. Therefore, calling **fstrerror** after **ftell** on Windows will not produce any error information.

SEE ALSO **fopen**, **ftell**

ftell

PURPOSE Gets the position of the file pointer in a file.

FORMAT *pos* = **ftell**(*f*);

INPUT *f* scalar, file handle of a file opened with **fopen**.

OUTPUT *pos* scalar, current position of the file pointer in a file.

REMARKS **ftell** returns the position of the file pointer in terms of bytes from the beginning of the file. The call may fail if the file buffer needs to be flushed (see **fflush**).

If an error occurs, **ftell** returns -1. You can call **fstrerror** to find out what the error was.

If you pass **ftell** the handle of a file opened with **open** (i.e., a data set or matrix file), your program will terminate with a fatal error.

SEE ALSO **fopen**, **fseek**

ftocv

PURPOSE Converts a matrix containing floating point numbers into a matrix containing the decimal character representation of each element.

FORMAT *y* = **ftocv**(*x*,*field*,*prec*);

INPUT

x	N×K 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	N×K matrix containing the decimal character equivalent of the corresponding elements in *x* in the format defined by *field* and *prec*.

REMARKS If a number is narrower than *field*, it will be padded on the left with zeros.

If *prec* = 0, the decimal point will be suppressed.

EXAMPLE

```
y = seqa(6,1,5);
x = 0 $+ "cat" $+ ftocv(y,2,0);
```

```
      cat06
      cat07
x =   cat08
      cat09
      cat10
```

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**

ftos

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

FORMAT *y* **= ftos(***x*,*fmat*,*field*,*prec***);**

INPUT

x	scalar, the number to be converted.
fmat	string, the format string to control the conversion.
field	scalar or 2×1 vector, the minimum field width. If *field* is 2×1, it specifies separate field widths for the real and imaginary parts of *x*.
prec	scalar or 2×1 vector, the number of places following the decimal point. If *prec* is 2×1, it specifies separate precisions for the real and imaginary parts of *x*.

OUTPUT

y	string containing the decimal character equivalent of *x* in the format specified.

REMARKS The format string corresponds to the **format /***jnt* (justification, notation, trailing character) slash parameter as follows:

```
/rdn    ''%*.*lf''
/ren    ''%*.*lE''
/ron    ''%#*.*lG''
/rzn    ''%*.*lG''

/ldn    ''%- *.*lf''
/len    ''%- *.*lE''
/lon    ''%-# *.*lG''
/lzn    ''%- *.*lG''
```

If *x* is complex, you can specify separate formats for the real and imaginary parts by putting two format specifications in the format string. You can also specify separate fields and precisions. You can position the sign of the imaginary part by placing a "+" between the two format specifications. If you use two formats, no "`i`" is appended to the imaginary part. This is so you can use an alternate format if you prefer, for example, prefacing the imaginary part with a "`j`".

The format string can be a maximum of 80 characters.

If you want special characters to be printed after *x*, include them as the last characters of the format string. For example:

''%*.*lf,''	right-justified decimal followed by a comma.
''%-*.*s ''	left-justified string followed by a space.
''%*.*lf''	right-justified decimal followed by nothing.

You can embed the format specification in the middle of other text:

```
"Time: %*.*lf seconds."
```

If you want the beginning of the field padded with zeros, then put a "**0**" before the first "*****" in the format string:

''%0*.*lf''	right-justified decimal.

If *prec* = 0, the decimal point will be suppressed.

EXAMPLE You can create custom formats for complex numbers with **ftos**. For example,

```
let c = 24.56124+6.3224e-2i;

field = 1;
prec = 3|5;
fmat = "%lf + j%le is a complex number.";
cc = ftos(c,fmat,field,prec);
```

results in

```
cc = "24.561 + j6.32240e-02 is a complex number."
```

Some other things you can do with **ftos**:

```
let x = 929.857435324123;
let y = 5.46;
let z = 5;

field = 1;
prec = 0;
fmat = "%*.*lf";
zz = ftos(z,fmat,field,prec);

field = 1;
prec = 10;
fmat = "%*.*lE";
xx = ftos(x,fmat,field,prec);

field = 7;
prec = 2;
```

```
fmat = "%*.*lf seconds";
s1 = ftos(x,fmat,field,prec);
s2 = ftos(y,fmat,field,prec);

field = 1;
prec = 2;
fmat = "The maximum resistance is %*.*lf ohms.";
om = ftos(x,fmat,field,prec);
```

The results:

```
zz  = "5"
xx  = "9.2985743532E+02"
s1  = " 929.86 seconds"
s2  = " 5.46 seconds"
om  = "The maximum resistance is 929.86 ohms."
```

SEE ALSO **ftocv**, **stof**, **format**

ftostrC

PURPOSE Converts a matrix to a string array using a C language format specification.

FORMAT *sa* **= ftostrC(***x***,***fmt***);**

INPUT

x	N×K matrix, real or complex.
fmt	K×1, 1×K or 1×1 string array containing format information.

OUTPUT

sa	N×K string array.

REMARKS If *fmt* has K elements, each column of *sa* can be formatted separately. If *x* is complex, there must be two format specifications in each element of *fmt*.

EXAMPLE

```
declare string fmtr = {
"%6.3lf",
"%11.8lf"
};

declare string fmtc = {
"(%6.3lf, %6.3lf)",
"(%11.8lf, %11.8lf)"
};

xr = rndn(4, 2);
xc = sqrt(xr')';

sar = ftostrC(xr, fmtr);
sac = ftostrC(xc, fmtc);

print sar;
print sac;
```

produces:

```
          -0.166       1.05565441
          -1.590      -0.79283296
           0.130      -1.84886957
           0.789       0.86089687

( 0.000, -0.407) ( 1.02745044,  0.00000000)
( 0.000, -1.261) ( 0.00000000, -0.89041168)
( 0.361,  0.000) ( 0.00000000, -1.35973143)
( 0.888,  0.000) ( 0.92784529,  0.00000000)
```

SEE ALSO **strtof**, **strtofcplx**

gamma

PURPOSE Returns the value of the gamma function.

FORMAT *y* **= gamma(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array.

REMARKS For each element of *x* this function returns the integral

$$\int_0^\infty t^{(x-1)}e^{-t}dt$$

All elements of *x* must be positive and less than or equal to 169. Values of *x* greater than 169 will cause an overflow.

The natural log of **gamma** is often what is required and it can be computed without the overflow problems of **gamma** using **lnfact**.

EXAMPLE
```
y = gamma(2.5);
```

```
y = 1.32934
```

SEE ALSO **cdfchic**, **cdfbeta**, **cdffc**, **cdfn**, **cdfnc**, **cdftc**, **erf**, **erfc**, **lnfact**

gammaii

PURPOSE Computes the inverse incomplete gamma function.

FORMAT *x* = **gammaii**(*a*,*p*);

INPUT
a M×N matrix, exponents.
p K×L matrix, E×E conformable with *a*, incomplete gamma values.

OUTPUT *x* max(M,K) by max(N,L) matrix, abscissae.

SOURCE cdfchii.src

GLOBALS **_ginvinc**, **__macheps**

gausset

PURPOSE Resets the global control variables declared in gauss.dec.

FORMAT **gausset**;

SOURCE gauss.src

GLOBALS **__altnam**, **__con**, **__ff**, **__fmtcv**, **__fmtnv**, **__header**, **__miss**, **__output**, **__row**, **__rowfac**, **__sort**, **__title**, **__tol**, **__vpad**, **__vtype**, **__weight**

gdaAppend

PURPOSE Appends data to a variable in a **GAUSS** Data Archive.

FORMAT *ret* = **gdaAppend**(*filename*,*x*,*varname*);

INPUT

filename string, name of data file.

x matrix, array, string or string array, data to append.

varname string, variable name.

OUTPUT

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1 Null file name.
2 File open error.
3 File write error.
4 File read error.
5 Invalid data file type.
8 Variable not found.
10 File contains no variables.
14 File too large to be read on current platform.
17 Type mismatch.
18 Argument wrong size.
19 Data must be real.
20 Data must be complex.

REMARKS

This command appends the data contained in *x* to the variable *varname* in *filename*. Both *x* and the variable referenced by *varname* must be the same data type, and they must both contain the same number of columns.

Because **gdaAppend** increases the size of the variable, it moves the variable to just after the last variable in the data file to make room for the added data, leaving empty bytes in the variable's old location. It also moves the variable descriptor table, so it is not overwritten by the variable data. This does not change the index of the variable because variable indices are determined NOT by the order of the variable data in a GDA, but by the order of the variable descriptors. Call **gdaPack** to pack the data in a GDA, so it contains no empty bytes.

EXAMPLE

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");
```

```
y = rndn(25,50);
ret = gdaAppend("myfile.gda",y,"x1");
```

This example adds 25*50=1250 elements to **x1**, making it a 125×50 matrix.

SEE ALSO **gdaWriteSome**, **gdaUpdate**, **gdaWrite**

gdaCreate

PURPOSE Creates a **GAUSS** Data Archive.

FORMAT *ret* = **gdaCreate**(*filename*,*overwrite*);

INPUT *filename* string, name of data file to create.

overwrite scalar, one of the following:

0 error out if file already exists.

1 overwrite file if it already exists.

OUTPUT *ret* scalar, return code, 0 if successful, otherwise one of the following error codes:

1 Null file name.

3 File write error.

6 File already exists.

7 Cannot create file.

REMARKS This command creates a **GAUSS** Data Archive containing only a header. To add data to the GDA, call **gdaWrite**.

It is recommended that you include a `.gda` extension in *filename*. However, **gdaCreate** will not force an extension.

EXAMPLE

```
ret = gdaCreate("myfile.gda",1);
```

SEE ALSO **gdaWrite**

gdaDStat

PURPOSE Computes descriptive statistics on multiple N×1 variables in a **GAUSS** Data Archive.

FORMAT *dout* = **gdaDStat(***dc0*,*filename*,*vars*);

INPUT *dc0* an instance of a **dstatmtControl** structure with the following members:

dc0.altnames	K×1 string array of alternate variable names for the output. Default = "".
dc0.maxbytes	scalar, the maximum number of bytes to be read per iteration of the read loop. Default = 1e9.
dc0.maxvec	scalar, the largest number of elements allowed in any one matrix. Default = 20000.
dc0.miss	scalar, one of the following: **0** There are no missing values (fastest). **1** Listwise deletion, drop a row if any missings occur in it. **2** Pairwise deletion. Default = 0.
dc0.output	scalar, one of the following: **0** Do not print output table. **1** Print output table. Default = 1.
dc0.row	scalar, the number of rows of *var* to be read per iteration of the read loop. If 0, (default) the number of rows will be calculated using *dc0*.**maxbytes** and *dc0*.**maxvec**.

filename string, name of data file.

vars K×1 string array, names of variables

- or -

K×1 vector, indices of variables.

OUTPUT *dout* an instance of a **dstatmtOut** structure with the following members:

dout.vnames	K×1 string array, the names of the variables used in the statistics.
dout.mean	K×1 vector, means.
dout.var	K×1 vector, variance.
dout.std	K×1 vector, standard deviation.
dout.min	K×1 vector, minima.
dout.max	K×1 vector, maxima.
dout.valid	K×1 vector, the number of valid cases.
dout.missing	K×1 vector, the number of missing cases.
dout.errcode	scalar, error code, 0 if successful, or one of the following: **1** No GDA indicated. **4** Not implemented for complex data. **5** Variable must be type matrix. **6** Too many variables specified. **7** Too many missings - no data left after packing. **8** Name variable wrong size. **9** **altnames** member of **dstatmtControl** structure wrong size. **11** Data read error.

REMARKS The variables referenced by *vars* must all be N×1.

The names of the variables in the GDA will be used for the output by default. To use alternate names, set the **altnames** member of the **dstatmtControl** structure.

If pairwise deletion is used, the minima and maxima will be the true values for the valid data. The means and standard deviations will be computed using the correct number of valid observations for each variable.

EXAMPLE

```
struct dstatmtControl dc0;
struct dstatmtOut dout;

dc0 = dstatmtControlCreate;
vars = { 1,4,5,8 };
dout = gdaDStat(dc0,"myfile.gda",vars);
```

This example computes descriptive statistics on the first, fourth, fifth and eighth variables in `myfile.gda`.

SOURCE `gdadstat.src`

SEE ALSO **gdaDStatMat**, **dstatmtControlCreate**

gdaDStatMat

PURPOSE Computes descriptive statistics on a selection of columns in a variable in a **GAUSS** Data Archive.

FORMAT *dout* **= gdaDStatMat(***dc0*,*filename*,*var*,*colind*,*vnamevar***);**

INPUT *dc0* an instance of a **dstatmtControl** structure with the following members:

dc0`.altnames`	K×1 string array of alternate variable names for the output. Default = "".
dc0`.maxbytes`	scalar, the maximum number of bytes to be read per iteration of the read loop. Default = 1e9.

dc0.maxvec — scalar, the largest number of elements allowed in any one matrix. Default = 20000.

dc0.miss — scalar, one of the following:

0 There are no missing values (fastest).

1 Listwise deletion, drop a row if any missings occur in it.

2 Pairwise deletion.

Default = 0.

dc0.output — scalar, one of the following:

0 Do not print output table.

1 Print output table.

Default = 1.

dc0.row — scalar, the number of rows of *var* to be read per iteration of the read loop.
If 0, (default) the number of rows will be calculated using *dc0*.**maxbytes** and *dc0*.**maxvec**.

filename string, name of data file.

var string, name of variable

- or -

scalar, index of variable.

colind K×1 vector, indices of columns in variable to use.

vnamevar string, name of variable containing names for output

- or -

scalar, index of variable containing names for output.

OUTPUT *dc0* an instance of a **dstatmtOut** structure with the following members:

dout.vnames — K×1 string array, the names of the variables used in the statistics.

dout.mean — K×1 vector, means.

dout.var — K×1 vector, variance.

dout.std — K×1 vector, standard deviation.

dout.min — K×1 vector, minima.

dout.max	K×1 vector, maxima.
dout.valid	K×1 vector, the number of valid cases.
dout.missing	K×1 vector, the number of missing cases.
dout.errcode	scalar, error code, 0 if successful, otherwise one of the following:

1 No GDA indicated.
3 Variable must be N×1.
4 Not implemented for complex data.
5 Variable must be type matrix.
7 Too many missings - no data left after packing.
9 **altnames** member of **dstatmtControl** structure wrong size.
11 Data read error.

REMARKS Set *colind* to a scalar 0 to use all of the columns in *var*.

vnamevar must either reference an M×1 string array variable containing variable names, where M is the number of columns in the data set variable, or be set to a scalar 0. If *vnamevar* references an M×1 string array variable, then only the elements indicated by *colind* will be used. Otherwise, if *vnamevar* is set to a scalar 0, then the variable names for the output will be generated automatically ("X1,X2,...,XK") unless the alternate variable names are set explicitly in the **altnames** member of the **dstatmtControl** structure.

If pairwise deletion is used, the minima and maxima will be the true values for the valid data. The means and standard deviations will be computed using the correct number of valid observations for each variable.

EXAMPLE

```
struct dstatmtControl dc0;
struct dstatmtOut dout;

dc0 = dstatmtControlCreate;
var = 3;
index = { 1,3,4,7 };
dout = gdaDStatMat(dc0,"myfile.gda",var,index,"");
```

This example computes descriptive statistics on the first, third, fourth and seventh columns of the third variable in `myfile.gda`, generating names for the output automatically.

SOURCE `gdadstat.src`

SEE ALSO **gdaDStat**, **dstatmtControlCreate**

gdaGetIndex

PURPOSE Gets the index of a variable in a **GAUSS** Data Archive.

FORMAT *ind* = **gdaGetIndex**(*filename*,*varname*);

INPUT
filename string, name of data file.
varname string, name of variable in the GDA.

OUTPUT
ind scalar, index of variable in the GDA.

REMARKS If **gdaGetIndex** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1 Null file name.
2 File open error.
4 File read error.
5 Invalid file type.
8 Variable not found.
10 File contains no variables.
14 File too large to be read on current platform.

EXAMPLE
```
ind = gdaGetIndex("myfile.gda","observed");
```

SEE ALSO **gdaGetName**, **gdaReadByIndex**

gdaGetName

PURPOSE Gets the name of a variable in a **GAUSS** Data Archive.

FORMAT *varname* = **gdaGetName**(*filename*,*varind*);

INPUT *filename* string, name of data file.
varind scalar, index of variable in the GDA.

OUTPUT *varname* string, name of variable in the GDA.

REMARKS If **gdaGetName** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1 Null file name.
2 File open error.
4 File read error.
5 Invalid file type.
8 Variable not found.

EXAMPLE
```
varname = gdaGetName("myfile.gda",5);
```

SEE ALSO **gdaGetIndex**, **gdaRead**, **gdaGetNames**

gdaGetNames

PURPOSE Gets the names of all the variables in a **GAUSS** Data Archive.

FORMAT *varnames* = **gdaGetNames**(*filename*);

INPUT *filename* string, name of data file.

OUTPUT *varnames* N×1 string array, names of all the variables in the GDA.

REMARKS If **gdaGetNames** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
10	File contains no variables.
13	Result too large for current platform.
14	File too large to be read on current platform.

EXAMPLE
```
varnames = gdaGetNames("myfile.gda");
```

SEE ALSO **gdaGetTypes**, **gdaGetName**

gdaGetOrders

PURPOSE Gets the orders of a variable in a **GAUSS** Data Archive.

FORMAT *ord* = **gdaGetOrders**(*filename*,*varname*);

INPUT *filename* string, name of data file.

varname string, name of variable in the GDA.

OUTPUT *ord* M×1 vector, orders of the variable in the GDA.

REMARKS If the specified variable is a matrix or string array, then *ord* will be a 2×1 vector containing the rows and columns of the variable respectively. If the variable is a string, then *ord* will be a scalar containing the length of the string. If the variable is an N-dimensional array, then *ord* will be an N×1 vector containing the sizes of each dimension.

If **gdaGetOrders** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1 Null file name.
2 File open error.
4 File read error.
5 Invalid file type.
8 Variable not found.
10 File contains no variables.
14 File too large to be read on current platform.

EXAMPLE

```
ord = gdaGetOrders("myfile.gda","x5");
```

SEE ALSO **gdaGetName**, **gdaGetIndex**

gdaGetType

PURPOSE Gets the type of a variable in a **GAUSS** Data Archive.

FORMAT *vartype* **= gdaGetType(***filename***,***varname***);**

INPUT *filename* string, name of data file.
varname string, name of variable in the GDA.

OUTPUT *vartype* scalar, type of the variable in the GDA.

REMARKS *vartype* may contain any of the following:

6 Matrix
13 String
15 String array
21 Array

If **gdaGetType** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1 Null file name.
2 File open error.
4 File read error.
5 Invalid file type.
8 Variable not found.
10 File contains no variables.
14 File too large to be read on current platform.

EXAMPLE
```
vartype = gdaGetType("myfile.gda","x1");
```

SEE ALSO **gdaGetTypes**

gdaGetTypes

PURPOSE Gets the types of all the variables in a **GAUSS** Data Archive.

FORMAT *vartypes* **= gdaGetTypes(***filename***);**

INPUT *filename* string, name of data file.

OUTPUT *vartypes* N×1 vector, types of all the variables in the GDA.

REMARKS *vartypes* may contain any of the following:

6 Matrix
13 String
15 String array
21 Array

If **gdaGetTypes** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. Valid error codes for this command include:

1 Null file name.
2 File open error.
4 File read error.
5 Invalid file type.
10 File contains no variables.
14 File too large to be read on current platform.

EXAMPLE
```
vartypes = gdaGetTypes("myfile.gda");
```

SEE ALSO **gdaGetNames**, **gdaRead**

gdaGetVarInfo

PURPOSE Gets information about all of the variables in a **GAUSS** Data Archive and returns it in an array of **gdavartable** structures.

INCLUDE `gdafns.sdf`

FORMAT *vtab* **= gdaGetVarInfo(***filename***);**

INPUT *filename* string, name of data file.

OUTPUT *vtab* N×1 array of **gdavartable** structures, where N is the number of variables in *filename*, containing the following members:

vtab[*i*]`.name`	string, name of variable.
vtab[*i*]`.type`	scalar, type of variable.
vtab[*i*]`.orders`	M×1 vector or scalar, orders of the variable.

REMARKS The size of *vtab*.**orders** is dependent on the type of the variable as follows:

Variable Type	*vtab*.**orders**
array	M×1 vector, where M is the number of dimensions in the array, containing the sizes of each dimension, from the slowest-moving dimension to the fastest-moving dimension.
matrix	2×1 vector containing the rows and columns of the matrix, respectively.
string	scalar containing the length of string, excluding the null terminating byte.
string array	2×1 vector containing the rows and columns of the string array, respectively.

vtab.**type** may contain any of the following:

6 matrix
13 string
15 string array
21 array

EXAMPLE

```
#include gdafns.sdf

struct gdavartable vtab;
vtab = gdaGetVarInfo("myfile.gda");
```

SOURCE gdafns.src

SEE ALSO **gdaReportVarInfo**, **gdaGetNames**, **gdaGetTypes**, **gdaGetOrders**

gdaIsCplx

PURPOSE Checks to see if a variable in a **GAUSS** Data Archive is complex.

FORMAT *y* **= gdaIsCplx(***filename***,***varname***);**

INPUT *filename* string, name of data file.

varname string, name of variable in the GDA.

OUTPUT *y* scalar, 1 if variable is complex; 0 if real.

REMARKS If **gdaIsCplx** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. Valid error codes for this command include:

1 Null file name.
2 File open error.
4 File read error.
5 Invalid file type.
8 Variable not found.
10 File contains no variables.
14 File too large to be read on current platform.

EXAMPLE
```
cplx = gdaIsCplx("myfile.gda","x1");
```

gdaLoad

PURPOSE Loads variables in a GDA into the workspace.

FORMAT *ret* = **gdaLoad**(*filename*,*create*,*modify*,*rename*,*ftypes*,*errh*,*report*);

INPUT *filename* string, name of data file.

create scalar, create flag:

0 do not create any new variables in the workspace.
1 create new variables in the workspace.

modify scalar, modify flag:

0 do not modify any variables in the workspace.
1 if the name of a variable in the data file matches the name of a variable already in the workspace, modify that variable.

rename scalar, rename flag:

0 do not rename a variable retrieved from the data file when copying it into the workspace.

1 rename variables retrieved from the data file when copying them into the workspace if there are name conflicts with existing variables, which may not be modified.

ftypes scalar, type force flag:

0 do not force a type change on any variables in the workspace when modifying.

1 force a type change on a variable in the workspace when modifying it with the data in a variable of the same name in the data file. Note that if *ftypes* is set to 1, **gdaLoad** will follow regular type change rules. The types of sparse matrix and structure variables will NOT be changed.

errh scalar, controls the error handling of **gdaLoad**:

0 skip operations that cannot be performed, without setting an error return.

1 return an error code if operations are skipped.

2 terminate program if operations are skipped.

report scalar, controls reporting:

0 no reporting.

1 report only name changes and operations that could not be performed.

2 report type changes, name changes, and operations that could not be performed.

3 report everything.

OUTPUT *ret* scalar, return code, 0 if successful, otherwise one of the following error codes:

4 File read error.

5 Invalid file type.

10 File contains no variables.

14 File too large to be read on current platform.

24 Variables skipped.

26 Cannot add structure definition.

27 Structure definition does not match.

REMARKS For each variable in *filename*, **gdaLoad** will first compare the name of the variable against the names of the variables already resident in the **GAUSS** workspace to see if there is a match. If there is not a match, and *create* is set to 1, it will create a new variable. Otherwise if *create* is set to 0, it will skip that variable.

If the variable name does match that of a variable already resident in the **GAUSS** workspace, and *modify* is set to 1, it will attempt to modify that variable. If the types of the two variables are different, and *ftype* is set to 1, it will force the type change if possible and modify the existing variable.

If it cannot modify the variable or *modify* is set to 0, it will check to see if *rename* is set to 1, and if so, attempt to rename the variable, appending an *_num* to the variable name, beginning with *num* = 1 and counting upward until it finds a name with which there are no conflicts. If the variable cannot be modified and *rename* is set to 0, then the variable will be skipped.

The *rename* argument also controls the handling of structure definitions. If a structure variable is encountered in the GDA file, and no variable of the same name exists in the workspace (or the variable is renamed), **gdaLoad** will attempt to find a structure definition in the workspace that matches the one in the GDA. Note that in order for structure definitions to match, the structure definition names must be the same as well as the number, order, names, and types of their members.

If no matching structure definition is found, the definition in the file will be loaded into the workspace. If there is already a non-matching structure definition with the same name in the workspace and *rename* is set to 1, then **gdaLoad** will attempt to rename the structure definition, using the same method as it does for variable names.

If a structure variable is encountered in the GDA file, a structure variable of the same name already exists in the workspace, and *modify* is set to 1, then **gdaLoad** will modify the existing variable, providing that the structure definitions of the two variables match.

EXAMPLE

```
ret = gdaLoad("myfile.gda",1,1,1,1,1,3);
```

This example loads the variables in `myfile.gda` into the workspace, creating a new variable if a variable of the same name does not already exist, modifying an existing variable if a variable of the same name does already exist and the modification does not result in an impossible type change, and renaming the variable if none of the above is possible. The example returns an error code if any variables in `myfile.gda` are skipped and reports all activity.

SEE ALSO **gdaSave**

gdaPack

PURPOSE Packs the data in a **GAUSS** Data Archive, removing all empty bytes and truncating the file.

FORMAT *ret* = **gdaPack**(*filename*);

INPUT *filename* string, name of data file.

OUTPUT *ret* scalar, return code, 0 if successful, otherwise one of the following error codes:

1 Null file name.
2 File open error.
3 File write error.
4 File read error.
5 Invalid data file type.
10 File contains no variables.
12 File truncate error.
14 File too large to be read on current platform.

REMARKS You may want to call **gdaPack** after several calls to **gdaUpdate** to remove all of the empty bytes from a GDA.

EXAMPLE
```
ret = gdaPack("myfile.gda");
```

SEE ALSO **gdaUpdate**, **gdaWrite**

gdaRead

PURPOSE Gets a variable from a **GAUSS** Data Archive.

FORMAT *y* **= gdaRead(***filename***,***varname***);**

INPUT
filename string, name of data file.

varname string, name of variable in the GDA.

OUTPUT
y matrix, array, string or string array, variable data.

REMARKS If **gdaRead** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

EXAMPLE
```
y = gdaRead("myfile.gda","x1");
```

SEE ALSO **gdaReadByIndex**, **gdaGetName**

gdaReadByIndex

PURPOSE Gets a variable from a **GAUSS** Data Archive given a variable index.

FORMAT *y* = **gdaReadByIndex**(*filename*,*varind*);

INPUT *filename* string, name of data file.

varind scalar, index of variable in the GDA.

OUTPUT *y* matrix, array, string or string array, variable data.

REMARKS If **gdaReadByIndex** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1 Null file name.
2 File open error.
4 File read error.
5 Invalid file type.
8 Variable not found.
10 File contains no variables.

EXAMPLE
```
y = gdaReadByIndex("myfile.gda",3);
```

SEE ALSO **gdaRead**, **gdaGetIndex**

gdaReadSome

PURPOSE Reads part of a variable from a **GAUSS** Data Archive.

FORMAT *y* = **gdaReadSome**(*filename*,*varname*,*index*,*orders*);

INPUT

filename string, name of data file.

varname string, name of variable in the GDA.

index scalar or N×1 vector, index into variable where read is to begin.

orders scalar or K×1 vector, orders of object to output.

OUTPUT

y matrix, array, string or string array, variable data.

REMARKS

This command reads part of the variable *varname* in *filename*, beginning at the position indicated by *index*. The *orders* argument determines the size and shape of the object outputted by **gdaReadSome**. The number of elements read equals the product of all of the elements in *orders*.

If *index* is a scalar, it will be interpreted as the $index^{th}$ element of the variable. Thus if *varname* references a 10×5 matrix, an *index* of 42 would indicate the 42^{nd} element, which is equivalent to the [8,2] element of the matrix (remember that **GAUSS** matrices are stored in row major order). If *index* is an N×1 vector, then N must equal the number of dimensions in the variable referenced by *varname*.

If *orders* is a K×1 vector, then *y* will be a K-dimensional object. If *orders* is a scalar r, then *y* will be an r×1 column vector. To specify a 1×r row vector, set *output* = { 1, r }.

If the variable referenced by *varname* is numeric (a matrix or array) and *orders* is a scalar or 2×1 vector, then *y* will of type matrix. If the variable is numeric and *orders* is an N×1 vector where N>2, then *y* will be of type array.

If *varname* references a string, then both *index* and *orders* must be scalars, and *index* must contain an index into the string in characters.

If **gdaReadSome** fails, it will return a scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1 Null file name.
2 File open error.
4 File read error.
5 Invalid file type.
8 Variable not found.
10 File contains no variables.
13 Result too large for current platform.
14 File too large to be read on current platform.
15 Argument out of range.
18 Argument wrong size.

EXAMPLE

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");

index = { 35,20 };
orders = { 25,5 };
y = gdaReadSome("myfile.gda","x1",index,orders);
```

This example reads 25*5=125 elements from **x1**, beginning with the [35,20] element. The 125 elements are returned as a 25×5 matrix, **y**.

SEE ALSO **gdaWriteSome**, **gdaRead**

gdaReadSparse

PURPOSE Gets a sparse matrix from a **GAUSS** Data Archive.

FORMAT *sm* **= gdaReadSparse(***filename***,***varname***);**

INPUT
filename string, name of data file.
varname string, name of sparse matrix variable in the GDA.

OUTPUT
sm sparse matrix.

REMARKS If **gdaReadSparse** fails, it will return a sparse scalar error code. Call **scalerr** to get the value of the error code. The error code may be any of the following:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.
8	Variable not found.
10	File contains no variables.
14	File too large to be read on current platform.

EXAMPLE
```
sparse matrix sm1;
sm1 = gdaReadSparse("myfile.gda","sm");
```

SEE ALSO **gdaRead**, **gdaReadStruct**, **gdaWrite**

gdaReadStruct

PURPOSE Gets a structure from a **GAUSS** Data Archive.

FORMAT { *instance*, *retcode* } = **gdaReadStruct**(*filename*, *varname*, *structure_type*);

INPUT *filename* string, name of data file.

varname string, name of structure instance in the GDA.

structure_type string, structure type.

OUTPUT *instance* instance of the structure.

retcode scalar, 0 if successful, otherwise, any of the following error codes:

1	Null file name.
2	File open error.
4	File read error.
5	Invalid file type.

8 Variable not found.

10 File contains no variables.

14 File too large to be read on current platform.

REMARKS *instance* can be an array of structures.

EXAMPLE
```
struct mystruct {
    matrix x;
    array a;
};

struct mystruct msw;
msw.x = rndn(500,25);
msw.a = areshape(rndn(5000,100),10|500|100);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",msw,"ms");

struct mystruct msr;
{ msr, ret } = gdaReadStruct("myfile.gda","ms","mystruct");
```

SEE ALSO **gdaRead**, **gdaReadSparse**, **gdaWrite**

gdaReportVarInfo

PURPOSE Gets information about all of the variables in a **GAUSS** Data Archive and returns it in a string array formatted for printing.

FORMAT *vinfo* **= gdaReportVarInfo(***filename***);**

INPUT *filename* string, name of data file.

OUTPUT *vinfo* N×1 string array containing variable information.

REMARKS If you just want to print the information to the window, call **gdaReportVarInfo** without assigning the output to a symbol name:

```
gdaReportVarInfo(filename);
```

EXAMPLE
```
x1 = rndn(100,50);
x2 = rndn(75,5);
a = areshape(rndn(10000,1),10|100|10);
fname = "myfile.gda";
ret = gdaCreate(fname,1);
ret = gdaWrite(fname,x1,"x1");
ret = gdaWrite(fname,x2,"x2");
ret = gdaWrite(fname,a,"a1");
gdaReportVarInfo(fname);
```

produces:

```
Index    Name    Type            Orders
1        x1      matrix          100x50
2        x2      matrix          75x5
3        a1      array           10x100x10
```

SOURCE gdafns.src

SEE ALSO **gdaGetVarInfo**, **gdaGetNames**, **gdaGetTypes**, **gdaGetOrders**

gdaSave

PURPOSE Writes variables in a workspace to a GDA.

FORMAT *ret* = **gdaSave**(*filename*,*varnames*,*exclude*,*overwrite*,*report*);

INPUT

filename string, name of data file.

varnames string or N×K string array, names of variables in the workspace to include or exclude.

exclude scalar, include/exclude flag:

0 include all variables contained in *varnames*.

1 exclude all variables contained in *varnames*.

overwrite scalar, controls the overwriting of the file and variables in the file:

0 if file exists, return with an error code.

1 if file exists, overwrite completely.

2 if file exists, append to file, appending to variable names if necessary to avoid name conflicts.

3 if file exists, update file. When a name confict occurs, update the existing variable in the file with the new variable.

report scalar, controls reporting:

0 no reporting.

1 report only name changes (note that name changes occur only when *overwrite* is set to 2).

3 report everything.

OUTPUT

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1 Null file name.

3 File write error.

4 File read error.

5 Invalid file type.

6 File exists and *overwrite* set to 0.

7 Cannot create file.

14 File too large to be read on current platform.

16 Cannot write to GDA – version outdated.

17 Type mismatch.

REMARKS Only initialized variables are written to the GDA with **gdaSave**.

If *varnames* is a null string, it will be interpreted as indicating all of the variables in the workspace.

You may add an asterisk (*****) to the end of a variable name in *varnames* to indicate that all variables beginning with the specified text are to be selected. For example, setting *varnames* to the string "**_***" and setting *exclude* to 1 indicates that all variables EXCEPT those starting with an underscore should be written to the GDA.

The names of the variables in the workspace are the names that are given to the variables when they are written to the GDA, with the exception of names that are changed to avoid conflicts.

If you set *overwrite* to 2, and variable name conflicts are encountered, **gdaSave** will append an underscore and a number to the name of the variable it is adding. It will first try changing the name to *name***_1**. If there is a conflict with that name, it will change it to *name***_2**, and so on until it finds a name that does not conflict with any of the variables already in the GDA.

EXAMPLE

```
run -r myfile.gau;
ret = gdaSave("myfile.gda","x*",0,2,3);
```

This example runs a **GAUSS** program called `myfile.gau` and then writes all initialized variables in the workspace beginning with '**x**' to the file `myfile.gda`. If `myfile.gda` already exists, this example appends to it, changing the names of the variables that it writes to the file if necessary to avoid name conficts. All writing and variable name changing is reported.

SEE ALSO **gdaLoad**

gdaUpdate

PURPOSE Updates a variable in a **GAUSS** Data Archive.

FORMAT *ret* = **gdaUpdate**(*filename*,*x*,*varname*);

INPUT

filename string, name of data file.

x matrix, array, string or string array, data.

varname string, variable name.

OUTPUT

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1 Null file name.
2 File open error.
3 File write error.
4 File read error.
5 Invalid data file type.
8 Variable not found.
10 File contains no variables.
14 File too large to be read on current platform.

REMARKS This command updates the variable *varname* in *filename* with the data contained in *x*.

If *x* is larger than the specified variable in the file, then **gdaUpdate** writes the new variable data after the last variable in the data file, moving the variable descriptor table to make room for the data and leaving empty bytes in the place of the old variable. This does not change the index of the variable because variable indices are determined NOT by the order of the variable data in a GDA, but by the order of the variable descriptors.

If *x* is the same size or smaller than the specified variable in the file, then **gdaUpdate** writes the data in *x* over the specified variable. If *x* is smaller, then **gdaUpdate** leaves empty bytes between the end of the updated variable and the beginning of the next variable in the data file.

This command updates variables quickly by not moving data in the file unnecessarily. However, calling **gdaUpdate** several times for one file may result in a file with a large number of empty bytes. To pack the data in a GDA,

so it contains no empty bytes, call **gdaPack**. Or to update a variable without leaving empty bytes in the file, call **gdaUpdateAndPack**.

EXAMPLE

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");

y = rndn(75,5);
ret = gdaUpdate("myfile.gda",y,"x1");
```

SEE ALSO **gdaUpdateAndPack**, **gdaPack**, **gdaWrite**

gdaUpdateAndPack

PURPOSE Updates a variable in a **GAUSS** Data Archive, leaving no empty bytes if the updated variable is smaller or larger than the variable it is replacing.

FORMAT *ret* **= gdaUpdateAndPack(***filename*,*x*,*varname***);**

INPUT

filename	string, name of data file.
x	matrix, array, string or string array, data.
varname	string, variable name.

OUTPUT

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.
2	File open error.
3	File write error.
4	File read error.
5	Invalid data file type.
8	Variable not found.
10	File contains no variables.

12 File truncate error.
14 File too large to be read on current platform.

REMARKS This command updates the variable *varname* in *filename* with the data contained in *x*. **gdaUpdateAndPack** always writes the data in *x* over the specified variable in the file. If *x* is larger than the specified variable, then it first moves all subsequent data in the file to make room for the new data. If *x* is smaller, then **gdaUpdateAndPack** writes the data, packs all of the subsequent data, leaving no empty bytes after the updated variable, and truncates the file.

This command uses disk space efficiently; however, it may be slow for large files (especially if the variable to be updated is one of the first variables in the file). If speed is a concern, you may want to use **gdaUpdate** instead.

EXAMPLE
```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");

y = rndn(75,5);
ret = gdaUpdateAndPack("myfile.gda",y,"x1");
```

SEE ALSO **gdaUpdate**, **gdaWrite**

gdaVars

PURPOSE Gets the number of variables in a **GAUSS** Data Archive.

FORMAT *nvars* **= gdaVars(***filename***);**

INPUT *filename* string, name of data file.

OUTPUT *nvars* scalar, the number of variables in *filename*.

EXAMPLE
```
nvars = gdaVars("myfile.gda");
```

SOURCE gdafns.src

SEE ALSO **gdaReportVarInfo**, **gdaGetNames**,

gdaWrite

PURPOSE Writes a variable to a **GAUSS** Data Archive.

FORMAT *ret* = **gdaWrite**(*filename*,*x*,*varname*);

INPUT

filename	string, name of data file.
x	matrix, array, string or string array, data to write to the GDA.
varname	string, variable name.

OUTPUT

ret scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.
2	File open error.
3	File write error.
4	File read error.
5	Invalid data file type.
9	Variable name too long.
11	Variable name must be unique.
14	File too large to be read on current platform.

REMARKS **gdaWrite** adds the data in *x* to the end of the variable data in *filename*, and gives the variable the name contained in *varname*.

EXAMPLE

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");
```

SEE ALSO **gdaWrite32**, **gdaCreate**

gdaWrite32

PURPOSE Writes a variable to a **GAUSS** Data Archive using 32-bit system file write commands.

FORMAT *ret* = **gdaWrite32**(*filename*,*x*,*varname*);

INPUT

filename	string, name of data file.
x	matrix, array, string or string array, data to write to the GDA.
varname	string, variable name.

OUTPUT *ret* scalar, return code, 0 if successful, otherwise one of the following error codes:

1	Null file name.
2	File open error.
3	File write error.
4	File read error.
5	Invalid data file type.
9	Variable name too long.
11	Variable name must be unique.
14	File too large to be read on current platform.
25	Not supported for use with a file created on a machine with a different byte order.

REMARKS **gdaWrite32** adds the data in *x* to the end of the variable data in *filename*, and gives the variable the name contained in *varname*.

This command is a speed optimization command for Windows. On all other platforms, this function is identical to **gdaWrite**. **gdaWrite** uses system file write commands that support 64-bit file sizes. These commands are slower on

Windows XP than the 32-bit file write commands that were used for binary writes in **GAUSS** 6.0. **gdaWrite32** uses the 32-bit Windows system write commands, which will be faster on Windows XP. Note, however, that **gdaWrite32** does not support 64-bit file sizes.

This command does not support writing to a GDA that was created on a platform with a different byte order than the current machine. **gdaWrite** supports full cross-platform writing to GDA's.

EXAMPLE

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite32("myfile.gda",x,"x1");
```

SEE ALSO **gdaWrite**, **gdaCreate**

gdaWriteSome

PURPOSE Overwrites part of a variable in a **GAUSS** Data Archive.

FORMAT *ret* **= gdaWriteSome(***filename***,***x***,***varname***,***index***);**

INPUT

filename	string, name of data file.
x	matrix, array, string or string array, data.
varname	string, variable name.
index	scalar or N×1 vector, index into variable where new data is to be written.

OUTPUT

ret	scalar, return code, 0 if successful, otherwise one of the following error codes:

1 Null file name.
2 File open error.
3 File write error.

4 File read error.
5 Invalid data file type.
8 Variable not found.
10 File contains no variables.
14 File too large to be read on current platform.
15 Argument out of range.
17 Type mismatch.
18 Argument wrong size.
19 Data must be real.
20 Data must be complex.

REMARKS This command overwrites part of the variable *varname* in *filename* with the data contained in *x*. The new data is written to *varname* beginning at the position indicated by *index*.

If *index* is a scalar, it will be interpreted as the $index^{th}$ element of the variable. Thus if *varname* references a 10×5 matrix, an *index* of 42 would indicate the 42^{nd} element, which is equivalent to the [8,2] element of the matrix (remember that **GAUSS** matrices are stored in row major order). If *index* is an N×1 vector, then N must equal the number of dimensions in the variable referenced by *varname*.

If *varname* references a string, then *index* must be a scalar containing an index into the string in characters.

gdaWriteSome may not be used to extend the size of a variable in a GDA. If there are more elements (or characters for strings) in *x* than there are from the indexed position of the specified variable to the end of that variable, then **gdaWriteSome** will fail. Call **gdaAppend** to append data to an existing variable.

The shape of *x* need not match the shape of the variable referenced by *varname*. If *varnum* references an N×K matrix, then *x* may be any L×M matrix (or P-dimensional array) that satisfies the size limitations described above. If *x* contains R elements, then the elements in *x* will simply replace the indexed element of the specified variable and the subsequent R-1 elements (as they are laid out in memory).

If *varname* references a string array, then the size of the overall variable will change if the sum of the length of the string array elements in *x* is different than the sum of the length of the elements that they are replacing.

In this case, if the variable increases in size, then the variable data will be rewritten after the last variable in the data file, moving the variable descriptor table to make room for the data and leaving empty bytes in its old location. This does not change the index of the variable because variable indices are determined NOT by the order of the variable data in a GDA, but by the order of the variable descriptors. If the variable decreases in size, then **gdaWriteSome** leaves empty bytes between the end of the variable and the beginning of the next variable in the data file. Call **gdaPack** to pack the data in a GDA, so it contains no empty bytes.

EXAMPLE

```
x = rndn(100,50);
ret = gdaCreate("myfile.gda",1);
ret = gdaWrite("myfile.gda",x,"x1");

y = rndn(75,5);
index = { 52,4 };
ret = gdaWriteSome("myfile.gda",y,"x1",index);
```

This example replaces 75*5=375 elements in **x1**, beginning with the [52,4] element, with the elements in **y**.

SEE ALSO **gdaReadSome**, **gdaUpdate**, **gdaWrite**

getarray

PURPOSE Gets a contiguous subarray from an N-dimensional array.

FORMAT *y* **= getarray(***a***,***loc***);**

INPUT *a* N-dimensional array.

loc M×1 vector of indices into the array to locate the subarray of interest, where 1<=M<=N.

OUTPUT *y* [N-M]-dimensional subarray or scalar.

REMARKS If N-M>0, **getarray** will return an array of [N-M] dimensions, otherwise, if N-M=0, it will return a scalar.

EXAMPLE
```
a = seqa(1,1,720);
a = areshape(a,2|3|4|5|6);
loc = { 2,1 };
y = getarray(a,loc);
```

y will be a 4×5×6 array of sequential values, beginning at [1,1,1] with 361, and ending at [4,5,6] with 480.

SEE ALSO **getmatrix**

getdims

PURPOSE Gets the number of dimensions in an array.

FORMAT *y* **= getdims(***a***);**

INPUT *a* N-dimensional array.

OUTPUT *y* scalar, the number of dimensions in the array.

EXAMPLE
```
a = arrayinit(3|4|5|6|7|2,0);
dims = getdims(a);
```

```
dims = 6
```

SEE ALSO **getorders**

getf

PURPOSE Loads an ASCII or binary file into a string.

FORMAT *y* **= getf(***filename***,***mode***);**

INPUT

filename	string, any valid file name.
mode	scalar 1 or 0 which determines if the file is to be loaded in ASCII mode (0) or binary mode (1).

OUTPUT

y	string containing the file.

REMARKS If the file is loaded in ASCII mode, it will be tested to see if it contains any end of file characters. These are ^Z (ASCII 26). The file will be truncated before the first ^Z, and there will be no ^Z's in the string. This is the correct way to load most text files because the ^Z's can cause problems when trying to print the string to a printer.

If the file is loaded in binary mode, it will be loaded just like it is with no changes.

EXAMPLE 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;
```

This produces:

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


SEE ALSO **load**, **save**, **let**, **con**

getmatrix

PURPOSE Gets a contiguous matrix from an N-dimensional array.

FORMAT *y* **= getmatrix(***a*,*loc***);**

INPUT

a	N-dimensional array.
loc	M×1 vector of indices into the array to locate the matrix of interest, where M equals N, N-1 or N-2.

OUTPUT

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

REMARKS Inputting an N×1 locator vector will return a scalar, an (N-1)×1 locator vector will return a 1×L matrix, and an (N-2)×1 locator vector will return a K×L matrix.

EXAMPLE
```
a = seqa(1,1,120);
a = areshape(a,2|3|4|5);
loc = { 1,2 };
y = getmatrix(a,loc);
```

$$y = \begin{matrix} 21 & 22 & 23 & 24 & 25 \\ 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 \\ 36 & 37 & 38 & 39 & 40 \end{matrix}$$

SEE ALSO **getarray**, **getmatrix4D**

getmatrix4D

PURPOSE Gets a contiguous matrix from a 4-dimensional array.

FORMAT *y* **= getmatrix4D(***a*,*i1*,*i2***);**

INPUT

a	4-dimensional array.
i1	scalar, index into the slowest moving dimension of the array.
i2	scalar, index into the second slowest moving dimension of the array.

OUTPUT

y	K×L matrix, where L is the size of the fastest moving dimension of the array and K is the size of the second fastest moving dimension.

REMARKS **getmatrix4D** returns the contiguous matrix that begins at the [*i1*,*i2*,1,1] position in array *a* and ends at the [*i1*,*i2*,K,L] position.

A call to **getmatrix4D** is faster than using the more general **getmatrix** function to get a matrix from a 4-dimensional array, especially when *i1* and *i2* are the counters from nested **for** loops.

EXAMPLE
```
a = seqa(1,1,120);
a = areshape(a,2|3|4|5);
y = getmatrix4D(a,2,3);
```

y =

101	102	103	104	105
106	107	108	109	110
111	112	113	114	115
116	117	118	119	120

SEE ALSO **getmatrix**, **getscalar4D**, **getarray**

getname

PURPOSE Returns a column vector containing the names of the variables in a **GAUSS** data set.

FORMAT *y* **= getname(***dset***);**

INPUT *dset* string specifying the name of the data set from which the function will obtain the variable names.

OUTPUT *y* N×1 vector containing the names of all of the variables in the specified data set.

REMARKS The output, *y*, will have as many rows as there are variables in the data set.

EXAMPLE
```
y = getname("olsdat");
format 8,8;
print $y;
```

produces:

```
 TIME
 DIST
 TEMP
FRICT
```

The above example assumes that the data set `olsdat` contains the variables: **TIME**, **DIST**, **TEMP**, **FRICT**.

Note that the extension is not included in the filename passed to the **getname** function.

SEE ALSO **getnamef**, **indcv**

getnamef

PURPOSE Returns a string array containing the names of the variables in a **GAUSS** data set.

FORMAT *y* = **getnamef**(*f*);

INPUT *f* scalar, file handle of an open data set

OUTPUT *y* N×1 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 that the data set `olsdat` contains the variables: **time**, **dist**, **temp**, **frict**.

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

SEE ALSO **getname**, **indcv**, **vartypef**

getNextTradingDay

PURPOSE Returns the next trading day.

FORMAT *n* = **getNextTradingDay**(*a*);

INPUT *a* scalar, date in DT scalar format.

OUTPUT *n* scalar, next trading day in DT scalar format.

REMARKS A trading day is a weekday that is not a holiday as defined by the New York Stock Exchange from 1888 through 2006. Holidays are defined in `holidays.asc`. You may edit that file to modify or add holidays.

SOURCE `finutils.src`

GLOBALS **_fin_holidays**

SEE ALSO **getPreviousTradingDay**, **annualTradingDays**

getNextWeekDay

PURPOSE Returns the next day that is not on a weekend.

FORMAT *n* = **getNextWeekDay**(*a*);

INPUT *a* scalar, date in DT scalar format.

OUTPUT *n* scalar, next week day in DT scalar format.

SOURCE finutils.src

SEE ALSO **getPreviousWeekDay**

getnr

PURPOSE Computes number of rows to read per iteration for a program that reads data from a disk file in a loop.

FORMAT *nr* = **getnr**(*nsets*,*ncols*);

INPUT *nsets* scalar, estimate of the maximum number of duplicate copies of the data matrix read by **readr** to be kept in memory during each iteration of the loop.

ncols scalar, columns in the data file.

OUTPUT *nr* scalar, number of rows **readr** should read per iteration of the read loop.

REMARKS If **__row** is greater than 0, *nr* will be set to **__row**.

If an insufficient memory error is encountered, change **__rowfac** to a number less than 1.0 (e.g., 0.75). The number of rows read will be reduced in size by this factor.

SOURCE gauss.src

GLOBALS **__row**, **__rowfac**, **__maxvec**

getnrmt

PURPOSE Computes number of rows to read per iteration for a program that reads data from a disk file in a loop.

FORMAT *nr* = **getnr**(*nsets*,*ncols*,*row*,*rowfac*,*maxv*);

INPUT

nsets	scalar, estimate of the maximum number of duplicate copies of the data matrix read by **readr** to be kept in memory during each iteration of the loop.
ncols	scalar, columns in the data file.
row	scalar, if row is greater than 0, *nr* will be set to *row*.
rowfac	scalar, *nr* will be reduced in size by this factor. If insufficient memory error is encounted, change this to a number less than one (e.g., 0.9).
maxv	scalar, the largest number of elements allowed in any one matrix.

OUTPUT

nr	scalar, number of rows **readr** should read per iteration of the read loop.

SOURCE gaussmt.src

getorders

PURPOSE Gets the vector of orders corresponding to an array.

FORMAT *y* **= getorders(***a***);**

INPUT *a* N-dimensional array.

OUTPUT *y* N×1 vector of orders, the sizes of the dimensions of the array.

EXAMPLE
```
a = arrayalloc(7|6|5|4|3,0);
orders = getorders(a);
```

```
          7
          6
orders =  5
          4
          3
```

SEE ALSO **getdims**

getpath

PURPOSE Returns an expanded filename including the drive and path.

FORMAT *fname* **= getpath(***pfname***);**

INPUT *pfname* string, partial filename with only partial or missing path information.

OUTPUT *fname* string, filename with full drive and path.

REMARKS This function handles relative path references.

EXAMPLE
```
y = getpath("temp.e");
print y;
```

produces:

```
/gauss/temp.e
```

assuming that `/gauss` is the current directory.

SOURCE `getpath.src`

getPreviousTradingDay

PURPOSE Returns the previous trading day.

FORMAT *n* **= getPreviousTradingDay(***a***);**

INPUT *a* scalar, date in DT scalar format.

OUTPUT *n* scalar, previous trading day in DT scalar format.

REMARKS A trading day is a weekday that is not a holiday as defined by the New York Stock Exchange from 1888 through 2006. Holidays are defined in `holidays.asc`. You may edit that file to modify or add holidays.

SOURCE `finutils.src`

GLOBALS **_fin_holidays**

SEE ALSO **getNextTradingDay**

getPreviousWeekDay

PURPOSE Returns the previous day that is not on a weekend.

FORMAT *n* = **getPreviousWeekDay(***a***);**

INPUT *a* scalar, date in DT scalar format.

OUTPUT *n* scalar, previous week day in DT scalar format.

SOURCE finutils.src

SEE ALSO **getNextWeekDay**

getRow

PURPOSE Returns a specified row from a matrix.

FORMAT *y* = **getRow(***a*,*row***);**

INPUT *a* N×K matrix

row The row of the matrix to extract.

OUTPUT *y* A 1×K row vector.

REMARKS **getRow** is designed to give an alternative access to rows in a matrix than indexing the matrix by brackets.

EXAMPLE
```
a = rndn(10,10);
y = getRow(a,3);
```

SEE ALSO **geTrRow**

getscalar3D

PURPOSE Gets a scalar from a 3-dimensional array.

FORMAT *y* = **getscalar3D**(*a*,*i1*,*i2*,*i3*);

INPUT

a	3-dimensional array.
i1	scalar, index into the slowest moving dimension of the array.
i2	scalar, index into the second slowest moving dimension of the array.
i3	scalar, index into the fastest moving dimension of the array.

OUTPUT

y	scalar, the element of the array indicated by the indices.

REMARKS **getscalar3D** returns the scalar that is located in the [*i1*,*i2*,*i3*] position of array *a*.

A call to **getscalar3D** is faster than using the more general **getmatrix** function to get a scalar from a 3-dimensional array.

EXAMPLE

```
a = seqa(1,1,24);
a = areshape(a,2|3|4);
y = getscalar3D(a,1,3,2);
```

```
y = 10
```

SEE ALSO **getmatrix**, **getscalar4D**, **getarray**

getscalar4D

PURPOSE Gets a scalar from a 4-dimensional array.

FORMAT *y* = **getscalar4D**(*a*,*i1*,*i2*,*i3*,*i4*);

INPUT

a	4-dimensional array.
i1	scalar, index into the slowest moving dimension of the array.
i2	scalar, index into the second slowest moving dimension of the array.
i3	scalar, index into the second fastest moving dimension of the array.
i4	scalar, index into the fastest moving dimension of the array.

OUTPUT

y	scalar, the element of the array indicated by the indices.

REMARKS **getscalar4D** returns the scalar that is located in the [*i1*,*i2*,*i3*,*i4*] position of array *a*.

A call to **getscalar4D** is faster than using the more general **getmatrix** function to get a scalar from a 4-dimensional array.

EXAMPLE

```
a = seqa(1,1,120);
a = areshape(a,2|3|4|5);
y = getscalar4D(a,1,3,2,5);
```

```
y = 50
```

SEE ALSO **getmatrix**, **getscalar3D**, **getarray**

getTrRow

PURPOSE Transposes a matrix and then returns a single row from it.

FORMAT *y* = **getTrRow**(*a*,*row*);

INPUT
	a	N×K matrix
	row	The row of the matrix to extract.

OUTPUT
	y	A 1×K row vector.

REMARKS **getRow** is designed to give an alternative access to rows in a matrix than indexing the matrix by brackets.

EXAMPLE
```
a = rndn(10,10);
y = getTrRow(a,3);
```

SEE ALSO **getRow**

getwind

PURPOSE Retrieve the current graphic panel number.

LIBRARY pgraph

FORMAT *n* = **getwind**;

OUTPUT
	n	scalar, graphic panel number of current graphic panel.

REMARKS The current graphic panel is the graphic panel in which the next graph will be drawn.

SOURCE pwindow.src

SEE ALSO **endwind**, **begwind**, **window**, **setwind**, **nextwind**

gosub

PURPOSE Causes a branch to a subroutine.

FORMAT
```
gosub label;
      .
      .
      .
label:
      .
      .
      .
return;
```

REMARKS For multi-line recursive user-defined functions, see PROCEDURES AND KEYWORDS, Chapter 12.

When a **gosub** statement is encountered, the program will branch to the label and begin executing from there. When a **return** statement is encountered, the program will resume executing at the statement following the **gosub** statement. Labels are 1-32 characters long and are followed by a colon. The characters can be A-Z or 0-9, but they must begin with an alphabetic character. Uppercase or lowercase is allowed.

It is possible to pass parameters to subroutines and receive parameters from them when they return. See the second example, following.

The only legal way to enter a subroutine is with a **gosub** statement.

If your subroutines are at the end of your program, you should have an **end**

statement before the first one to prevent the program from running into a subroutine without using a **gosub**. This will result in a **Return without gosub** error message.

The variables used in subroutines are not local to the subroutine and can be accessed from other places in your program. (See PROCEDURES AND KEYWORDS, Chapter 12.)

EXAMPLE

In the program below the name **mysub** is a label. When the **gosub** statement is executed, the program will jump to the label **mysub** and continue executing from there. When the **return** statement is executed, the program will resume executing at the statement following the **gosub**.


```
x = rndn(3,3); z = 0;
gosub mysub;
print z;
end;

/* ------ Subroutines Follow ------  */

mysub:

    z = inv(x);
    return;
```

Parameters can be passed to subroutines in the following way (line numbers are added for clarity):

```
1.  gosub mysub(x,y);
2.  pop j;    /* b will be in j */
3.  pop k;    /* a will be in k */
4.  t = j*k;
5.  print t;
6.  end;
7.
```

```
8.  /* ---- Subroutines Follow ----- */
9.
10.  mysub:
11.      pop b;    /* y will be in b */
12.      pop a;    /* x will be in a */
13.
14.      a = inv(b)*b+a;
15.      b = a'b;
16.      return(a,b);
```

In the above example, when the **gosub** statement is executed, the following sequence of events results (line numbers are included for clarity):

1. **x** and **y** are pushed on the stack and the program branches to the label **mysub** in line 10.

11. the second argument that was pushed, **y**, is **pop**'ped into **b**.

12. the first argument that was pushed, **x**, is **pop**'ped into **a**.

14. **inv(b)*b+a** is assigned to **a**.

15. **a'b** is assigned to **b**.

16. **a** and **b** are pushed on the stack and the program branches to the statement following the **gosub**, which is line 2.

2. the second argument that was pushed, **b**, is **pop**'ped into **j**.

3. the first argument that was pushed, **a**, is **pop**'ped into **k**.

4. **j*k** is assigned to **t**.

5. **t** is printed.

6. the program is terminated with the **end** statement.

Matrices are pushed on a last-in/first-out stack in the **gosub()** and **return()** statements. They must be **pop**'ped off in the reverse order. No intervening statements are allowed between the label and the **pop** or the **gosub** and the **pop**. Only one matrix may be **pop**'ped per **pop** statement.

SEE ALSO **goto**, **proc**, **pop**, **return**

goto

PURPOSE Causes a branch to a label.

FORMAT
```
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:

```
                1.0000000       2.0000000       3.0000000
```

SEE ALSO **gosub**, **if**

gradMT

PURPOSE Computes numerical gradient.

INCLUDE `optim.sdf`

FORMAT *g* = **gradMT(&***fct*,*par1*,*data1*);

INPUT

&*fct*	scalar, pointer to procedure returning either N×1 vector or 1×1 scalar.
par1	an instance of structure of type **PV** containing parameter vector at which gradient is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.

OUTPUT

g	N×K Jacobian or 1×K gradient.

REMARKS *par1* must be created using the **pvPack** procedures.

EXAMPLE

```
#include optim.sdf

struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);
```

```
proc fct(struct PV p0, struct DS d0);
    local p,y;
    p = pvUnpack(p0,"P");
    y = p[1] * exp( -p[2] * d0.dataMatrix );
    retp(y);
endp;

g = gradMT(&fct,p1,d0);
```

SOURCE gradmt.src

gradMTm

PURPOSE Computes numerical gradient with mask.

INCLUDE optim.sdf

FORMAT *g* **= gradMTm(&***fct***,***par1***,***data1***,***mask***);**

INPUT

&*fct*	scalar, pointer to procedure returning either N×1 vector or 1×1 scalar.
par1	an instance of structure of type **PV** containing parameter vector at which gradient is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.
mask	K×1 matrix, elements in *g* corresponding to elements of *mask* set to zero are not computed, otherwise they are computed.

OUTPUT

g	N×K Jacobian or 1×K gradient.

REMARKS *par1* must be created using the **pvPack** procedures.

EXAMPLE

```
#include optim.sdf
```

```
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc fct(struct PV p0, struct DS d0);
    local p,y;
    p = pvUnpack(p0,"P");
    y = p[1] * exp( -p[2] * d0.dataMatrix );
    retp(y);
endp;

mask = { 0, 1 };
g = gradMTm(&fct,p1,d0,mask);
```

SOURCE gradmt.src

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* **= gradp(&***f***,***x0***);**

INPUT **&***f* a pointer to a vector-valued function (f: K×1 → N×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×1 vector:

```
proc f(x);
    retp( exp(x.*b) );
endp;
```

x0 K×1 vector of points at which to compute gradient.

OUTPUT

g N×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×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

```
proc myfunc(x);
     retp( x.*2 .* exp( x.*x./3 ) );
endp;

x0 = 2.5|3.0|3.5;
y = gradp(&myfunc,x0);
```

```
            82.98901842      0.00000000      0.00000000
y =          0.00000000    281.19752975      0.00000000
             0.00000000      0.00000000   1087.95414117
```

It is a 3×3 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**

graphprt

PURPOSE Controls automatic printer hardcopy and conversion file output.

LIBRARY pgraph

FORMAT **graphprt**(*str*);

INPUT *str* string, control string.

PORTABILITY **UNIX**

Not supported.

REMARKS **graphprt** is used to create hardcopy output automatically without user intervention. The input string *str* can have any of the following items, separated by spaces. If *str* is a null string, the interactive mode is entered. This is the default.

-P	print graph.
-PO=*c*	set print orientation:
	L landscape.
	P portrait.
-C=*n*	convert to another file format:
	1 Encapsulated PostScript file.
	3 HPGL Plotter file.
	5 BMP (Windows Bitmap).
	8 WMF (Windows Enhanced Metafile).
-CF=*name*	set converted output file name.
-I	minimize (iconize) the graphics window.
-Q	close window after processing.

-W=*n* display graph, wait *n* seconds, then continue.

If you are not using graphic panels, you can call **graphprt** anytime before the call to the graphics routine. If you are using graphic panels, call **graphprt** just before the **endwind** statement.

The print option default values are obtained from the viewer application. Any parameters passed through **graphprt** will override the default values. See PUBLICATION QUALITY GRAPHICS, Chapter 24.

EXAMPLE Automatic print using a single graphics call:


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

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

```
library pgraph;
graphset;
load x,y;
begwind;
window(1,2,0);  /* create two windows  */

setwind(1);
    xy(x,y);    /* first graphics call */

nextwind;
    xy(x,y);    /* second graphics call */

graphprt("-p");
endwind;        /* print page containing all graphs */
```

The next example shows how to build a string to be used with **graphprt**:

```
library pgraph;
graphset;
load x,y;
cvtnam = "mycvt.eps";  /* name of output file */

/* concatenate options into one string */
cmdstr = "-c=1" $+ " -cf=" $+ cvtnam;
cmdstr = cmdstr $+ " -q";

graphprt(cmdstr);  /* tell "xy" to convert and close */
xy(x,y);           /* create graph and convert */
```

The above string **cmdstr** will read as follows:

"-c=1 -cf=mycvt.eps -q"

SOURCE pgraph.src

graphset

PURPOSE Reset graphics global variables to default values.

LIBRARY pgraph

FORMAT **graphset;**

REMARKS This procedure is used to reset the defaults between graphs.

graphset may be called between each graphic panel to be displayed.

To change the default values of the global control variables, make the appropriate changes in the file `pgraph.dec` and to the procedure **graphset**.

SOURCE `pgraph.src`

hasimag

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

FORMAT *y* = **hasimag**(*x*);

INPUT *x* N×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

prcnm	string, name of procedure that calls **header**.
dataset	string, name of data set.
ver	2×1 numeric vector, the first element is the major version number of the program, the second element is the revision number. Normally this argument will be the version/revision global **(__??_ver)** associated with the module within which header is called. This argument will be ignored if set to 0.

GLOBAL INPUT

__header	string, containing one or more of the following letters:

- **t** title is to be printed
- **l** lines are to bracket the title
- **d** a date and time is to be printed
- **v** version number of program is to be printed

f file name being analyzed is to be printed

__title string, title for header.

SOURCE gauss.src

headermt

PURPOSE Prints a header for a report.

FORMAT **headermt(***prcnm*,*dataset*,*ver*,*header*,*title***);**

INPUT

prcnm string, name of procedure that calls **header**.

dataset string, name of data set.

ver 2×1 numeric vector, the first element is the major version number of the program, the second element is the revision number. Normally this argument will be the version/revision global **(__??_ver)** associated with the module within which header is called. This argument will be ignored if set to 0.

header string, containing one or more of the following letters:

t title is to be printed
l lines are to bracket the title
d a date and time is to be printed
v version number of program is to be printed
f file name being analyzed is to be printed

title string, title for header.

SOURCE gaussmt.src

hess

PURPOSE Computes the Hessenberg form of a square matrix.

FORMAT { *h*,*z* } = **hess**(*x*);

INPUT
x K×K matrix.

OUTPUT
h K×K matrix, Hessenberg form.
z K×K matrix, transformation matrix.

REMARKS **hess** computes the Hessenberg form of a square matrix. The Hessenberg form is an intermediate step in computing eigenvalues. It also is useful for solving certain matrix equations that occur in control theory (see Van Loan, Charles F. "Using the Hessenberg Decomposition in Control Theory". *Algorithms and Theory in Filtering and Control*. Sorenson, D.C. and R.J. Wets, eds., Mathematical Programming Study No. 18, North Holland, Amsterdam, 1982, 102-111).

z is an orthogonal matrix that transforms *x* into *h* and vice versa. Thus:

$$h = z' xz$$

and since *z* is orthogonal,

$$x = zhz'$$

x is reduced to upper Hessenberg form using orthogonal similiarity transformations. This preserves the Frobenious norm of the matrix and the condition numbers of the eigenvalues.

hess uses the ORTRAN and ORTHES functions from EISPACK.

EXAMPLE

```
let x[3,3] = 1 2 3
             4 5 6
             7 8 9;

{ h,z } = hess(x);
```

```
          1.00000000  −3.59700730  −0.24806947
h =      −8.06225775  14.04615385   2.83076923
          0.00000000   0.83076923  −0.04615385
```

```
          1.00000000   0.00000000   0.00000000
z =       0.00000000  −0.49613894  −0.86824314
          0.00000000  −0.86824314   0.49613894
```


SEE ALSO **schur**

hessMT

PURPOSE Computes numerical Hessian.

INCLUDE `optim.sdf`

FORMAT *h* = **hessMT(&***fct***,***par1***,***data1***);**

INPUT

&*fct*	scalar, pointer to procedure returning either N×1 vector or 1×1 scalar.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.

OUTPUT

h	K×K matrix, Hessian.

REMARKS *par1* must be created using the **pvPack** procedures.

EXAMPLE
```
#include optim.sdf

struct PV p1;
struct DS d0;

p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc fct(struct PV p0, struct DS d0);
    local p,y;

    p = pvUnpack(p0,"P");
    y = p[1] * exp( -p[2] * d0.dataMatrix );
    retp(y);
endp;

h = hessMT(&fct,p1,d0);
```

SOURCE hessmt.src

hessMTg

PURPOSE Computes numerical Hessian using gradient procedure.

INCLUDE optim.sdf

FORMAT *h* **= hessMTg(&***gfct***,***par1***,***data1***);**

INPUT **&***gfct* scalar, pointer to procedure computing either 1×K gradient or N×K Jacobian.

par1 an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.

data1 structure of type **DS** containing any data needed by *gfct*.

OUTPUT *h* K×K matrix, Hessian.

REMARKS *par1* must be created using the **pvPack** procedures.

EXAMPLE
```
#include optim.sdf

struct PV p1;
struct DS d0;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

proc gfct(&fct, struct PV p0, struct DS d0);
    local p,y,g1,g2;

    p = pvUnpack(p0,"P");
    g1 = exp( -p[2] * d0.dataMatrix );
    y = p[1] * exp( -p[2] * d0.dataMatrix );
    g2 = -p[1] * d0.dataMatrix .* g1;
    retp(g1~g2);
endp;

h = hessMTg(&gfct,p1,d0);
```

SOURCE hessmt.src

hessMTgw

PURPOSE Computes numerical Hessian using gradient procedure with weights.

INCLUDE `optim.sdf`

FORMAT *h* = **hessMTgw(&***gfct*,*par1*,*data1*,*wgts*);

INPUT

&*gfct*	scalar, pointer to procedure computing either N×K Jacobian.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *gfct*.
wgts	N×1 vector.

OUTPUT

h	K×K matrix, Hessian.

REMARKS *par1* must be created using the **pvPack** procedures.

EXAMPLE

```
#include optim.sdf

struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");
struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);
wgts = zeros(5,1) | ones(10,1);

proc gfct(&fct, struct PV p0, struct DS d0);
    local p,y,g1,g2;

    p = pvUnpack(p0,"P");
    g1 = exp( -p[2] * d0.dataMatrix );
    y = p[1] * exp( -p[2] * d0.dataMatrix );
    g2 = -p[1] * d0.dataMatrix .* g1;
    retp(g1~g2);
endp;

h = hessMTgw(&gfct,p1,d0,wgts);
```

SOURCE hessmt.src

hessMTm

PURPOSE Computes numerical Hessian with mask.

INCLUDE optim.sdf

FORMAT *h* = **hessMTm(&***fct*,*par1*,*data1*,*mask*);

INPUT

&*fct*	scalar, pointer to procedure returning either N×1 vector or scalar.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.
mask	K×K matrix, elements in *h* corresponding to elements of mask set to zero are not computed, otherwise are computed.

OUTPUT

h	K×K matrix, Hessian.

REMARKS *par1* must be created using the **pvPack** procedures. Only lower left part of mask looked at.

EXAMPLE

```
#include optim.sdf

struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");
struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);

mask = { 1 1,
```

```
                1 0 };

proc fct(struct PV p0, struct DS d0);
    local p,y;

    p = pvUnpack(p0,"P");
    y = p[1] * exp( -p[2] * d0.dataMatrix );
    retp(y);
endp;

h = hessMTm(&fct,p1,d0,mask);
```

SOURCE hessmt.src

hessMTmw

PURPOSE Computes numerical Hessian with mask and weights.

INCLUDE optim.sdf

FORMAT *h* = **hessMTmw(&***fct*,*par1*,*data1*,*mask*,*wgts*);

INPUT

&*fct*	scalar, pointer to procedure returning N×1 vector.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.
mask	K×K matrix, elements in *h* corresponding to elements of mask set to zero are not computed, otherwise are computed.
wgts	N×1 vector, weights.

OUTPUT

h	K×K matrix, Hessian.

REMARKS *fct* must evaluate to an N×1 vector conformable to the weight vector. *par1* must be created using the **pvPack** procedures.

EXAMPLE
```
#include optim.sdf

struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");
struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);
wgts = zeros(5,1) | ones(10,1);

mask = { 1 1,
         1 0 };

proc fct(&fct, struct PV p0, struct DS d0, wgts);
    local p,y;

    p = pvUnpack(p0,"P");
    y = p[1] * exp( -p[2] * d0.dataMatrix );
    retp(y);
endp;

h = hessMTmw(&fct,p1,d0,mask,wgt);
```

SOURCE hessmt.src

hessMTw

PURPOSE Computes numerical Hessian with weights.

INCLUDE optim.sdf

FORMAT *h* = **hessMTw(&***fct*,*par1*,*data1*,*wgts*);

INPUT

&*fct*	scalar, pointer to procedure returning N×1 vector.
par1	an instance of structure of type **PV** containing parameter vector at which Hessian is to be evaluated.
data1	structure of type **DS** containing any data needed by *fct*.
wgts	N×1 vector, weights.

OUTPUT

h	K×K matrix, Hessian.

REMARKS *fct* must evaluate to an N×1 vector conformable to the weight vector. *par1* must be created using the **pvPack** procedures.

EXAMPLE

```
#include optim.sdf

struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,0.1|0.2,"P");
struct DS d0;
d0 = dsCreate;
d0.dataMatrix = seqa(1,1,15);
wgt = zeros(5,1) | ones(10,1);

proc fct(&fct, struct PV p0, struct DS d0, wgt);
    local p,y;

    p = pvUnpack(p0,"P");
    y = p[1] * exp( -p[2] * d0.dataMatrix );
    retp(y);
endp;

h = hessMTw(&fct,p1,d0,wgt);
```

SOURCE hessmt.src

hessp

PURPOSE Computes the matrix of second partial derivatives (Hessian matrix) of a function defined as a procedure.

FORMAT *h* = **hessp(&*f*,*x0*);**

INPUT

&*f*	pointer to a single-valued function $f(x)$, defined as a procedure, taking a single K×1 vector argument (f: K×1 → 1×1); $f(x)$ may be defined in terms of global arguments in addition to x.
x0	K×1 vector specifying the point at which the Hessian of $f(x)$ is to be computed.

OUTPUT

h	K×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 = hessp(&g,b0);
```

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**

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

x	M×1 vector of data.
v	N×1 vector, the breakpoints to be used to compute the frequencies - or - scalar, the number of categories.

OUTPUT

b	P×1 vector, the breakpoints used for each category.
m	P×1 vector, the midpoints of each category.
freq	P×1 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:

$$
\begin{array}{lllllll}
freq[1] & = & & & x & <= & b[1] \\
freq[2] & = & b[1] & < & x & <= & b[2] \\
freq[3] & = & b[2] & < & x & <= & b[3] \\
\quad . & & & & & & \\
\quad . & & & & & & \\
\quad . & & & & & & \\
freq[P] & = & b[P-1] & < & x & <= & b[P]
\end{array}
$$

EXAMPLE

```
library pgraph;
x = rndn(5000,1);
{ b,m,f } = hist(x,20);
```

SOURCE phist.src

SEE ALSO **histp**, **histf**, **bar**

histf

PURPOSE Graphs a histogram given a vector of frequency counts.

LIBRARY pgraph

FORMAT **histf**(*f*,*c*);

INPUT *f* N×1 vector, frequencies to be graphed.

c N×1 vector, numeric labels for categories. If this is a scalar 0, a sequence from 1 to **rows(***f***)** will be created.

REMARKS The axes are not automatically labeled. Use **xlabel** for the category axis and **ylabel** for the frequency axis.

SOURCE phist.src

SEE ALSO **hist**, **bar**, **xlabel**, **ylabel**

histp

PURPOSE Computes and graphs a percent frequency histogram of a vector. The percentages in each category are plotted.

LIBRARY pgraph

FORMAT { *b*,*m*,*freq* } = **histp(***x*,*v***);**

INPUT
x M×1 vector of data.

v N×1 vector, the breakpoints to be used to compute the frequencies
- or -
scalar, the number of categories.

OUTPUT
b P×1 vector, the breakpoints used for each category.

m P×1 vector, the midpoints of each category.

freq P×1 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:

$$
\begin{array}{lllllll}
freq[1] & = & & & x & <= & b[1] \\
freq[2] & = & b[1] & < & x & <= & b[2] \\
freq[3] & = & b[2] & < & x & <= & b[3] \\
\quad . & & & & & & \\
\quad . & & & & & & \\
\quad . & & & & & & \\
freq[P] & = & b[P-1] & < & x & <= & b[P]
\end{array}
$$

SOURCE phist.src

SEE ALSO **hist**, **histf**, **bar**

hsec

PURPOSE Returns the number of hundredths of a second since midnight.

FORMAT *y* **= hsec;**

OUTPUT *y* scalar, hundredths of a second since midnight.

REMARKS The number of hundredths of a second since midnight can also be accessed as the [4,1] element of the vector returned by the **date** function.

EXAMPLE
```
x = rndu(1000,1000);
ts = hsec;
```

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

In this example, **hsec** is used to time a 1000×1000 multiplication in **GAUSS**. A 1000×1000 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.

EXAMPLE

```
if x < 0;
    y = -1;
elseif x > 0;
    y = 1;
else;
    y = 0;
endif;
```

SEE ALSO **do**

imag

PURPOSE Returns the imaginary part of *x*.

FORMAT *zi* **= imag(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *zi* N×K matrix or N-dimensional array, the imaginary part of *x*.

REMARKS If *x* is real, *zi* will be an N×K matrix or N-dimensional array of zeros.

EXAMPLE

```
x = { 4i 9 3,
      2 5-6i 7i };
y = imag(x);
```

$$y = \begin{matrix} 4.0000000 & 0.0000000 & 0.0000000 \\ 0.0000000 & -6.0000000 & 7.0000000 \end{matrix}$$

SEE ALSO **complex**, **real**

#include

PURPOSE Inserts code from another file into a **GAUSS** program.

FORMAT **#include** *filename*;

#include ''*filename***'';**

REMARKS *filename* can be any legitimate file name.

This command makes it possible to write a section of general-purpose code, and insert it into other programs.

The code from the **#include**'d file is inserted literally as if it were merged into that place in the program with a text editor.

If a path is specified for the file, then no additional searching will be attempted if the file is not found.

If a path is not specified, the current directory will be searched first, then each directory listed in **src_path**. **src_path** is defined in `gauss.cfg`.

`#include /gauss/myprog.prc;`	No additional search will be made if the file is not found.
`#include myprog.prc;`	The directories listed in **src_path** will be searched for `myprog.prc` if the file is not found in the current directory.

Compile time errors will return the line number and the name of the file in which they occur. For execution time errors, if a program is compiled with **#lineson**, the line number and name of the file where the error occurred will be printed. For files that have been **#include**'d this reflects the actual line number within the **#include**'d file. See **#lineson** for a more complete discussion of the use of and the validity of line numbers when debugging.

EXAMPLE

```
#include "/gauss/inc/cond.inc";
```

The command will cause the code in the program `cond.inc` to be merged into the current program at the point at which this statement appears.

SEE ALSO **run**, **#lineson**

indcv

PURPOSE Checks one character vector against another and returns the indices of the elements of the first vector in the second vector.

FORMAT *z* **= indcv(***what*,*where***);**

INPUT

what	N×1 character vector which contains the elements to be found in vector *where*.
where	M×1 character vector to be searched for matches to the elements of *what*.

OUTPUT

z	N×1 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
```

```
    1
z = 5
    2
```

SEE ALSO **indnv**, **indsav**

indexcat

PURPOSE Returns the indices of the elements of a vector which fall into a specified

category

FORMAT `y = indexcat(x,v);`

INPUT

x N×1 vector.

v scalar or 2×1 vector.

If scalar, the function returns the indices of all elements of *x* equal to *v*.

If 2×1, 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 L×1 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);
```

x =

1.0
4.0
3.3
4.2
6.0
5.7
8.1
5.5

v =

4
6

$$y = \begin{matrix} 4 \\ 5 \\ 6 \\ 8 \end{matrix}$$

indices

PURPOSE Processes a set of variable names or indices and returns a vector of variable names and a vector of indices.

FORMAT { *name*,*indx* } = **indices**(*dataset*,*vars*);

INPUT

dataset string, the name of the data set.

vars N×1 vector, a character vector of names or a numeric vector of column indices.

If scalar 0, all variables in the data set will be selected.

OUTPUT

name N×1 character vector, the names associated with *vars*.

indx N×1 numeric vector, the column indices associated with *vars*.

REMARKS If an error occurs, **indices** will either return a scalar error code or terminate the program with an error message, depending on the **trap** state. If the low order bit of the trap flag is 0, **indices** will terminate with an error message. If the low order bit of the trap flag is 1, **indices** will return an error code. The value of the trap flag can be tested with **trapchk**; the return from **indices** can be tested with **scalerr**. You only need to check one argument; they will both be the same. The following error codes are possible:

1 Can't open dataset.
2 Index of variable out of range, or undefined data set variables.

SOURCE indices.src

indices2

PURPOSE Processes two sets of variable names or indices from a single file. The first is a single variable and the second is a set of variables. The first must not occur in the second set and all must be in the file.

FORMAT { *name1*,*indx1*,*name2*,*indx2* } = **indices2**(*dataset*,*var1*,*var2*);

INPUT

dataset string, the name of the data set.

var1 string or scalar, variable name or index.
This can be either the name of the variable, or the column index of the variable.
If null or 0, the last variable in the data set will be used.

var2 N×1 vector, a character vector of names or a numeric vector of column indices.
If scalar 0, all variables in the data set except the one associated with *var1* will be selected.

OUTPUT

name1 scalar character matrix containing the name of the variable associated with *var1*.

indx1 scalar, the column index of *var1*.

name2 N×1 character vector, the names associated with *var2*.

indx2 N×1 numeric vector, the column indices of *var2*.

REMARKS If an error occurs, **indices2** will either return a scalar error code or terminate the program with an error message, depending on the **trap** state. If the low order bit of the trap flag is 0, **indices2** will terminate with an error message. If the low order bit of the trap flag is 1, **indices2** will return an error code. The value of the trap flag can be tested with **trapchk**; the return from **indices2** can be tested with **scalerr**. You only need to check one argument; they will all be the same. The following error codes are possible:

1 Can't open dataset.
2 Index of variable out of range, or undefined data set variables.
3 First variable must be a single name or index.
4 First variable contained in second set.

SOURCE `indices2.src`

indicesf

PURPOSE Processes a set of variable names or indices and returns a vector of variable names and a vector of indices.

FORMAT { *name*,*indx* } = **indicesf**(*fp*,*namein*,*indxin*);

INPUT

fp	scalar, file handle of an open data set.
namein	N×1 string array, names of selected columns in the data set. If set to a null string, columns are selected using *indxin*
indxin	N×1 vector, indices of selected columns in the data set. If set to 0, columns are selected using *namein*.

OUTPUT

name	N×1 string array, the names of the selected columns.
indx	N×1 vector, the indices of the selected columns.

REMARKS If *namein* is a null string and *indxin* is 0, all columns of the data set will be selected.

If an error occurs, *indx* will be set to a scalar error code. The following error codes are possible:

1 Can't open data file
2 Variable not found
3 Indices outside of range of columns

SOURCE indices.src

SEE ALSO **indicesfn**, **indices**

indicesfn

PURPOSE Processes a set of variable names or indices and returns a vector of variable names and a vector of indices.

FORMAT { *name*,*indx* } = **indicesfn**(*dataset*,*namein*,*indxin*);

INPUT

dataset	string, name of the data set.
namein	N×1 string array, names of selected columns in the data set. If set to a null string, columns are selected using *indxin*
indxin	N×1 vector, indices of selected columns in the data set. If set to 0, columns are selected using *namein*.

OUTPUT

name	N×1 string array, the names of the selected columns.
indx	N×1 vector, the indices of the selected columns.

REMARKS If *namein* is a null string and *indxin* is 0, all columns of the data set will be selected.

If an error occurs, *indx* will be set to a scalar error code. The following error codes are possible:

1 Can't open data file
2 Variable not found
3 Indices outside of range of columns

SOURCE indices.src

SEE ALSO **indicesf**, **indices**

indnv

PURPOSE Checks one numeric vector against another and returns the indices of the elements of the first vector in the second vector.

FORMAT *z* = **indnv**(*what*,*where*);

INPUT

what N×1 numeric vector which contains the values to be found in vector *where*.

where M×1 numeric vector to be searched for matches to the values in *what*.

OUTPUT

z N×1 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


          3
z =       2
          5
```

SEE ALSO **indcv**

indsav

PURPOSE Checks one string array against another and returns the indices of the first string array in the second string array.

FORMAT *indx* **= indsav(***what***,***where***);**

INPUT

what	N×1 string array which contains the values to be found in vector *where*.
where	M×1 string array to be searched for the corresponding elements of *what*.

OUTPUT

indx	N×1 vector of indices, the values of *what* in *where*.

REMARKS If no matches are found, those elements in the returned vector are set to the **GAUSS** missing value code.

If there are duplicate elements in *where*, the index of the first match will be returned.

intgrat2

PURPOSE Integrates the following double integral, using user-defined functions f, g_1 and g_2 and scalars a and b:

$$\int_a^b \int_{g_2(x)}^{g_1(x)} f(x,y)dydx$$

FORMAT *y* = **intgrat2(&***f***,***xl***,***gl***);**

INPUT

&*f*	scalar, pointer to the procedure containing the function to be integrated.
xl	2×1 or 2×N matrix, the limits of x. These must be scalar limits.
gl	2×1 or 2×N matrix of function pointers, the limits of y. For *xl* and *gl*, the first row is the upper limit and the second row is the lower limit. N integrations are computed.

GLOBAL INPUT

_intord	scalar, the order of the integration. The larger **_intord**, the more precise the final result will be. **_intord** may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.
_intrec	scalar. This variable is used to keep track of the level of recursion of **intgrat2** and may start out with a different value if your program terminated inside of the integration function on a previous run. Always set **_intrec** explicitly to 0 before any call to **intgrat2**.

OUTPUT

y	N×1 vector of the estimated integral(s) of $f(x,y)$, evaluated between the limits given by *xl* and *gl*.

REMARKS The user-defined functions specified by f and *gl* must either

1. Return a scalar constant
   - or -
2. Return a vector of function values. **intgrat2** will pass to user-defined functions a vector or matrix for *x* and *y* and expect a vector or matrix to be returned. Use **.*** and **./** instead of ***** and **/**.

EXAMPLE

```
proc f(x,y);
    retp(cos(x) + 1).*(sin(y) + 1));
endp;

proc g1(x);
   retp(sqrt(1-x^2));
endp;

proc g2(x);
    retp(0);
endp;

xl = 1|-1;
g0 = &g1|&g2;
_intord = 40;
_intrec = 0;
y = intgrat2(&f,xl,g0);
```

This will integrate the function $f(x, y) = (cos(x) + 1)(sin(y) + 1)$ over the upper half of the unit circle. Note the use of the **.*** operator instead of just ***** in the definition of $f(x, y)$. This allows f to return a vector or matrix of function values.

SOURCE intgrat.src

GLOBALS **_intord**, **_intq12**, **_intq16**, **_intq2**, **_intq20**, **_intq24**, **_intq3**, **_intq32**, **_intq4**, **_intq40**, **_intq6**, **_intq8**, **_intrec**

SEE ALSO **intgrat3**, **intquad1**, **intquad2**, **intquad3**, **intsimp**

intgrat3

PURPOSE Integrates the following triple integral, using user-defined functions and scalars for bounds:

$$\int_a^b \int_{g_2(x)}^{g_1(x)} \int_{h_2(x,y)}^{h_1(x,y)} f(x,y,z)dzdydx$$

FORMAT *y* = **intgrat3(&***f*,*xl*,*gl*,*hl*);

INPUT

&*f*	scalar, pointer to the procedure containing the function to be integrated. *f* is a function of (*x*,*y*,*z*).
xl	2×1 or 2×N matrix, the limits of *x*. These must be scalar limits.
gl	2×1 or 2×N matrix of function pointers. These procedures are functions of *x*.
hl	2×1 or 2×N 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.

GLOBAL INPUT

_intord	scalar, the order of the integration. The larger **_intord**, the more precise the final result will be. **_intord** may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.
_intrec	scalar. This variable is used to keep track of the level of recursion of **intgrat3** and may start out with a different value if your program terminated inside of the integration function on a previous run. Always set **_intrec** explicitly to 0 before any call to **intgrat3**.

OUTPUT

y	N×1 vector of the estimated integral(s) of $f(x,y,z)$ evaluated between the limits given by *xl*, *gl* and *hl*.

REMARKS User-defined functions *f*, and those used in *gl* and *hl* must either:

1. Return a scalar constant
   - or -
2. Return a vector of function values. **intgrat3** will pass to user-defined functions a vector or matrix for *x* and *y* and expect a vector or matrix to be returned. Use **.*** and **./** operators instead of just ***** and **/**.

EXAMPLE

```
proc f(x,y,z);
   retp(2);
endp;

proc g1(x);
    retp(sqrt(25-x^2));
endp;

proc g2(x);
    retp(-g1(x));
endp;

proc h1(x,y);
     retp(sqrt(25 - x^2 - y^2));
endp;

proc h2(x,y);
     retp(-h1(x,y));
endp;

xl = 5|-5;
g0 = &g1|&g2;
h0 = &h1|&h2;
_intrec = 0;
_intord = 40;
y = intgrat3(&f,xl,g0,h0);
```

This will integrate the function $f(x, y, z) = 2$ 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**

inthp1

PURPOSE Integrates a user-defined function over an infinite interval.

INCLUDE inthp.sdf

FORMAT *y* = **inthp1(&***f***,***pds***,***ctl***);**

INPUT **&***f* scalar, pointer to the procedure containing the function to be integrated.

pds scalar, pointer to instance of a **DS** structure. The members of the **DS** are:

pds->dataMatrix N×K matrix.
pds->dataArray N×K×L... array.
pds->vnames string array.
pds->dsname string.
pds->type scalar.

The contents, if any, are set by the user and are passed by **inthp1** to the user-provided function without modification.

ctl instance of an **inthpControl** structure with members

ctl.maxEvaluations scalar, maximum number of function evaluations, default = 1e5;

	ctl.p	scalar, termination parameter **0** heuristic termination, default. **1** deterministic termination with infinity norm. **2,...** deterministic termination with p-th norm.
	ctl.d	scalar termination parameter **1** if heuristic termination **0** < *ctl*.**d** ≤ $\pi/2$ if deterministic termination
	ctl.eps	scalar, relative error bound. Default = 1e-6.

A default *ctl* can be generated by calling **inthpControlCreate**.

OUTPUT

y	scalar, the estimated integral of $f(x)$ evaluated over the interval $(-\infty, +\infty)$.

REMARKS The user-provided function must have the following format

f(**struct DS ***pds*,*x*)

where

pds	scalar, pointer to an instance of a **DS** structure.
x	scalar, value at which integral will be evaluated.

If *ctl.d* can be specified (see Sikorski and Stenger, 1984), deterministic termination can be specified and accuracy guaranteed. if not, the heuristic method can be used and the value of *clt.d* is disregarded.

The pointer to the instance of the data structure, *pds*, is passed untouched to the user-provided procedure computing the function to be integrated. Any information needed by that function can be put into that data structure.

EXAMPLE

```
#include inthp.sdf

proc fct(struct DS *pds, x);
```

```
    local var;
    var = pds->dataMatrix;
    retp( exp( -(x*x) / (2*var) ) );
endp;

struct DS d0;
struct DS *pds;

variance = 3;

pds = &d0;
d0.dataMatrix = variance;

struct inthpControl c0;
c0 = inthpControlCreate;

r = inthp1(&fct,pds,c0);

format /ld 16,10;
print r;
print sqrt(2*pi*variance);

   4.3416075273
   4.3416075273
```

REFERENCES

1. "Optimal Quadratures in H_p Spaces" by K. Sikorski and F. Stenger, *ACM Transactions on Mathematical Software*, 10(2):140-151, June 1984.

SOURCE inthp.src

SEE ALSO **inthpControlCreate**, **inthp2**, **inthp3**, **inthp4**

inthp2

PURPOSE Integrates a user-defined function over the $[a, +\infty)$ interval.

INCLUDE `inthp.sdf`

FORMAT *y* = **inthp2**(&*f*,*pds*,*ctl*,*a*);

INPUT

&*f* scalar, pointer to the procedure containing the function to be integrated.

pds scalar, pointer to instance of a **DS** structure. The members of the **DS** are:

pds->dataMatrix	N×K matrix.
pds->dataArray	N×K×L... array.
pds->vnames	string array.
pds->dsname	string.
pds->type	scalar.

The contents, if any, are set by the user and are passed by **inthp1** to the user-provided function without modification.

ctl instance if an **inthpControl** structure with members

ctl.maxEvaluations scalar, maximum number of function evaluations, default = 1e5;

ctl.p scalar, termination parameter

- **0** heuristic termination, default.
- **1** deterministic termination with infinity norm.
- **2,...** deterministic termination with p-th norm.

ctl.d scalar termination parameter

- **1** if heuristic termination
- $\mathbf{0} < ctl\mathbf{.d} \leq \pi/2$ if deterministic termination

ctl.eps scalar, relative error bound. Default = 1e-6.

A default *ctl* can be generated by calling **inthpControlCreate**.

a 1×N vector, lower limits of integration

OUTPUT

y N×1 vector, the estimated integrals of $f(x)$ evaluated over the interval $[a, +\infty)$.

REMARKS

The user-provided function must have the following format

f`(struct DS *`*pds*`,`*x*`)`

where

pds scalar, pointer to an instance of a **DS** structure.

x scalar, value at which integral will be evaluated.

If *ctl.d* can be specified (see Sikorski and Stenger, 1984), deterministic termination can be specified and accuracy guaranteed. if not, the heuristic method can be used and the value of *clt.d* is disregarded.

The pointer to the instance of the data structure, *pds*, is passed untouched to the user-provided procedure computing the function to be integrated. Any information needed by that function can be put into that data structure.

EXAMPLE

```
#include inthp.sdf

proc normal(struct DS *pd0, x);
    local var;
    var = pd0->dataMatrix;
    retp( (1/sqrt(2*pi*var))*exp( -(x*x) / (2*var) ) );
endp;


struct DS d0;
struct DS *pd0;

pd0 = &d0;
d0.dataMatrix = var;

struct inthpControl c0;
c0 = inthpControlCreate;
```

```
lim = 2;

c0.d = pi/4;
c0.p = 2;

var = 1;

r = inthp2(&normal,pd0,c0,lim);

format /ld 16,10;
print r;
print cdfnc(2);

    0.0227501281
    0.0227501319
```


REFERENCES 1. "Optimal Quadratures in H_p Spaces" by K. Sikorski and F. Stenger, *ACM Transactions on Mathematical Software*, 10(2):140-151, June 1984.

SOURCE inthp.src

SEE ALSO **inthpControlCreate**, **inthp1**, **inthp3**, **inthp4**

inthp3

PURPOSE Integrates a user-defined function over the $[a, +\infty)$ interval that is oscillatory.

INCLUDE inthp.sdf

FORMAT *y* **= inthp3(&***f***,***pds***,***ctl***,***a***);**

INPUT **&***f* scalar, pointer to the procedure containing the function to be integrated.

pds scalar, pointer to instance of a **DS** structure. The members of the **DS** are:

pds->dataMatrix N×K matrix.
pds->dataArray N×K×L... array.
pds->vnames string array.
pds->dsname string.
pds->type scalar.

The contents, if any, are set by the user and are passed by **inthp1** to the user-provided function without modification.

ctl instance if an **inthpControl** structure with members

ctl.maxEvaluations scalar, maximum number of function evaluations, default = 1e5;

ctl.p scalar, termination parameter

0 heuristic termination, default.

1 deterministic termination with infinity norm.

2,... deterministic termination with p-th norm.

ctl.d scalar termination parameter

1 if heuristic termination

0 < *ctl*.**d** ≤ $\pi/2$ if deterministic termination

ctl.eps scalar, relative error bound. Default = 1e-6.

A default *ctl* can be generated by calling **inthpControlCreate**.

a 1×N vector, lower limits of integration

OUTPUT *y* N×1 vector, the estimated integrals of $f(x)$ evaluated over the interval $[a, +\infty)$.

REMARKS This procedure is designed especially for oscillatory functions.

The user-provided function must have the following format

f(**struct DS ***pds*,*x*)

where

pds scalar, pointer to an instance of a **DS** structure.

x scalar, value at which integral will be evaluated.

If *ctl.d* can be specified (see Sikorski and Stenger, 1984), deterministic termination can be specified and accuracy guaranteed. if not, the heuristic method can be used and the value of *clt.d* is disregarded.

The pointer to the instance of the data structure, *pds*, is passed untouched to the user-provided procedure computing the function to be integrated. Any information needed by that function can be put into that data structure.

EXAMPLE

```
#include inthp.sdf

proc fct(struct DS *pd0, x);
    local m,a;
    m = pd0->dataMatrix[1];
    a = pd0->dataMatrix[2];
    retp( exp(-a*x)*cos(m*x) );
endp;

struct DS d0;
struct DS *pd0;

struct inthpControl c0;
c0 = inthpControlCreate;

c0.p = 2;
c0.d = pi/3;

m = 2;
a = 1;
pd0 = &d0;
d0.dataMatrix = m | a;

lim = 0;

r = inthp3(&fct,pd0,c0,lim);
```

```
format /ld 16,10;
print r;
print a/(a*a + m*m);

   0.2000000000
   0.2000000000
```

REFERENCES 1. "Optimal Quadratures in H_p Spaces" by K. Sikorski and F. Stenger, *ACM Transactions on Mathematical Software*, 10(2):140-151, June 1984.

SOURCE inthp.src

SEE ALSO **inthpControlCreate**, **inthp1**, **inthp2**, **inthp4**

inthp4

PURPOSE Integrates a user-defined function over the [*a*,*b*] interval.

INCLUDE inthp.sdf

FORMAT *y* = **inthp4(&***f*,*pds*,*ctl*,*c*);

INPUT **&***f* scalar, pointer to the procedure containing the function to be integrated.

pds scalar, pointer to instance of a **DS** structure. The members of the **DS** are:

pds->dataMatrix	N×K matrix.
pds->dataArray	N×K×L... array.
pds->vnames	string array.
pds->dsname	string.
pds->type	scalar.

The contents, if any, are set by the user and are passed by **inthp1** to the user-provided function without modification.

ctl instance if an **inthpControl** structure with members

ctl.maxEvaluations scalar, maximum number of function evaluations, default = 1e5;

ctl.p scalar, termination parameter

0 heuristic termination, default.

1 deterministic termination with infinity norm.

2,... deterministic termination with p-th norm.

ctl.d scalar termination parameter

1 if heuristic termination

$\mathbf{0} < ctl\mathbf{.d} \leq \pi/2$ if deterministic termination

ctl.eps scalar, relative error bound. Default = 1e-6.

A default *ctl* can be generated by calling **inthpControlCreate**.

c 2×N vector, upper and lower limits of integration, the first row contains upper limits and the second row the lower.

OUTPUT *y* N×1 vector, the estimated integrals of $f(x)$ evaluated over the interval $[a,b]$.

REMARKS The user-provided function must have the following format

f(**struct DS ***_pds_,_x_)

where

pds scalar, pointer to an instance of a **DS** structure.

x scalar, value at which integral will be evaluated.

If *ctl.d* can be specified (see Sikorski and Stenger, 1984), deterministic termination can be specified and accuracy guaranteed. if not, the heuristic method can be used and the value of *clt.d* is disregarded.

The pointer to the instance of the data structure, *pds*, is passed untouched to the user-provided procedure computing the function to be integrated. Any information needed by that function can be put into that data structure.

EXAMPLE

```
#include inthp.sdf

proc fct(struct DS *pd0, x);
    local a,b,c;
    a = pd0->dataMatrix[1];
    b = pd0->dataMatrix[2];
    c = pd0->dataMatrix[3];
    retp( 1/sqrt(a*x*x + b*x + c) );
endp;

struct DS d0;
struct DS *pd0;

struct inthpControl c0;
c0 = inthpControlCreate;

c0.p = 2;
c0.d = pi/2;

a = -1;
b = -2;
c = 3;
pd0 = &d0;
d0.dataMatrix = a|b|c;

lim = 1 | -1;

r = inthp4(&fct,pd0,c0,lim);

format /ld 16,10;
print r;
print pi/2;
```

```
1.5707962283
1.5707963268
```

REFERENCES 1. "Optimal Quadratures in H_p Spaces" by K. Sikorski and F. Stenger, *ACM Transactions on Mathematical Software*, 10(2):140-151, June 1984.

SOURCE inthp.src

SEE ALSO **inthpControlCreate**, **inthp1**, **inthp2**, **inthp3**

inthpControlCreate

PURPOSE Creates default **inthpControl** structure.

INCLUDE inthp.sdf

FORMAT *c* = **inthpControlCreate;**

OUTPUT *c* instance of **inthpControl** structure with members set to default values.

SOURCE inthp.src

SEE ALSO **inthp1**, **inthp2**, **inthp3**, **inthp4**

intquad1

PURPOSE Integrates a specified function using Gauss-Legendre quadrature. A suite of upper and lower bounds may be calculated in one procedure call.

FORMAT *y* = **intquad1(&*f*,*xl*);**

INPUT

&*f*	scalar, pointer to the procedure containing the function to be integrated. This must be a function of *x*.
xl	2×N matrix, the limits of *x*. The first row is the upper limit and the second row is the lower limit. N integrations are computed.

GLOBAL INPUT

_intord	scalar, the order of the integration. The larger **_intord**, the more precise the final result will be. **_intord** may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.

OUTPUT

y	N×1 vector of the estimated integral(s) of $f(x)$ evaluated between the limits given by *xl*.

REMARKS The user-defined function *f* must return a vector of function values. **intquad1** will pass to the user-defined function a vector or matrix for *x* and expect a vector or matrix to be returned. Use the **.*** and **./** instead of ***** and **/**.

EXAMPLE

```
proc f(x);
   retp(x.*sin(x));
endp;

xl =  1|0;
y = intquad1(&f,xl);
```

This will integrate the function $f(x) = xsin(x)$ between 0 and 1. Note the use of the **.*** instead of *****.

SOURCE integral.src

GLOBALS **_intord**, **_intq12**, **_intq16**, **_intq2**, **_intq20**, **_intq24**, **_intq3**, **_intq32**, **_intq4**, **_intq40**, **_intq6**, **_intq8**

SEE ALSO **intsimp**, **intquad2**, **intquad3**, **intgrat2**, **intgrat3**

intquad2

PURPOSE Integrates a specified function using Gauss-Legendre quadrature. A suite of upper and lower bounds may be calculated in one procedure call.

FORMAT *y* **= intquad2(&***f***,***xl***,***yl***);**

INPUT

&*f*	scalar, pointer to the procedure containing the function to be integrated.
xl	2×1 or 2×N matrix, the limits of *x*.
yl	2×1 or 2×N matrix, the limits of *y*.

For *xl* and *yl*, the first row is the upper limit and the second row is the lower limit. N integrations are computed.

GLOBAL INPUT

_intord	scalar, the order of the integration. The larger **_intord**, the more precise the final result will be. **_intord** may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.
_intrec	scalar. This variable is used to keep track of the level of recursion of **intquad2** and may start out with a different value if your program terminated inside of the integration function on a previous run. Always set **_intrec** explicitly to 0 before any calls to **intquad2**.

OUTPUT

y	N×1 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**.

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* **= 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 2×1 or 2×N matrix, the limits of x.

yl 2×1 or 2×N matrix, the limits of y.

zl 2×1 or 2×N matrix, the limits of z.

For *xl*, *yl*, and *zl*, the first row is the upper limit and the second row is the lower limit. N integrations are computed.

GLOBAL INPUT

_intord scalar, the order of the integration. The larger **_intord**, the more precise the final result will be. **_intord** may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40.
Default = 12.

_intrec scalar. This variable is used to keep track of the level of recursion of **intquad3** and may start out with a different value if your program terminated inside of the integration function on a previous run. Always set **_intrec** explicitly to 0 before any calls to **intquad3**.

OUTPUT

y N×1 vector of the estimated integral(s) of $f(x, y, z)$ evaluated between the limits given by *xl*, *yl*, and *zl*.

REMARKS

The user-defined function f must return a vector of function values. **intquad3** will pass to the user-defined function a vector or matrix for x, y and z and expect a vector or matrix to be returned. Use **.*** and **./** instead of ***** and **/**.

intquad3 will expand scalars to the appropriate size. This means that functions can be defined to return a scalar constant. If users write their functions incorrectly (using ***** instead of **.***, for example), **intquad3** may not compute the expected integral, but the integral of a constant function.

To integrate over a region which is bounded by functions, rather than just scalars, use **intgrat2** or **intgrat3**.

EXAMPLE

```
proc f(x,y,z);
   retp(x.*y.*z);
endp;
```

```
xl = 1|0;
yl = 1|0;
zl = { 1 2 3, 0 0 0 };

_intrec = 0;
y = intquad3(&f,xl,yl,zl);
```

This will integrate the function $f(x) = x * y * z$ over 3 sets of limits, since *zl* is defined to be a 2×3 matrix.

SOURCE integral.src

GLOBALS **_intord**, **_intq12**, **_intq16**, **_intq2**, **_intq20**, **_intq24**, **_intq3**, **_intq32**, **_intq4**, **_intq40**, **_intq6**, **_intq8**, **_intrec**

SEE ALSO **intquad1**, **intquad2**, **intsimp**, **intgrat2**, **intgrat3**

intrleav

PURPOSE Interleaves the rows of two files that have been sorted on a common variable to produce a single file sorted on that variable.

FORMAT **intrleav(***infile1***,***infile2***,***outfile***,***keyvar***,***keytyp***);**

INPUT

infile1	string, name of input file 1.
infile2	string, name of input file 2.
outfile	string, name of output file.
keyvar	string, name of key variable; this is the column the files are sorted on.
keytyp	scalar, data type of key variable.
	1 numeric key, ascending order

2 character key, ascending order
-1 numeric key, descending order
-2 character key, descending order

REMARKS The two files MUST have exactly the same variables, that is, the same number of columns AND the same variable names. They must both already be sorted on the key column. This procedure will combine them into one large file, sorted by the key variable.

If the inputs are null ("" or 0), the procedure will ask for them.

SOURCE `sortd.src`

SEE ALSO **`intrleavsa`**

intrleavsa

PURPOSE Interleaves the rows of two string arrays that have been sorted on a common column.

FORMAT *y* **`= intrleavsa(`***sa1*`,`*sa2*`,`*ikey*`);`**

INPUT

sa1	N×K string array 1.
sa2	M×K string array 2.
ikey	scalar integer, index of the key column the string arrays are sorted on.

OUTPUT

y	L×K interleaved (combined) string array.

REMARKS The two string arrays MUST have exactly the same number of columns AND have been already sorted on a key column.

This procedure will combine them into one large string array, sorted by the key column.

SOURCE sortd.src

SEE ALSO **intrleav**

intrsect

PURPOSE Returns the intersection of two vectors, with duplicates removed.

FORMAT *y* = **intrsect**(*v1*,*v2*,*flag*);

INPUT

v1	N×1 vector.
v2	M×1 vector.
flag	scalar, if 1, *v1* and *v2* are numeric; if 0, character.

OUTPUT

y	L×1 vector containing all unique values that are in both *v1* and *v2*, sorted in ascending order.

REMARKS Place smaller vector first for fastest operation.

If there are a lot of duplicates within a vector, it is faster to remove them with **unique** before calling **intrsect**.

SOURCE intrsect.src

EXAMPLE

```
v1 = { 3, 9, 5, 2, 10, 15 };
v2 = { 4, 9, 8, 5, 12, 3, 1 };
y = intrsect(v1,v2,1);
```

```
      3
y =   5
      9
```

SEE ALSO **intrsectsa**

intrsectsa

PURPOSE Returns the intersection of two string vectors, with duplicates removed.

FORMAT *y* **= intrsectsa(***sv1*,*sv2***);**

INPUT
sv1 N×1 or 1×N string vector.
sv2 M×1 or 1×M string vector.

OUTPUT
sy L×1 vector containing all unique strings that are in both *sv1* and *sv2*, sorted in ascending order.

REMARKS Place smaller vector first for fastest operation.

If there are a lot of duplicates it is faster to remove them with unique before calling intrsectsa.

EXAMPLE
```
string sv1 = { "mary", "jane", "linda", "dawn" };
string sv2 = { "mary", "sally", "jane", "lisa", "ruth" };
sy = intrsectsa(sv1,sv2);
```

SOURCE intrsect.src

SEE ALSO **intrsect**

intsimp

PURPOSE Integrates a specified function using Simpson's method with end correction. A single integral is computed in one function call.

FORMAT *y* = **intsimp(&*f*,*xl*,*tol*);**

INPUT

&*f*	pointer to the procedure containing the function to be integrated.
xl	2×1 vector, the limits of *x*. The first element is the upper limit and the second element is the lower limit.
tol	The tolerance to be used in testing for convergence.

OUTPUT

y	The estimated integral of $f(x)$ between $xl[1]$ and $xl[2]$.

EXAMPLE

```
proc f(x);
   retp(sin(x));
endp;

let xl = { 1,
           0 };

y = intsimp(&f,xl,1E-8);
```

```
     y = 0.45969769
```

This will integrate the function between 0 and 1.

SOURCE intsimp.src

SEE ALSO **intquad1**, **intquad2**, **intquad3**, **intgrat2**, **intgrat3**

inv, invpd

PURPOSE **inv** returns the inverse of an invertible matrix.

invpd returns the inverse of a symmetric, positive definite matrix.

FORMAT `y = inv(x);`

`y = invpd(x);`

INPUT *x* N×N matrix or K-dimensional array where the last two dimensions are N×N.

OUTPUT *y* N×N matrix or K-dimensional array where the last two dimensions are N×N, containing the inverse of *x*.

REMARKS *x* can be any legitimate expression that returns a matrix or array that is legal for the function.

If *x* is an array, the result will be an array containing the inverses of each 2-dimensional array described by the two trailing dimensions of *x*. In other words, for a 10×4×4 array, the result will be an array of the same size containing the inverses of each of the 10 4×4 arrays contained in *x*

For **inv**, if *x* is a matrix, it must be square and invertible. Otherwise, if *x* is an array, the 2-dimensional arrays described by the last two dimensions of *x* must be square and invertible.

For **invpd**, if *x* is a matrix, it must be symmetric and positive definite. Otherwise, if *x* is an array, the 2-dimensional arrays described by the last two dimensions of *x* must be symmetric and positive definite.

If the input matrix is not invertible by these functions, they will either terminate the program with an error message or return an error code which can be tested for with the **scalerr** function. This depends on the **trap** state as follows:

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 ones (to create a constant term). The true coefficients are specified, and the dependent variable **y** is created. Then the **ols** coefficient estimates are computed.

invswp

PURPOSE Computes a generalized sweep inverse.

FORMAT *y* **= invswp(***x***);**

INPUT *x* N×N matrix.

OUTPUT *y* N×N matrix, the generalized inverse of *x*.

REMARKS This will invert any general matrix. That is, even matrices which will not invert using **inv** because they are singular will invert using **invswp**.

x and *y* will satisfy the two conditions:

1. $xyx = x$
2. $yxy = y$

invswp returns a row and column with zeros when the pivot fails. This is good for quadratic forms since it essentially removes rows with redundant information, i.e., the statistices generated will be "correct" but with reduced degrees of freedom.

The tolerance used to determine if a pivot element is zero is taken from the **crout** singularity tolerance. The corresponding row and column are zeroed out. See SINGULARITY TOLERANCE, Appendix C.

EXAMPLE
```
let x[3,3] = 1 2 3 4 5 6 7 8 9;
y = invswp(x);
```

```
      −1.6666667    0.66666667   0.0000000
y =    1.3333333   −0.33333333   0.0000000
       0.0000000    0.0000000    0.0000000
```

iscplx

PURPOSE Returns whether a matrix or N-dimensional array is complex or real.

FORMAT *y* **= iscplx(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* scalar, 1 if *x* is complex, 0 if it is real.

EXAMPLE
```
x = { 1, 2i, 3 };
y = iscplx(x);
```

```
y = 1.0000000
```

SEE ALSO **hasimag**, **iscplxf**

iscplxf

PURPOSE Returns whether a data set is complex or real.

FORMAT *y* **= iscplxf(***fh***);**

INPUT *fh* scalar, file handle of an open file.

OUTPUT *y* scalar, 1 if the data set is complex, 0 if it is real.

SEE ALSO **hasimag**, **iscplx**

isden

PURPOSE Returns whether a scalar, matrix or N-dimensional array contains denormals.

FORMAT *y* **= isden(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* scalar, 1 if *x* contains a denormal, 0 if it does not.

EXAMPLE
```
x = { 1, exp(-724.5), 3 };
y = isden(x);
```

```
y = 1.0000000
```

SEE ALSO **denToZero**

isinfnanmiss

PURPOSE Returns true if the argument contains an infinity, NaN, or missing value.

FORMAT *y* **= isinfnanmiss(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* scalar, 1 if *x* contains any infinities, NaNs, or missing values, else 0.

SEE ALSO **scalinfnanmiss**, **ismiss**, **scalmiss**

ismiss

PURPOSE Returns a 1 if its matrix argument contains any missing values, otherwise returns a 0.

FORMAT *y* **= ismiss(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* scalar, 1 if *x* contains any missing values, otherwise 0.

REMARKS An element of *x* is considered to be a missing if and only if it contains a missing value in the real part. Thus, if *x* = 1 + .i, **ismiss(*x*)** will return a 0.

EXAMPLE
```
x = { 1 6 3 4 };
y = ismiss(x);
```

```
y = 0.0000000
```

SEE ALSO **scalmiss**, **miss**, **missrv**

keep (dataloop)

PURPOSE Specifies columns (variables) to be saved to the output data set in a data loop.

FORMAT **keep** *variable_list*;

REMARKS Commas are optional in *variable_list*.

Retains only the specified variables in the output data set. Any variables referenced must already exist, either as elements of the source data set, or as the result of a previous **make**, **vector**, or **code** statement.

If neither **keep** nor **drop** is used, the output data set will contain all variables from the source data set, as well as any newly defined variables. The effects of multiple **keep** and **drop** statements are cumulative.

EXAMPLE
```
keep age, pay, sex;
```

SEE ALSO **drop (dataloop)**

key

PURPOSE Returns the ASCII value of the next key available in the keyboard buffer.

FORMAT *y* = **key;**

OUTPUT *y* scalar, ASCII value of next available key in keyboard buffer.

REMARKS If you are workng in terminal mode, **key** does not "see" any keystrokes until ENTER is pressed. The value returned will be zero if no key is available in the buffer or it will equal the ASCII value of the key if one is available. The key is taken from the buffer at this time and the next call to **key** will return the next key.

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	CURSOR LEFT
1077	CURSOR RIGHT
1079	END
1080	CURSOR DOWN
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+CURSOR LEFT
1116	CTRL+CURSOR RIGHT
1117	CTRL+END
1118	CTRL+PAGE DOWN
1119	CTRL+HOME
1120-1131	ALT+1,2,3,4,5,6,7,8,9,0,-,=
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 escape key (ASCII 27) is pressed.

SEE ALSO **vals**, **chrs**, **upper**, **lower**, **con**, **cons**

keyav

PURPOSE Check if keystroke is available.

FORMAT *x* **= keyav;**

OUTPUT *x* scalar, value of key or 0 if no key is available.

SEE ALSO **keyw**, **key**

keyw

PURPOSE Waits for and gets a key.

FORMAT *k* **= keyw;**

OUTPUT *k* scalar, ASCII value of the key pressed.

REMARKS If you are working in terminal mode, **GAUSS** will not see any input until you press the ENTER key. **keyw** gets the next key from the keyboard buffer. If the keyboard buffer is empty, **keyw** waits for a keystroke. For normal keys, **keyw** returns the ASCII value of the key. See **key** for a table of return values for extended and function keys.

SEE ALSO **key**

keyword

PURPOSE Begins the definition of a keyword procedure. Keywords are user-defined functions with local or global variables.

FORMAT **keyword** *name*(*str*);

INPUT

name literal, name of the keyword. This name will be a global symbol.

str string, a name to be used inside the keyword to refer to the argument that is passed to the keyword when the keyword is called. This will always be local to the keyword, and cannot be accessed from outside the keyword or from other keywords or procedures.

REMARKS A keyword definition begins with the **keyword** statement and ends with the **endp** statement. See PROCEDURES AND KEYWORDS, Chapter 12.

Keywords always have 1 string argument and 0 returns. **GAUSS** will take everything past *name*, excluding leading spaces, and pass it as a string argument to the keyword. Inside the keyword, the argument is a local string. The user is responsible to manipulate or parse the string.

An example of a keyword definition is:

```
keyword add(str);
    local tok,sum;
    sum = 0;
    do until str $=\,= "";
         { tok, str } = token(str);
         sum = sum + stof(tok);
    endo;
    print "Sum is: " sum;
endp;
```

To use this keyword, type:

```
add 1 2 3 4 5;
```

This keyword will respond by printing:

```
Sum is: 15
```

SEE ALSO **proc**, **local**, **endp**

lag (dataloop)

PURPOSE Lags variables a specified number of periods.

FORMAT **lag** *nv1* = *var1*:*p1* [[*nv2* = *var2*:*p2...*]];

INPUT

var	name of the variable to lag.
p	scalar constant, number of periods to lag.

OUTPUT

nv	name of the new lagged variable.

REMARKS You can specify any number of variables to lag. Each variable can be lagged a different number of periods. Both positive and negative lags are allowed.

Lagging is executed before any other transformations. If the new variable name is different from that of the variable to lag, the new variable is first created and appended to a temporary data set. This temporary data set becomes the input data set for the dataloop, and is then automatically deleted.

lag1

PURPOSE Lags a matrix by one time period for time series analysis.

FORMAT *y* = **lag1**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* N×K 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**

lagn

PURPOSE Lags a matrix a specified number of time periods for time series analysis.

FORMAT *y* = **lagn**(*x*,*t*);

INPUT *x* N×K matrix.

t scalar, number of time periods.

OUTPUT *y* N×K matrix, *x* lagged *t* periods.

REMARKS If *t* is positive, **lagn** lags *x* back *t* time periods, so the first *t* observations of *y* are missing. If *t* is negative, **lagn** lags *x* forward *t* time periods, so the last *t* observations of *y* are missing.

SOURCE lag.src

SEE ALSO **lag1**

lapeighb

PURPOSE Computes eigenvalues only of a real symmetric or complex Hermitian matrix selected by bounds.

FORMAT *ve* = **lapeighb**(*x*,*vl*,*vu*);

INPUT

x	N×N matrix, real symmetric or complex Hermitian.
vl	scalar, lower bound of the interval to be searched for eigenvalues.
vu	scalar, upper bound of the interval to be searched for eigenvalues; *vu* must be greater than *vl*.
abstol	scalar, the absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to *abstol* + EPS*max(| *a*|,|*b*|), where EPS is machine precision. If *abstol* is less than or equal to zero, then EPS*||*T*|| will be used in its place, where *T* is the tridiagonal matrix obtained by reducing the input matrix to tridiagonal form.

OUTPUT

ve	M×1 vector, eigenvalues, where M is the number of eigenvalues on the half open interval [*vl*,*vu*]. If no eigenvalues are found then *ve* is a scalar missing value.

REMARKS **lapeighb** computes eigenvalues only which are found on on the half open interval [*vl*,*vu*]. To find eigenvalues within a specified range of indices see **lapeighi**. For eigenvectors see **lapeighvi**, or **lapeighvb**. **lapeighb** is based on the LAPACK drivers DYESVX and ZHEEVX. Further documentation of these functions may be found in the LAPACK User's Guide.

EXAMPLE

```
x = { 5   2   1,
      2   6   2,
      1   2   9 };

vl = 5;
vu = 10;
ve = lapeighi(x,vl,vu,0);
print ve;

6.0000
```

SEE ALSO **lapeighb**, **lapeighvi**, **lapeighvb**

lapeighi

PURPOSE Computes eigenvalues only of a real symmetric or complex Hermitian matrix selected by index.

FORMAT *ve* **= lapeighi(***x*,*il*,*iu*,*abstol***);**

INPUT

x	N×N matrix, real symmetric or complex Hermitian.
il	scalar, index of the smallest desired eigenvalue ranking them from smallest to largest.
iu	scalar, index of the largest desired eigenvalue, *iu* must be greater than *il*.
abstol	scalar, the absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to *abstol* + EPS*max(|*a*|,|*b*|), where EPS is machine precision. If *abstol* is less than or equal to zero, then EPS*||*T*|| will be used in its place, where *T* is the tridiagonal matrix obtained by reducing the input matrix to tridiagonal form.

OUTPUT *ve* (*iu*-*il*+1)×1 vector, eigenvalues.

REMARKS **lapeighi** computes *iu*-*il*+1 eigenvalues only given a range of indices, i.e., the i^{th} to j^{th} eigenvalues, ranking them from smallest to largest. To find eigenvalues within a specified range see **lapeighxb**. For eigenvectors see **lapeighvi**, or **lapeighvb**. **lapeighi** is based on the LAPACK drivers DYESVX and ZHEEVX. Further documentation of these functions may be found in the LAPACK User's Guide.

EXAMPLE
```
x = { 5   2   1,
      2   6   2,
      1   2   9 };

il = 2;
iu = 3;
ve = lapeighi(x,il,iu,0);
print ve;

6.0000 10.6056
```

SEE ALSO **lapeighb**, **lapeighvi**, **lapeighvb**

lapeighvb

PURPOSE Computes eigenvalues and eigenvectors of a real symmetric or complex Hermitian matrix selected by bounds.

FORMAT { *ve*,*va* } = **lapeighvb**(*x*,*vl*,*vu*,*abstol*);

INPUT

x	N×N matrix, real symmetric or complex Hermitian.
vl	scalar, lower bound of the interval to be searched for eigenvalues.
vu	scalar, upper bound of the interval to be searched for eigenvalues; *vu* must be greater than *vl*.

abstol scalar, the absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to *abstol* + EPS*max(|*a*|,|*b*|), where EPS is machine precision. If *abstol* is less than or equal to zero, then EPS*||*T*|| will be used in its place, where *T* is the tridiagonal matrix obtained by reducing the input matrix to tridiagonal form.

OUTPUT

ve M×1 vector, eigenvalues, where M is the number of eigenvalues on the half open interval [*vl*,*vu*]. If no eigenvalues are found then s is a scalar missing value.

va N×M matrix, eigenvectors.

REMARKS **lapeighvb** computes eigenvalues and eigenvectors which are found on the half open interval [*vl*,*vu*]. **lapeighvb** is based on the LAPACK drivers DYESVX and ZHEEVX. Further documentation of these functions may be found in the LAPACK User's Guide.

EXAMPLE

```
x = { 5   2   1,
      2   6   2,
      1   2   9 };

vl = 5;
vu = 10;
{ ve,va } = lapeighvb(x,vl,vu,0);
print ve;

      6.00000000

print va;

     -0.57735027
      0.00000000
      0.00000000
```

SEE ALSO **lapeighvb**

lapeighvi

PURPOSE Computes selected eigenvalues and eigenvectors of a real symmetric or complex Hermitian matrix.

FORMAT { *ve*,*va* } = **lapeighvi**(*x*,*il*,*iu*,*abstol*);

INPUT

x	N×N matrix, real symmetric or complex Hermitian.
il	scalar, index of the smallest desired eigenvalue ranking them from smallest to largest.
iu	scalar, index of the largest desired eigenvalue, *iu* must be greater than *il*.
abstol	scalar, the absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [*a*,*b*] of width less than or equal to *abstol* + EPS*max(|*a*|,|*b*|), where EPS is machine precision. If *abstol* is less than or equal to zero, then EPS*||*T*|| will be used in its place, where *T* is the tridiagonal matrix obtained by reducing the input matrix to tridiagonal form.

OUTPUT

ve	(*iu*-*il*+1)×1 vector, eigenvalues.
va	N×(*iu*-*il*+1) matrix, eigenvectors.

REMARKS **lapeighvi** computes *iu*-*il*+1 eigenvalues and eigenvectors given a range of indices, i.e., the i^{th} to j^{th} eigenvalues, ranking them from smallest to largest. To find eigenvalues and eigenvectors within a specified range see **lapeighvb**. **lapeighvi** is based on the LAPACK drivers DYESVX and ZHEEVX. Further documentation of these functions may be found in the LAPACK User's Guide.

EXAMPLE

```
x = { 5   2   1,
      2   6   2,
      1   2   9 };
```

```
il = 2;
iu = 3;
{ ve,va } = lapeighvi(x,il,iu,0);
print ve;

      6.00000000
     10.60555128

print va;

     -0.57735027      -0.57735027
     -0.31970025      -0.49079864
      0.00000000       0.00000000
```

SEE ALSO **lapeighvb**, **lapeighb**

lapgeig

PURPOSE Computes generalized eigenvalues for a pair of real or complex general matrices.

FORMAT { *va1*,*va2* } = **lapgeig**(*A*,*B*);

INPUT

A	N×N matrix, real or complex general matrix.
B	N×N matrix, real or complex general matrix.

OUTPUT

va1	N×1 vector, numerator of eigenvalues.
va2	N×1 vector, denominator of eigenvalues.

REMARKS *va1* and *va2* are the vectors of the numerators and denominators respectively of the eigenvalues of the solution of the generalized symmetric eigenproblem of the form $Aw = eBw$ where A and B are real or complex general matrices and w = *va1 ./ va2*. The generalized eigenvalues are not computed directly because

some elements of *va2* may be zero, i.e., the eigenvalues may be infinite. This procedure calls the LAPACK routines DGEGV and ZGEGV.

SEE ALSO **lapgeig**, **lapgeigh**

lapgeigh

PURPOSE Computes generalized eigenvalues for a pair of real symmetric or Hermitian matrices.

FORMAT *ve* **= lapgeigh(***A*,*B***);**

INPUT

A	N×N matrix, real or complex symmetric or Hermitian matrix.
B	N×N matrix, real or complex positive definite symmetric or Hermitian matrix.

OUTPUT

ve	N×1 vector, eigenvalues.

REMARKS *ve* is the vector of eigenvalues of the solution of the generalized symmetric eigenproblem of the form $Ax = \lambda Bx$.

EXAMPLE

```
A = { 3  4  5,
      2  5  2,
      3  2  4 };

B = { 4  2  2,
      2  6  1,
      2  1  8 };

ve = lapgeigh(A,B);

print ve;
```

```
-0.70051730
 0.48661989
 1.27818313
```

This procedure calls the LAPACK routines DSYGV and ZHEGV.

SEE ALSO **lapgeig**, **lapgeighv**

lapgeighv

PURPOSE Computes generalized eigenvalues and eigenvectors for a pair of real symmetric or Hermitian matrices.

FORMAT { *ve*,*va* } **= lapgeighv(***A*,*B***);**

INPUT

A	N×N matrix, real or complex symmetric or Hermitian matrix.
B	N×N matrix, real or complex positive definite symmetric or Hermitian matrix.

OUTPUT

ve	N×1 vector, eigenvalues.
va	N×N matrix, eigenvectors.

REMARKS *ve* and *va* are the eigenvalues and eigenvectors of the solution of the generalized symmetric eigenproblem of the form $Ax = \lambda Bx$. Equivalently, *va* diagonalizes $U'^{-1}AU^{-1}$ in the following way

$$vaU'^{-1}AY^{-1}va' = ve$$

where $B = U'U$. This procedure calls the LAPACK routines DSYGV and ZHEGV.

EXAMPLE

```
A = { 3  4  5,
```

```
      2  5  2,
      3  2  4 };

B = { 4  2  2,
      2  6  1,
      2  1  8 };

{ ve, va } = lapgeighv(A,B);

print ve;

    -0.0425
     0.5082
     0.8694

print va;

     0.3575   -0.0996   0.9286
    -0.2594    0.9446   0.2012
    -0.8972   -0.3128   0.3118
```

SEE ALSO **lapgeig**, **lapgeigh**

lapgeigv

PURPOSE Computes generalized eigenvalues, left eigenvectors, and right eigenvectors for a pair of real or complex general matrices.

FORMAT { *va1*,*va2*,*lve*,*rve* } = **lapgeigv**(*A*,*B*);

INPUT

A	N×N matrix, real or complex general matrix.
B	N×N matrix, real or complex general matrix.

OUTPUT

va1	N×1 vector, numerator of eigenvalues.

	va2	N×1 vector, denominator of eigenvalues.
	lve	N×N left eigenvectors.
	rve	N×N right eigenvectors.

REMARKS *va1* and *va2* are the vectors of the numerators and denominators respectively of the eigenvalues of the solution of the generalized symmetric eigenproblem of the form $Aw = \lambda Bw$ where A and B are real or complex general matrices and $w = va1./va2$. The generalized eigenvalues are not computed directly because some elements of *va2* may be zero, i.e., the eigenvalues may be infinite.

The left and right eigenvectors diagonalize $U'^{-1}AU^{-1}$ where $B = U'U$, that is,

$$lve\ U'^{-1}AU\ lve' = w$$

and

$$rve'U'^{-1}AU^{-1}rve = w$$

This procedure calls the LAPACK routines DGEGV and ZGEGV.

SEE ALSO **lapgeig**, **lapgeigh**

lapgsvdcst

PURPOSE Compute the generalized singular value decomposition of a pair of real or complex general matrices.

FORMAT { *C*,*S*,*R*,*U*,*V*,*Q* } = **lapgsvdcst**(*A*,*B*);

INPUT	*A*	M×N matrix.

	B	P×N matrix.
OUTPUT	*C*	L×1 vector, singular values for *A*.
	S	L×1 vector, singular values for *B*.
	R	(K+L)×(K+L) upper triangular matrix.
	U	M×M matrix, orthogonal transformation matrix.
	V	P×P matrix, orthogonal transformation matrix.
	Q	N×N matrix, orthogonal transformation matrix.

REMARKS (1) The generalized singular value decomposition of *A* and *B* is

$$U'AQ = D_1Z$$

$$V'BQ = D_2Z$$

where *U*, *V*, and *Q* are orthogonal matrices (see **lapgsvdcst** and **lapgsvdst**). Letting K+L = the rank of *A*|*B* then *R* is a (K+L)×(K+L) upper triangular matrix, *D1* and *D2* are M×(K+L) and P×(K+L) matrices with entries on the diagonal, *Z* = [0*R*], and if M-K-L >= 0

$$D1 = \begin{array}{c} \\ K \\ L \\ M-K-L \end{array} \begin{array}{c} \begin{array}{ccc} & K & L \end{array} \\ \begin{bmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{bmatrix} \end{array}$$

$$D2 = \begin{array}{c} \\ P \\ P-L \end{array} \begin{array}{c} \begin{array}{cc} K & L \end{array} \\ \begin{bmatrix} 0 & S \\ 0 & 0 \end{bmatrix} \end{array}$$

$$\begin{bmatrix} 0 & R \end{bmatrix} = \begin{array}{c} \\ K \\ L \end{array} \begin{array}{c} \begin{array}{ccc} N-K-L & K & L \end{array} \\ \begin{bmatrix} 0 & R11 & R12 \\ 0 & 0 & R22 \end{bmatrix} \end{array}$$

or if M-K-L < 0

$$D1 = \begin{array}{c} \\ K \\ M-K \end{array} \begin{array}{c} \begin{array}{cccc} K & M-K & K+L-M & \end{array} \\ \begin{bmatrix} I & 0 & 0 & \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 & \end{bmatrix} \end{array}$$

$$D2 = \begin{array}{c} \\ M-K \\ K+L-M \\ P-L \end{array} \begin{array}{c} \begin{array}{ccc} K & M-K & K+L-M \end{array} \\ \begin{bmatrix} 0 & S & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & I \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$\begin{bmatrix} 0 & R \end{bmatrix} = \begin{array}{c} \\ K \\ M-K \\ K+L-M \end{array} \begin{array}{c} \begin{array}{cccc} N-K-L & K & M-K & K+L-M \end{array} \\ \begin{bmatrix} 0 & R11 & R12 & R13 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & R22 & R23 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 & R33 \end{bmatrix} \end{array}$$

(2) Form the matrix

$$X = Q \begin{bmatrix} I & 0 \\ 0 & R^{-1} \end{bmatrix}$$

then

$$A = U'^{-1} E_1 X$$

$$B = V'^{-1} E_2 X^{-1}$$

where $E1 = \begin{bmatrix} 0 & D1 \end{bmatrix}$ and $E2 = \begin{bmatrix} 0 & D2 \end{bmatrix}$.

(3) The generalized singular value decomposition of A and B implicitly

produces the singular value decomposition of AB^{-1}:

$$AB^{-1} = UD_1D_2^{-1}V'$$

This procedure calls the LAPACK routines DGGSVD and ZGGSVD.

SEE ALSO **lapgsvds**, **lapgsvdst**

lapgsvds

PURPOSE Compute the generalized singular value decomposition of a pair of real or complex general matrices.

FORMAT { *C*,*S*,*R* } = **lapgsvds**(*A*,*B*);

INPUT

A	M×N real or complex matrix.
B	P×N real or complex matrix.

OUTPUT

C	L×1 vector, singular values for *A*.
S	L×1 vector, singular values for *B*.
R	(K+L)×(K+L) upper triangular matrix.

REMARKS (1) The generalized singular value decomposition of *A* and *B* is

$$U'AQ = D_1Z$$

$$V'BQ = D_2Z$$

where *U*, *V*, and *Q* are orthogonal matrices (see **lapgsvdcst** and **lapgsvdst**). Letting K+L = the rank of *A*|*B* then *R* is a (K+L)×(K+L) upper triangular

matrix, D_1 and D_2 are M×(K+L) and P×(K+L) matrices with entries on the diagonal, Z = [0R], and if M-K-L >= 0

$$D1 = \begin{array}{c} \\ K \\ L \\ M-K-L \end{array} \begin{array}{c} \begin{array}{ccc} K & L & \end{array} \\ \begin{bmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{bmatrix} \end{array}$$

$$D2 = \begin{array}{c} \\ P \\ P-L \end{array} \begin{array}{c} \begin{array}{cc} K & L \end{array} \\ \begin{bmatrix} 0 & S \\ 0 & 0 \end{bmatrix} \end{array}$$

$$[0R] = \begin{array}{c} \\ K \\ L \end{array} \begin{array}{c} \begin{array}{ccc} N-K-L & K & L \end{array} \\ \begin{bmatrix} 0 & R11 & R12 \\ 0 & 0 & R22 \end{bmatrix} \end{array}$$

or if M-K-L < 0

$$D1 = \begin{array}{c} \\ K \\ M-K \end{array} \begin{array}{c} \begin{array}{ccc} K & M-K & K+L-M \end{array} \\ \begin{bmatrix} I & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$D2 = \begin{array}{c} \\ M-K \\ K+L-M \\ P-L \end{array} \begin{array}{c} \begin{array}{ccc} K & M-K & K+L-M \end{array} \\ \begin{bmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$[0R] = \begin{array}{c} \\ K \\ M-K \\ K+L-M \end{array} \begin{array}{c} \begin{array}{cccc} N-K-L & K & M-K & K+L-M \end{array} \\ \begin{bmatrix} 0 & R11 & R12 & R13 \\ 0 & 0 & R22 & R23 \\ 0 & 0 & 0 & R33 \end{bmatrix} \end{array}$$

(2) Form the matrix

$$X = Q \begin{bmatrix} I & 0 \\ 0 & R^{-1} \end{bmatrix}$$

then

$$A = U'^{-1} E_1 X$$

$$B = V'^{-1} E_2 X^{-1}$$

where $E1 = \begin{bmatrix} 0 & D1 \end{bmatrix}$ and $E2 = \begin{bmatrix} 0 & D2 \end{bmatrix}$.

(3) The generalized singular value decomposition of A and B implicitly produces the singular value decomposition of AB^{-1}:

$$AB^{-1} = U D_1 D_2^{-1} V'$$

This procedure calls the LAPACK routines DGGSVD and ZGGSVD.

SEE ALSO **lapgsvdcst**, **lapgsvdst**

lapgsvdst

PURPOSE Compute the generalized singular value decomposition of a pair of real or complex general matrices.

FORMAT { *D1*,*D2*,*Z*,*U*,*V*,*Q* } = **lapgsvdst**(*A*,*B*);

INPUT

A	M×N matrix.
B	P×N matrix.

OUTPUT

D1	M×(K+L) matrix, with singular values for *A* on diagonal.
D2	P×(K+L) matrix, with singular values for *B* on diagonal.
Z	(K+L)×N matrix, partitioned matrix composed of a zero matrix and upper triangular matrix.
U	M×M matrix, orthogonal transformation matrix.
V	P×P matrix, orthogonal transformation matrix.
Q	N×N matrix, orthogonal transformation matrix.

REMARKS (1) The generalized singular value decomposition of *A* and *B* is

$$U'AQ = D1Z$$

$$V'BQ = D2Z$$

where *U*, *V*, and *Q* are orthogonal matrices (see **lapgsvdcst** and **lapgsvdst)**. Letting K+L = the rank of *A*|*B* then *R* is a (K+L)×(K+L) upper triangular matrix, *D1* and *D2* are M×(K+L) and P×(K+L) matrices with entries on the diagonal, $Z = [0R]$, and if M-K-L >= 0

$$D1 = \begin{array}{c} \\ K \\ L \\ M-K-L \end{array} \begin{array}{c} \begin{array}{ccc} K & L & \end{array} \\ \begin{bmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{bmatrix} \end{array}$$

$$D2 = \begin{array}{c} \\ P \\ P-L \end{array} \begin{array}{c} \begin{array}{cc} K & L \end{array} \\ \begin{bmatrix} 0 & S \\ 0 & 0 \end{bmatrix} \end{array}$$

$$\begin{array}{cc} & \begin{matrix} N-K-L & K & L \end{matrix} \\ \begin{matrix} [\ 0 \ \ R \ ] \ = \\ \\ \end{matrix} & \begin{matrix} K \\ L \end{matrix} \begin{bmatrix} 0 & R11 & R12 \\ 0 & 0 & R22 \end{bmatrix} \end{array}$$

or if M-K-L < 0

$$D1 = \begin{array}{c} \\ K \\ M-K \end{array} \begin{array}{c} \begin{matrix} K & M-K & K+L-M \end{matrix} \\ \begin{bmatrix} I & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$D2 = \begin{array}{c} \\ M-K \\ K+L-M \\ P-L \end{array} \begin{array}{c} \begin{matrix} K & M-K & K+L-M \end{matrix} \\ \begin{bmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$\begin{array}{cc} & \begin{matrix} N-K-L & K & M-K & K+L-M \end{matrix} \\ \begin{matrix} [\ 0 \ \ R \ ] \ = \\ \\ \\ \end{matrix} & \begin{matrix} K \\ M-K \\ K+L-M \end{matrix} \begin{bmatrix} 0 & R11 & R12 & R13 \\ 0 & 0 & R22 & R23 \\ 0 & 0 & 0 & R33 \end{bmatrix} \end{array}$$

(2) Form the matrix

$$X = Q \begin{bmatrix} I & 0 \\ 0 & R^{-1} \end{bmatrix}$$

then

$$A = U'^{-1} E_1 X$$

$$B = V'^{-1} E_2 X^{-1}$$

where $E1 = \begin{bmatrix} 0 & D1 \end{bmatrix}$ and $E2 = \begin{bmatrix} 0 & D2 \end{bmatrix}$.

(3) The generalized singular value decomposition of A and B implicitly produces the singular value decomposition of AB^{-1}:

$$AB^{-1} = UD_1D_2^{-1}V'$$

This procedure calls the LAPACK routines DGGSVD and ZGGSVD.

SEE ALSO **lapgsvds**, **lapgsvdcst**

lapgschur

PURPOSE Compute the generalized Schur form of a pair of real or complex general matrices.

FORMAT { *sa*,*sb*,*q*,*z* } = **lapgschur**(*A*,*B*);

INPUT

A	N×N matrix, real or complex general matrix.
B	N×N matrix, real or complex general matrix.

OUTPUT

sa	N×N matrix, Schur form of *A*.
sb	N×N matrix, Schur form of *B*.
q	N×N matrix, left Schur vectors.
z	N×N matrix, right Schur vectors.

REMARKS The pair of matrices A and B are in generalized real Schur form when B is upper triangular with non-negative diagonal, and A is block upper triangular with 1×1 and 2×2 blocks. The 1×1 blocks correspond to real generalized eigenvalues and the 2×2 blocks to pairs of complex conjugate eigenvalues. The real generalized eigenvalues can be computed by dividing the diagonal element of *sa* by the

corresponding diagonal element of *sb*. The complex generalized eigenvalues are computed by first constructing two complex conjugate numbers from 2×2 block where the real parts are on the diagonal of the block and the imaginary part on the off-diagonal. The eigenvalues are then computed by dividing the two complex conjugate values by their corresponding diagonal elements of *sb*. The generalized Schur vectors *q* and *z* are orthogonal matrices that reduce *A* and *B* to Schur form:

$$sa = q'Az$$

$$sb = q'Bz$$

This procedure calls the LAPACK routines DGEGS and ZGEGS.

SOURCE `lapschur.src`

lapsvdcusv

PURPOSE Computes the singular value decomposition of a real or complex rectangular matrix, returns compact *u* and *v*.

FORMAT { *u*,*s*,*v* } = **lapsvdcusv**(*x*);

INPUT

x	M×N matrix, real or complex rectangular matrix.

OUTPUT

u	M×min(M,N) matrix, left singular vectors.
s	min(M,N)×N matrix, singular values.
v	N×N matrix, right singular values.

REMARKS **lapsvdcusv** computes the singular value decomposition of a real or complex

rectangular matrix. The SVD is

$$x = usv'$$

where v is the matrix of right singular vectors. **lapsvdcusv** is based on the LAPACK drivers DGESVD and ZGESVD. Further documentation of these functions may be found in the LAPACK User's Guide.

EXAMPLE

```
x = { 2.143 4.345 6.124,
      1.244 5.124 3.412, 0.235 5.657 8.214 };

{ u,s,v }  = lapsvdusv(x);

print s;

      -0.55531277      0.049048431       0.83019394
      -0.43090168       0.83684123      -0.33766923
      -0.71130266      -0.54524400      -0.44357356

print s;

       13.895868        0.0000000        0.0000000
       0.0000000        2.1893939        0.0000000
       0.0000000        0.0000000        1.4344261

print v;

      -0.13624432      -0.62209955      -0.77099263
       0.46497296       0.64704876      -0.60425826
       0.87477862      -0.44081748       0.20110275
```

SEE ALSO **lapsvds**, **lapsvdusv**

lapsvds

PURPOSE Computes the singular values of a real or complex rectangular matrix

FORMAT *s* = **lapsvds**(*x*);

INPUT *x* M×N matrix, real or complex rectangular matrix.

OUTPUT *s* min(M,N)×1 vector, singular values.

REMARKS **lapsvd** computes the singular values of a real or complex rectangular matrix. The SVD is

$$x = usv'$$

where *v* is the matrix of right singular vectors. For the computation of the singular vectors, see **lapsvdcusv** and **lapsvdusv**.

lapsvd is based on the LAPACK drivers DGESVD and ZGESVD. Further documentation of these functions may be found in the LAPACK User's Guide.

EXAMPLE

```
x = { 2.143 4.345 6.124,
      1.244 5.124 3.412,
      0.235 5.657 8.214 };

va = lapsvd(x);
print va;

13.895868 2.1893939 1.4344261

xi = { 4+1 3+1 2+2,
       1+2 5+3 2+2,
       1+1 2+1 6+2 };
```

```
ve = lapsvds(xi);
print ve;

10.352877 4.0190557 2.3801546
```

SEE ALSO **lapsvdcusv**, **lapsvdusv**

lapsvdusv

PURPOSE Computes the singular value decomposition a real or complex rectangular matrix.

FORMAT { *u*,*s*,*v* } = **lapsvdusv**(*x*);

INPUT

x	M×N matrix, real or complex rectangular matrix.

OUTPUT

u	M×M matrix, left singular vectors.
s	M×N matrix, singular values.
v	N×N matrix, right singular values.

REMARKS **lapsvdusv** computes the singular value decomposition of a real or complex rectangular matrix. The SVD is

$$x = usv'$$

where *v* is the matrix of right singular vectors. **lapsvdusv** is based on the LAPACK drivers DGESVD and ZGESVD. Further documentation of these functions may be found in the LAPACK User's Guide.

EXAMPLE

```
x = { 2.143 4.345 6.124,
```

```
      1.244 5.124 3.412,
      0.235 5.657 8.214 };

{ u,s,v } = lapsvdusv(x);

print u;

-0.5553  0.0490  0.8302
-0.4309  0.8368 -0.3377
-0.7113 -0.5452 -0.4436

print s;

13.8959  0.0000  0.0000
 0.0000  2.1894  0.0000
 0.0000  0.0000  1.4344

print v;

-0.1362 0.4650  0.8748
 0.6221 0.6470 -0.4408
-0.7710  -0.6043  0.2011
```

SEE ALSO **lapsvds**, **lapsvdcusv**

let

PURPOSE Creates a matrix from a list of numeric or character values. The result is always of type matrix, string, or string array.

FORMAT **let** *x* **=** *constant_list*;

REMARKS Expressions and variable names are not allowed in the **let** command, expressions such as this:

```
let x[2,1] = 3*a b
```

are illegal. To define matrices by combining matrices and expressions, use an expression containing the concatenation operators: ~ and | .

Numbers can be entered in scientific notation. The syntax is *d*E±*n*, where *d* is a number and *n* is an integer (denoting the power of 10):

```
let x = 1e+10 1.1e-4 4.019e+2;
```

Complex numbers can be entered by joining the real and imaginary parts with a sign (+ or -); there should be no spaces between the numbers and the sign. Numbers with no real part can be entered by appending an "`i`" to the number:

```
let x = 1.2+23 8.56i 3-2.1i -4.2e+6i 1.2e-4-4.5e+3i;
```

If curly braces are used, the **`let`** is optional.

```
let x = { 1 2 3, 4 5 6, 7 8 9 };
```

```
x = { 1 2 3, 4 5 6, 7 8 9 };
```

If indices are given, a matrix of that size will be created:

```
let x[2,2] = 1 2 3 4;
```

$$\mathtt{x} = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

If indices are not given, a column vector will be created:

```
let x = 1 2 3 4;
```

```
      1
      2
x =   3
      4
```

You can create matrices with no elements, i.e., “empty matrices” . Just use a set of empty curly braces:

```
x = {};
```

Empty matrices are chiefly used as the starting point for building up a matrix, for example in a **do** loop. See MATRICES, Section 10.6.2, for more information on empty matrices.

Character elements are allowed in a **let** statement:

```
let x = age pay sex;
```

```
      AGE
x =   PAY
      SEX
```

Lowercase elements can be created if quotation marks are used. Note that each element must be quoted.

```
let x = "age" "pay" "sex";
```

```
          age
x =       pay
          sex
```

EXAMPLE

```
let x;
```

```
x = 0
```

```
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
```

```
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
```

```
let string x = { "Median Income", "Country" };
```

```
x =  Median Income
          Country
```

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**.

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 produce 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. The file will have a unique name beginning with `liblst_`.

For more information about the library system, see LIBRARIES, Chapter 18.

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.

Library files are simple ASCII files that you can create with a text editor. Here is an example:

```
/*
```

```
** This is a GAUSS library file.
*/

eig.src
    eig      : proc
    eigsym   : proc
    _eigerr  : matrix
svd.src
    cond     : proc
    pinv     : proc
    rank     : proc
    svd      : proc
    _svdtol  : matrix
```

The lines not indented are the file names. The lines that are indented are the symbols defined in that file. As you can see, a **GAUSS** library is a dictionary of files and the global symbols they contain.

Any line beginning with **/***, ******, or ***/** is considered a comment. Blank lines are okay.

To make the autoloading process more efficient, you can put the full pathname for each file in the library:

```
/gauss/src/eig.src
    eig      : proc
    eigsym   : proc
    _eigerr  : matrix
/gauss/src/svd.src
    cond     : proc
    pinv     : proc
    rank     : proc
    svd      : proc
    _svdtol  : matrix
```

Here's a debugging hint. If your program is acting strange and you suspect it is autoloading the wrong copy of a procedure, use the 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**

#lineson, #linesoff

PURPOSE The **#lineson** command causes **GAUSS** to embed line number and file name records in a program for the purpose of reporting the location where an error occurs. The **#linesoff** command causes **GAUSS** to stop embedding line and file records in a program.

FORMAT
```
#lineson

#linesoff
```

REMARKS In the "lines on" mode, **GAUSS** keeps track of line numbers and file names and reports the location of an error when an execution time error occurs. In the "lines off" mode, **GAUSS** does not keep track of lines and files at execution time. During the compile phase, line numbers and file names will always be given when errors occur in a program stored in a disk file.

It is easier to debug a program when the locations of errors are reported, but this slows down execution. In programs with several scalar operations, the time spent tracking line numbers and file names is most significant.

These commands have no effect on interactive programs (that is, those typed in the window and run from the command line), since there are no line numbers in such programs.

Line number tracking can be turned on and off through the user interface, but the **#lineson** and **#linesoff** commands will override that.

The line numbers and file names given at run-time will reflect the last record encountered in the code. If you have a mixture of procedures that were compiled without line and file records and procedures that were compiled with line and file records, use the **trace** command to locate exactly where the error occurs.

The **Currently active call** error message will always be correct. If it states that it was executing procedure **xyz** at line number **nnn** in file ABC and **xyz** has no line **nnn** or is not in file ABC, you know that it just did not encounter any line or file records in **xyz** before it crashed.

When using **#include**'d files, the line number and file name will be correct for the file the error was in within the limits stated above.

SEE ALSO **trace**

linsolve

PURPOSE Solves $Ax = b$ using the inverse function.

FORMAT x **= linsolve(**b**,**A**);**

INPUT

b	N×K matrix.
A	N×N matrix.

OUTPUT

x	N×K matrix, the linear solution of b/A for each column in b.

REMARKS **linsolve** solves for x by computing **inv(**A**)***b. If A is square and b contains more than 1 column, it is much faster to use **linsolve** than the **/** operator. However, while faster, there is some sacrifice in accuracy.

A test shows **linsolve** to be acccurate to within approximately 1.2e-11, while the **/** operator is accurate to within approximately 4e-13.

EXAMPLE

```
b = { 2, 3, 4 };
```

```
a = { 10 2 3, 6 14 2, 1 1 9 };
x = linsolve(b,A);

print x

     0.045863309
     0.13399281
     0.42446043
```

SEE ALSO **qrsol**, **qrtsol**, **solpd**, **cholsol**

listwise (dataloop)

PURPOSE Controls listwise deletion of missing values.

FORMAT **listwise** [[**read**]|[[**write**]];

REMARKS If **read** is specified, the deletion of all rows containing missing values happens immediately after reading the input file and before any transformations. If **write** is specified, the deletion of missing values happens after any transformations and just before writing to the output file. If no **listwise** statement is present, rows with missing values are not deleted.

The default is **read**.

ln

PURPOSE Computes the natural log of all elements of *x*.

FORMAT *y* = **ln**(*x*);

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array containing the natural log values of the elements of *x*.

REMARKS **ln** is defined for $x \neq 0$.

If *x* is negative, complex results are returned.

You can turn the generation of complex numbers for negative inputs on or off in the **GAUSS** configuration file, and with the **sysstate** function, case 8. If you turn it off, **ln** will generate an error for negative inputs.

If *x* is already complex, the complex number state doesn't matter; **ln** will compute a complex result.

x can be any expression that returns a matrix.

EXAMPLE
```
y = ln(16);
```

```
y = 2.7725887
```

SEE ALSO **log**

lncdfbvn

PURPOSE Computes natural log of bivariate Normal cumulative distribution function.

FORMAT *y* **= lncdfbvn(***x1***,***x2***,***r***);**

INPUT *x1* N×K matrix, abscissae.

x2 L×M matrix, abscissae.

	r	P×Q matrix, correlations.
OUTPUT	*y*	max(N,L,P) × max(K,M,Q) matrix, *ln Pr*($X < x1, X < x2|r$).

REMARKS *x1*, *x2*, and *r* must be E×E conformable.

SOURCE lncdfn.src

SEE ALSO **cdfbvn**, **lncdfmvn**

lncdfbvn2

PURPOSE Returns natural log of standardized bivariate Normal cumulative distribution function of a bounded rectangle.

FORMAT *y* = **lncdfbvn2**(*h*,*dh*,*k*,*dk*,*r*);

INPUT	*h*	N×1 vector, upper limits of integration for variable 1.
	dh	N×1 vector, increments for variable 1.
	k	N×1 vector, upper limits of integration for variable 2.
	dk	N×1 vector, increments for variable 2.
	r	N×1 vector, correlation coefficients between the two variables.
OUTPUT	*y*	N×1 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 N×1 vectors.

lncdfbvn2 will abort if the computed integral is negative.

lncdfbvn2 computes an error estimate for each set of inputs–the real integral is **exp**(*y*)±*err*. The size of the error depends on the input arguments. If **trap 2** is set, a warning message is displayed when *err* **>= exp**(*y*)**/100**.

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

EXAMPLE Example 1

```
lncdfbvn2(1,1,1,1,0.5);
```

produces:

```
-3.2180110258198771e+000
```

Example 2

```
trap 0,2;
lncdfbvn2(1,1e-15,1,1e-15,0.5);
```

produces:

```
-7.1171016046360151e+001
```

Example 3

```
trap 2,2;
lncdfbvn2(1,-1e-45,1,1e-45,0.5);
```

produces:

```
WARNING: Dubious accuracy from lncdfbvn2:
0.000e+000 +/- 2.8e-060
            -INF
```

SEE ALSO **cdfbvn2**, **cdfbvn2e**

lncdfmvn

PURPOSE Computes natural log of multivariate Normal cumulative distribution function.

FORMAT *y* = **lncdfmvn**(*x*,*r*);

INPUT
x K×L matrix, abscissae.
r K×K matrix, correlation matrix.

OUTPUT
y L×1 vector, *ln Pr*($X < x|r$).

REMARKS You can pass more than one set of abscissae at a time; each column of *x* is treated separately.

SOURCE lncdfn.src

SEE ALSO **cdfmvn**, **lncdfbvn**

lncdfn

PURPOSE Computes natural log of Normal cumulative distribution function.

FORMAT *y* = **lncdfn**(*x*);

INPUT
x N×K matrix or N-dimensional array, abscissae.

OUTPUT
y N×K matrix or N-dimensional array, *ln Pr*($X < x$).

SOURCE lncdfn.src

lncdfn2

PURPOSE Computes natural log of interval of Normal cumulative distribution function.

FORMAT y **= lncdfn2(**x**,**r**);**

INPUT
x M×N matrix, abscissae.

r K×L matrix, E×E conformable with x, intervals.

OUTPUT
y max(M,K) × 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:

$\|x\| \le 1$	and	$dx \le 1$	$\pm 1e-14$
$1 < \|x\| < 37$	and	$\|dx\| < 1/\|x\|$	$\pm 1e-13$
$min(x, x+dx) > -37$	and	$y > -690$	$\pm 1e-11$ or better

A relative error of ±1e-14 implies that the answer is accurate to better than ±1 in the 14^{th} digit after the decimal point.

EXAMPLE
```
print lncdfn2(-10,29);

  -7.6198530241605269e-24

print lncdfn2(0,1);

  -1.0748623268620716e+00

print lncdfn2(5,1);

  -1.5068446096529453e+01
```

SOURCE lncdfn.src

SEE ALSO **cdfn2**

lncdfnc

PURPOSE Computes natural log of complement of Normal cumulative distribution function.

FORMAT *y* = **lncdfnc(***x***);**

INPUT *x* N×K matrix, abscissae.

OUTPUT *y* N×K matrix, $ln(1 - Pr(X < x))$.

SOURCE lncdfn.src

lnfact

PURPOSE Computes the natural log of the factorial function and can be used to compute log gamma.

FORMAT *y* = **lnfact(***x***);**

INPUT *x* N×K matrix or N-dimensional array, all elements must be positive.

OUTPUT *y* N×K matrix containing the natural log of the factorial of each of the elements in *x*.

REMARKS For integer *x*, this is (approximately) **ln(***x***!)**. However, the computation is done using a formula, and the function is defined for noninteger *x*.

In most formulae in which the factorial operator appears, it is possible to avoid computing the factorial directly, and to use **lnfact** instead. The advantage of this is that **lnfact** does not have the overflow problems that the factorial (**!**) operator has.

For $x \geq 1$, 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:

```
lngamma(x) = lnfact(x-1);
```

EXAMPLE

```
let x = 100 500 1000;
y = lnfact(x);
```

```
          363.739375560
y =       2611.33045846
          5912.12817849
```

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.

lnpdfmvn

PURPOSE Computes multivariate Normal log-probabilities.

lnpdfmvt

FORMAT `z = lnpdfmvn(x,s);`

INPUT
- *x* N×K matrix, data.
- *s* K×K matrix, covariance matrix.

OUTPUT
- *z* N×1 vector, log-probabilities.

REMARKS This computes the multivariate Normal log-probability for each row of *x*.

SOURCE `lnpdfn.src`

lnpdfmvt

PURPOSE Computes multivariate Student's t log-probabilities.

FORMAT `z = lnpdfmvt(x,s,nu);`

INPUT
- *x* N×K matrix, data.
- *s* K×K matrix, covariance matrix.
- *nu* scalar, degrees of freedom.

OUTPUT
- *z* N×1 vector, log-probabilities.

SOURCE `lnpdfn.src`

SEE ALSO **lnpdft**

lnpdfn

PURPOSE Computes standard Normal log-probabilities.

FORMAT *z* = **lnpdfn**(*x*);

INPUT *x* N×K matrix or N-dimensional array, data.

OUTPUT *z* N×K matrix or N-dimensional array, log-probabilities.

REMARKS This computes the log of the scalar Normal density function for each element of *x*. *z* could be computed by the following **GAUSS** code:

```
z = -ln(sqrt(2*pi))-x.*x/2;
```

For multivariate log-probabilities, see **lnpdfmvn**.

EXAMPLE
```
x = { -2, -1, 0, 1, 2 };
z = lnpdfn(x);
```

```
          −2.9189385
          −1.4189385
z =       −0.91893853
          −1.4189385
          −2.9189385
```

lnpdft

PURPOSE Computes Student's t log-probabilities.

FORMAT *z* = **lnpdft**(*x*,*nu*);

INPUT
x N×K matrix, data.
nu scalar, degrees of freedom.

OUTPUT
z N×K matrix, log-probabilities.

REMARKS This does not compute the log of the joint Student's t pdf. Instead, the scalar Normal density function is computed element-by-element.

For multivariate probabilities with covariance matrix see **lnpdfmvt**.

SEE ALSO **lnpdfmvt**

load, loadf, loadk, loadm, loadp, loads

PURPOSE Loads from a disk file.

FORMAT **load** [[**path**=*path*]] *x*, *y*[]=*filename*, *z*=*filename*;

REMARKS All the **load***xx* commands use the same syntax–they only differ in the types of symbols you use them for:

load, loadm	matrix
loads	string
loadf	function (**fn**)
loadk	keyword (**keyword**)
loadp	procedure (**proc**)

If no filename is given, as with *x* above, then the symbol name the file is to be loaded into is used as the filename, and the proper extension is added.

If more than one item is to be loaded in a single statement, the names should be separated by commas.

The filename can be either a literal or a string. If the filename is in a string variable, then the ^ (caret) operator must precede the name of the string, as in:

```
filestr = "mydata/char";
loadm x = ^filestr;
```

If no extension is supplied, the proper extension for each type of file will be used automatically as follows:

load	.fmt - matrix file or delimited ASCII file
loadm	.fmt - matrix file or delimited ASCII file
loads	.fst - string file
loadf	.fcg - user-defined function (**fn**) file
loadk	.fcg - user-defined keyword (**keyword**) file
loadp	.fcg - user-defined procedure (**proc**) file

These commands also signal to the compiler what type of object the symbol is so that later references to it will be compiled correctly.

A dummy definition must exist in the program for each symbol that is loaded in using **loadf**, **loadk**, or **loadp**. This resolves the need to have the symbol initialized at compile time. When the load executes, the dummy definition will be replaced with the saved definition:

```
proc corrmat; endp;
loadp corrmat;
y = corrmat;

keyword regress(x); endp;
loadk regress;
regress x on y z t from data01;
```

```
fn sqrd=;
loadf sqrd;
y = sqrd(4.5);
```

To load **GAUSS** files created with the **save** command, no brackets are used with the symbol name.

If you use **save** to save a scalar error code 65535 (i.e., **error(65535)**), it will be interpreted as an empty matrix when you **load** it again.

ASCII data files

To load ASCII data files, square brackets follow the name of the symbol.

Numbers in ASCII files must be delimited with spaces, commas, tabs, or newlines. If the size of the matrix to be loaded is not explicitly given, as in:

```
load x[] = data.asc;
```

GAUSS will load as many elements as possible from the file and create an N×1 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 N×1 matrix can be **reshape**'d to the desired form. You could, for instance, put the number of rows and columns of the matrix right in the file as the first and second elements and **reshape** the remainder of the vector to the desired form using those values.

If the size of the matrix is explicitly given in the **load** command, then no checking will be done. If you use:

```
load x[500,6] = data.asc;
```

GAUSS will still load as many elements as possible from the file into an N×1 matrix and then automatically reshape it using the dimensions given.

If you **load** data from a file, `data.asc`, which contains nine numbers (1 2 3 4 5 6 7 8 9), then the resulting matrix will be as follows:

```
load x[1,9] = data.asc;

x =  1  2  3  4  5  6  7  8  9

load x[3,3] = data.asc;

     1  2  3
x =  4  5  6
     7  8  9

load x[2,2] = data.asc;

x =  1  2
     3  4

load x[2,9] = data.asc;

x =  1  2  3  4  5  6  7  8  9
     1  2  3  4  5  6  7  8  9

load x[3,5] = data.asc;

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

load accepts pathnames. The following is legal:

```
loadm k = /gauss/x;
```

This will load /gauss/x.fmt into **k**.

If the **path=** subcommand is used with **load** and **save**, the path string will be remembered until changed in a subsequent command. This path will be used whenever none is specified. There are four separate paths for:

1. **load, loadm**
2. **loadf, loadp**
3. **loads**
4. **save**

Setting any of the four paths will not affect the others. The current path settings can be obtained (and changed) with the **sysstate** function, cases 4-7.

```
loadm path = /data;
```

This will change the **loadm** path without loading anything.

```
load path = /gauss x,y,z;
```

This will load x.fmt, y.fmt, and z.fmt using /gauss as a path. This path will be used for the next load if none is specified.

The **load** path or **save** path can be overridden in any particular **load** or **save** by putting an explicit path on the filename given to **load** from or **save** to as follows:

```
loadm path = /miscdata;
loadm x = /data/mydata1, y, z = hisdata;
```

In the above program:

`/data/mydata1.fmt` would be loaded into a matrix called **x**.

`/miscdata/y.fmt` would be loaded into a matrix called **y**.

`/miscdata/hisdata.fmt` would be loaded into a matrix called **z**.

```
oldmpath = sysstate(5,"/data");
load x, y;
call sysstate(5,oldmpath);
```

This will get the old **loadm** path, set it to `/data`, load `x.fmt` and `y.fmt`, and reset the **loadm** path to its original setting.

SEE ALSO **loadd**, **dataload**, **save**, **let**, **con**, **cons**, **sysstate**

loadarray

PURPOSE Loads an N-dimensional array from a disk file.

FORMAT **loadarray** [[**path**=*path*]] *x*, *y*=*filename*;

REMARKS If no filename is given, as with *x* above, then the symbol name the file is to be loaded into is used as the filename, and the proper extension is added.

If more than one item is to be loaded in a single statement, the names should be separated by commas.

The filename can be either a literal or a string. If the filename is in a string variable, then the ^ (caret) operator must precede the name of the string, as in:

```
filestr = "mydata/adat";
```

```
loadarray x = ^filestr;
```

If no extension is supplied, then an `.fmt` extension will be assumed.

loadarray accepts pathnames. The following is legal:

```
loadarray k = /gauss/a;
```

This will load `/gauss/a.fmt` into **k**.

If the **path=** subcommand is used, the path string will be remembered until changed in a subsequent command. This path will be used for all **loadarray**, **loadm**, and **load** calls whenever none is specified.

The current path setting can be obtained (and changed) with the **sysstate** function, case 5.

```
loadarray path = /data;
```

This will change the **loadarray** path without loading anything.

```
loadarray path = /gauss a,b,c;
```

This will load `a.fmt`, `b.fmt`, and `c.fmt` using `/gauss` as a path. This path will be used for the next **loadarray**, **loadm**, or **load** call if none is specified.

The **load** path or **save** path can be overridden in any particular **load** or **save** by putting an explicit path on the filename given to **load** from or **save** to as follows:

```
loadarray path = /miscdata;
loadarray a = /data/mydata1, b, c = hisdata;
```

In the above program:

`/data/mydata1.fmt` would be loaded into an array called **a**.

`/miscdata/b.fmt` would be loaded into an array called **b**.

`/miscdata/hisdata.fmt` would be loaded into an array called **c**.

```
oldarraypath = sysstate(5,"/data");
loadarray a, b;
call sysstate(5,oldarraypath);
```

This will get the old **loadarray** path, set it to `/data`, load `a.fmt` and `b.fmt`, and reset the **loadarray** path to its original setting.

SEE ALSO **load**, **loadm**, **save**, **let**, **sysstate**

loadd

PURPOSE Loads a data set.

FORMAT *y* **= loadd(***dataset***);**

INPUT *dataset* string, name of data set.

OUTPUT *y* N×K matrix of data.

REMARKS The data set must not be larger than a single **GAUSS** matrix.

If *dataset* is a null string or 0, the data set `temp.dat` will be loaded. To load a matrix file, use an `.fmt` extension on *dataset*.

SOURCE `saveload.src`

GLOBALS **__maxvec**

loadstruct

PURPOSE Loads a structure into memory from a file on the disk.

FORMAT { *instance*,*retcode* } = **loadstruct(***file_name*,*structure_type*);

INPUT *file_name* string, name of file containing structure.
structure_type string, structure type.

OUTPUT *instance* instance of the structure.
retcode scalar, 0 if successful, otherwise 1.

REMARKS *instance* can be an array of structures.

EXAMPLE
```
#include ds.sdf

struct DS p3;

{ p3, retc } = loadstruct("p2", "ds");
```

loadwind

PURPOSE Load a previously saved graphic panel configuration.

LIBRARY pgraph

FORMAT *err* = **loadwind(***namestr***);**

INPUT *namestr* string, name of file to be loaded.

OUTPUT *err* scalar, 0 if successful, 1 if graphic panel matrix is invalid. Note that the current graphic panel configuration will be overwritten in either case.

SOURCE `pwindow.src`

GLOBALS **_pwindmx**

SEE ALSO **savewind**

local

PURPOSE Declare variables that are to exist only inside a procedure.

FORMAT **local** *x*, *y*, *f*:**proc**;

REMARKS The statement above would place the names *x*, *y*, and *f* in the local symbol table for the current procedure being compiled. This statement is legal only between the **proc** statement and the **endp** statement of a procedure definition.

These symbols cannot be accessed outside of the procedure.

The symbol *f* in the statement above will be treated as a procedure whenever it is accessed in the current procedure. What is actually passed in is a pointer to a procedure.

See PROCEDURES AND KEYWORDS, Chapter 12.

SEE ALSO **proc**

locate

PURPOSE Positions the cursor in the window.

FORMAT **locate** *m*, *n*;

PORTABILITY **Windows** only

REMARKS **locate** locates the cursor in the current output window.

m and *n* denote the row and column, respectively, at which the cursor is to be located.

The origin (1,1) is the upper left corner.

m and *n* may be any expressions that return scalars. Nonintegers will be truncated to an integer.

EXAMPLE
```
r = csrlin;
c = csrcol;
cls;
locate r,c;
```

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

SEE ALSO **csrlin**, **csrcol**

loess

PURPOSE Computes coefficients of locally weighted regression.

FORMAT { *yhat*,*ys*,*xs* } = **loess**(*depvar*,*indvars*);

INPUT *depvar* N×1 vector, dependent variable.

indvars N×K 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* N×1 vector, predicted *depvar* given *indvars*.

ys **_loess_numEval**×1 vector, ordinate values given abscissae values in *xs*.

xs **_loess_numEval**×1 vector, equally spaced abscissae values.

REMARKS Based on Cleveland, William S. "Robust Locally Weighted Regression and Smoothing Scatterplots." JASA, Vol. 74, 1979, 829-836.

SOURCE `loess.src`

loessmt

PURPOSE Computes coefficients of locally weighted regression.

INCLUDE `loessmt.sdf`

FORMAT { *yhat*,*ys*,*xs* } = **loessmt**(*lc0*,*depvar*,*indvars*);

INPUT *lc0* an instance of a **loessmtControl** structure, containing the following members:

	lc0.Span	scalar, degree of smoothing. Must be greater than 2/N. Default = .67777.
	lc0.NumEval	scalar, number of points in *ys* and *xs*. Default = 50.
	lc0.Degree	scalar, if 2, quadratic fit, otherwise linear. Default = 1.
	lc0.WgtType	scalar, type of weights. If 1, robust, symmetric weights, otherwise Gaussian. Default = 1.
	lc0.output	scalar, if 1, iteration information and results are printed, otherwise nothing is printed.
depvar	N×1 vector, dependent variable.	
indvars	N×K matrix, independent variables.	

OUTPUT

yhat	N×1 vector, predicted *depvar* given *indvars*.
ys	*lc0*.**numEval**×1 vector, ordinate values given abscissae values in *xs*.
xs	*lc0*.**numEval**×1 vector, equally spaced abscissae values.

REMARKS Based on Cleveland, William S. "Robust Locally Weighted Regression and Smoothing Scatterplots." JASA, Vol. 74, 1979, 829-836.

SOURCE loessmt.src

SEE ALSO **loessmtControlCreate**

loessmtControlCreate

PURPOSE Creates default **loessmtControl** structure.

INCLUDE loessmt.sdf

FORMAT *c* = **loessmtControlCreate;**

OUTPUT *c* instance of a **loessmtControl** structure with members set to default values.

SOURCE loessmt.src

SEE ALSO **loessmt**

log

PURPOSE Computes the $\log_{10}$ of all elements of *x*.

FORMAT *y* = **log**(*x*);

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array containing the log 10 values of the elements of *x*.

REMARKS **log** is defined for $x \neq 0$.

You can turn the generation of complex numbers for negative inputs on or off in the **GAUSS** configuration file, and with the **sysstate** function, case 8. If you turn it off, **log** will generate an error for negative inputs.

If *x* is already complex, the complex number state doesn't matter; **log** will compute a complex result.

x can be any expression that returns a matrix.

EXAMPLE
```
x = round(rndu(3,3)*10+1);
y = log(x);
```

```
             4.0000000000   2.0000000000   1.0000000000
x =  10.0000000000   4.0000000000   8.0000000000
             7.0000000000   2.0000000000   6.0000000000


             0.6020599913   0.3010299957   0.3010299957
y =  1.0000000000   0.6020599913   0.9030899870
             0.8450980400   0.3010299957   0.7781512504
```

SEE ALSO **ln**

loglog

PURPOSE Graphs X vs. Y using log coordinates.

LIBRARY pgraph

FORMAT **loglog**(*x*,*y*);

INPUT

x	N×1 or N×M matrix. Each column contains the X values for a particular line.
y	N×1 or N×M matrix. Each column contains the Y values for a particular line.

SOURCE ploglog.src

SEE ALSO **xy**, **logx**, **logy**

logx

PURPOSE Graphs X vs. Y using log coordinates for the X axis.

LIBRARY pgraph

FORMAT **logx**(*x*,*y*);

INPUT
x N×1 or N×M matrix. Each column contains the X values for a particular line.

y N×1 or N×M matrix. Each column contains the Y values for a particular line.

SOURCE plogx.src

SEE ALSO **xy**, **logy**, **loglog**

logy

PURPOSE Graphs X vs. Y using log coordinates for the Y axis.

LIBRARY pgraph

FORMAT **logy**(*x*,*y*);

INPUT
x N×1 or N×M matrix. Each column represents the X values for a particular line.

y N×1 or N×M matrix. Each column represents the Y values for a particular line.

SOURCE plogy.src

SEE ALSO **xy**, **logx**, **loglog**

loopnextindex

PURPOSE Increments an index vector to the next logical index and jumps to the specified label if the index did not wrap to the beginning.

FORMAT **loopnextindex** *lab*,*i*,*o* [[,*dim*]];

INPUT

lab	literal, label to jump to if **loopnextindex** succeeds.
i	M×1 vector of indices into an array, where M<=N.
o	N×1 vector of orders of an N-dimensional array.
dim	scalar [1-M], index into the vector of indices *i*, corresponding to the dimension to walk through, positive to walk the index forward, or negative to walk backward.

REMARKS If the argument *dim* is given, **loopnextindex** will walk through only the dimension indicated by *dim* in the specified direction. Otherwise, if *dim* is not given, each call to **loopnextindex** will increment *i* to index the next element or subarray of the corresponding array.

loopnextindex will jump to the label indicated by *lab* if the index can walk further in the specified dimension and direction, otherwise it will fall out of the loop and continue through the program.

When the index matches the vector of orders, the index will be reset to the beginning and program execution will resume at the statement following the **loopnextindex** statement.

EXAMPLE

```
orders = { 2,3,4,5,6,7 };
a = arrayalloc(orders,0);
ind = { 1,1,1,1 };

loopni:

    setarray a, ind, rndn(6,7);
```

```
    loopnextindex loopni, ind, orders;
```

This example sets each 6×7 subarray of array **a**, by incrementing the index at each call of **loopnextindex** and then going to the label **loopni**. When **ind** cannot be incremented, the program drops out of the loop and continues.

```
ind = { 1,1,4,5 };

loopni2:

    setarray a, ind, rndn(6,7);
    loopnextindex loopni2, ind, orders, 2;
```

Using the array and vector of orders from the example above, this example increments the second value of the index vector **ind** during each call to **loopnextindex**. This loop will set the 6×7 subarrays of **a** that begin at [1,1,4,5,1,1], [1,2,4,5,1,1], and [1,3,4,5,1,1], and then drop out of the loop.

SEE ALSO **nextindex**, **previousindex**, **walkindex**

lower

PURPOSE Converts a string or character matrix to lowercase.

FORMAT *y* = **lower**(*x*);

INPUT *x* string or N×K matrix of character data to be converted to lowercase.

OUTPUT *y* string or N×K matrix which contains the lowercase equivalent of the data in *x*.

REMARKS If *x* is a numeric matrix, *y* will contain garbage. No error message will be

generated since **GAUSS** does not distinguish between numeric and character data in matrices.

EXAMPLE

```
x = "MATH 401";
y = lower(x);
print y;
```

produces:

```
math 401
```

SEE ALSO **upper**

lowmat, lowmat1

PURPOSE Returns the lower portion of a matrix. **lowmat** returns the main diagonal and every element below. **lowmat1** is the same except it replaces the main diagonal with ones.

FORMAT *L* = **lowmat**(*x*);

L = **lowmat1**(*x*);

INPUT *x* N×N matrix.

OUTPUT *L* N×N 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 = { 1  2 -1,
      2  3 -2,
      1 -2  1 };
```

```
L = lowmat(x);
L1 = lowmat1(x);
```

The resulting matrices are

$$\mathtt{L} = \begin{matrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 1 & -2 & 1 \end{matrix}$$

$$\mathtt{L1} = \begin{matrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{matrix}$$

SOURCE diag.src

SEE ALSO **upmat**, **upmat1**, **diag**, **diagrv**, **crout**, **croutp**

ltrisol

PURPOSE Computes the solution of $Lx = b$ where L is a lower triangular matrix.

FORMAT x **= ltrisol(**b**,**L**);**

INPUT

b	P×K matrix.
L	P×P lower triangular matrix.

OUTPUT

x	P×K matrix, soluion of $Lx = b$.

ltrisol applies a forward solve to $Lx = b$ to solve for x. If b has more than one column, each column will be solved for separately, i.e., **ltrisol** will apply a forward solve to L*****x**[.,**i**] =** b**[.,**i**]**.

lu

PURPOSE Computes the LU decomposition of a square matrix with partial (row) pivoting, such that: $X = LU$.

FORMAT { *l*,*u* } = **lu**(*x*);

INPUT *x* N×N square nonsingular matrix.

OUTPUT *l* N×N "scrambled" lower triangular matrix. This is a lower triangular matrix that has been reordered based on the row pivoting.

u N×N 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;
```

$$\mathtt{x} = \begin{matrix} 0.1523 + 0.7685i & -0.8957 + 0.0342i & 2.4353 + 2.7736i \\ -1.1953 + 1.2187i & 1.2118 + 0.2571i & -0.0446 - 1.7768i \\ 0.8038 + 1.3668i & 1.2950 - 1.6929i & 1.6267 + 0.2844i \end{matrix}$$

$$\mathtt{l} = \begin{matrix} 0.2589 - 0.3789i & -1.2417 - 0.5225i & 1.0000 \\ 1.0000 & 0.0000 & 0.0000 \\ 0.2419 - 0.8968i & 1.0000 & 0.0000 \end{matrix}$$

$$\mathtt{u} = \begin{matrix} -1.1953 + 1.2187i & 1.2118 + 0.2571i & -0.0446 - 1.7768i \\ 0.0000 & 0.7713 - 0.6683i & 3.2309 + 0.6742i \\ 0.0000 & 0.0000 & 6.7795 + 5.7420i \end{matrix}$$

$$\texttt{x2} = \begin{matrix} 0.1523 + 0.7685i & -0.8957 + 0.0342i & 2.4353 + 2.7736i \\ -1.1953 + 1.2187i & 1.2118 + 0.2571i & -0.0446 - 1.7768i \\ 0.8038 + 1.3668i & 1.2950 - 1.6929i & 1.6267 + 0.2844i \end{matrix}$$

SEE ALSO **crout**, **croutp**, **chol**

lusol

PURPOSE Computes the solution of $LUx = b$ where L is a lower triangular matrix and U is an upper triangular matrix.

FORMAT *x* **= lusol(***b***,***L***,***U***);**

INPUT

b	P×K matrix.
L	P×P lower triangular matrix.
U	P×P upper triangular matrix.

OUTPUT

x	P×K matrix, solution of $LUx = b$.

REMARKS If *b* has more than one column, each column is solved for separately, i.e., **lusol** solves *LUx***[.,***i***] = ***b***[.,***i***]**.

machEpsilon

PURPOSE Returns the smallest number such that 1+ *eps* > 1.

FORMAT *eps* **= machEpsilon;**

OUTPUT

eps	scalar, machine epsilon.

SOURCE machchonst.src

make (dataloop)

PURPOSE Specifies the creation of a new variable within a data loop.

FORMAT **make** [[#]] *numvar* = *numeric_expression*;

make $ *charvar* = *character_expression*;

REMARKS A *numeric_expression* is any valid expression returning a numeric vector. A *character_expression* is any valid expression returning a character vector. If neither '**$**' nor '**#**' is specified, '**#**' is assumed.

The expression may contain explicit variable names and/or **GAUSS** commands. Any variables referenced must already exist, either as elements of the source data set, as **extern**'s, or as the result of a previous **make**, **vector**, or **code** statement. The variable name must be unique. A variable cannot be made more than once, or an error is generated.

EXAMPLE
```
make sqvpt =  sqrt(velocity * pressure * temp);
make $ sex = lower(sex);
```

SEE ALSO **vector (dataloop)**

makevars

PURPOSE Creates separate global vectors from the columns of a matrix.

FORMAT **makevars(***x*,*vnames*,*xnames***);**

INPUT

x	N×K matrix whose columns will be converted into individual vectors.
vnames	string or M×1 character vector containing names of global vectors to create. If 0, all names in *xnames* will be used.
xnames	string or K×1 character vector containing names to be associated with the columns of the matrix *x*.

REMARKS

If *xnames* = 0, the prefix X will be used to create names. Therefore, if there are 9 columns in *x*, the names will be X1-X9, if there are 10, they will be X01-X10, and so on.

If *xnames* or *vnames* is a string, the individual names must be separated by spaces or commas:

```
vnames = "age pay sex";
```

Since these new vectors are created at execution time, the compiler will not know they exist until after **makevars** has executed once. This means that you cannot access them by name unless you previously **clear** them or otherwise add them to the symbol table. (See **setvars** for a quick interactive solution to this.)

This function is the opposite of **mergevar**.

EXAMPLE

```
let x[3,3] = 101 35 50000
             102 29 13000
             103 37 18000;
let xnames = id age pay;
let vnames = age pay;
makevars(x,vnames,xnames);
```

Two global vectors, called **age** and **pay**, are created from the columns of **x**.

```
let x[3,3] = 101 35 50000
             102 29 13000
```

```
                103 37 18000;
xnames = "id age pay";
vnames = "age pay";
makevars(x,vnames,xnames);
```

This is the same as the example above, except that strings are used for the variable names.

SOURCE vars.src

GLOBALS __vpad

SEE ALSO mergevar, setvars

makewind

PURPOSE Creates a graphic panel of specific size and position and adds it to the list of graphic panels.

LIBRARY pgraph

FORMAT **makewind(***xsize*,*ysize*,*xshft*,*yshft*,*typ***);**

INPUT

xsize	scalar, horizontal size of the graphic panel in inches.
ysize	scalar, vertical size of the graphic panel in inches.
xshft	scalar, horizontal distance from left edge of window in inches.
yshft	scalar, vertical distance from bottom edge of window in inches.
typ	scalar, graphic panel attribute type. If this value is 1, the graphic panels will be transparent. If 0, the graphic panels will be nontransparent.

REMARKS Note that if this procedure is used when rotating the page, the passed parameters are scaled appropriately to the newly oriented page. The size and shift values will not be true inches when printed, but the graphic panel size to page size ratio will remain the same. The result of this implementation automates the rotation and eliminates the required graphic panel recalculations by the user.

See the **window** command for creating tiled graphic panels. For more information on using graphic panels, see GRAPHIC PANELS, Section 24.3.

SOURCE pwindow.src

SEE ALSO **window**, **endwind**, **setwind**, **getwind**, **begwind**, **nextwind**

margin

PURPOSE Sets the margins for the current graph's graphic panel.

LIBRARY pgraph

FORMAT **margin**(*l*,*r*,*t*,*b*);

INPUT

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

REMARKS By default, the dimensions of the graph are the same as the graphic panel dimensions. With this function the graph dimensions may be decreased. The result will be a smaller plot area surrounded by the specified margin. This procedure takes into consideration the axes labels and numbers for correct placement.

All input inch values for this procedure are based on a full size window of 9×6.855 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**

matalloc

PURPOSE Allocates a matrix with unspecified contents.

FORMAT *y* = **matalloc**(*r*,*c*);

INPUT
r scalar, rows.

c scalar, columns.

OUTPUT
y *r*×*c* matrix.

REMARKS The contents are unspecified. This function is used to allocate a matrix that will be written to in sections using indexing or used with the Foreign Language Interface as an output matrix for a function called with **dllcall**.

SEE ALSO **matinit**, **ones**, **zeros**, **eye**

matinit

PURPOSE Allocates a matrix with a specified fill value.

FORMAT *y* = **matinit**(*r*,*c*,*v*);

INPUT
r scalar, rows.
c scalar, columns.
v scalar, value to initialize.

OUTPUT *y* *r*×*c* matrix with each element equal to the value of *v*.

SEE ALSO **matalloc**, **ones**, **zeros**, **eye**

mattoarray

PURPOSE Converts a matrix to a type array.

FORMAT *y* = **mattoarray**(*x*);

INPUT *x* matrix.

OUTPUT *y* 1-or-2-dimensional array.

REMARKS If the argument *x* is a scalar, **mattoarray** will simply return the scalar, without changing it to a type array.

EXAMPLE
```
x = 5*ones(2,3);
y = mattoarray(x);
```

y will be a 2×3 array of fives.

SEE ALSO **arraytomat**

maxc

PURPOSE Returns a column vector containing the largest element in each column of a matrix.

FORMAT *y* = **maxc**(*x*);

INPUT *x* N×K matrix or sparse matrix.

OUTPUT *y* K×1 matrix containing the largest element in each column of *x*.

REMARKS If *x* is complex, **maxc** uses the complex modulus (**abs**(*x*)) to determine the largest elements.

To find the maximum elements in each row of a matrix, transpose the matrix before applying the **maxc** function.

To find the maximum value in the whole matrix if the matrix has more than one column, nest two calls to **maxc**:

```
y = maxc(maxc(x));
```

EXAMPLE

```
x = rndn(4,2);
y = 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**

maxindc

PURPOSE Returns a column vector containing the index (i.e., row number) of the maximum element in each column of a matrix.

FORMAT *y* **= maxindc(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* K×1 matrix containing the index of the maximum element in each column of *x*.

REMARKS If *x* is complex, **maxindc** uses the complex modulus (**abs(***x***)**) to determine the largest elements.

To find the index of the maximum element in each row of a matrix, transpose the matrix before applying **maxindc**.

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);
```

$$\mathtt{x} = \begin{matrix} 1 & -11 & 0 & 5 \\ 0 & 0 & -2 & -6 \\ -8 & 0 & 3 & 2 \\ -11 & 5 & -4 & 5 \end{matrix}$$

$$\mathtt{y} = \begin{matrix} 1 \\ 5 \\ 3 \\ 5 \end{matrix}$$

$$\mathtt{z} = \begin{matrix} 1 \\ 4 \\ 3 \\ 1 \end{matrix}$$

SEE ALSO **maxc**, **minindc**, **minc**

maxv

PURPOSE Performs an element by element comparison of two matrices and returns the maximum value for each element.

FORMAT *z* **= maxv(***x***,***y***);**

GLOBAL INPUT

x	N×K matrix	
y	N×K matrix	

OUTPUT

z	A N×K matrix whose values are the maximum of each element from the arguments *x* and *y*.

REMARKS **maxv** works for sparse matrices as well as arrays.

```
EXAMPLE x = rndn(10,10);
        y = rndn(10,10);

        z = maxv(x,y);
        \end{vebatim}

        \See        \commandname{minv}

        \endcr\CR{maxvec}
        \index{maxvec#@\commandname{maxvec}}

        \Purpose

        Returns maximum vector length allowed.

        \Format  \textcode{\substitute{y} = maxvec;}

        \GInput
        \begin{commandlist}
        \item[\_\_maxvec] scalar, maximum vector length allowed.
        \end{commandlist}
        \Output

        \begin{argumentlist}
        \item[y] scalar, maximum vector length.
        \end{argumentlist}

        \Remarks


        \commandname{maxvec} returns the value in the global scalar
        \commandname{\_\_maxvec}, which can be reset in the calling program.

        \commandname{maxvec} is called by \rtl{} functions and applications
        when determining how many rows can be read from a data set in one
        call to \commandname{readr}.
```

```
Using a value that is too large can cause excessive disk thrashing.
The trick is to allow
the algorithm making the disk reads to execute entirely in RAM.

\Example
\begin{verbatim}
     y = maxvec;
     print y;

          20000.000
```

SOURCE system.src

maxbytes

PURPOSE Returns maximum memory to be used.

FORMAT *y* = **maxbytes;**

GLOBAL INPUT **__maxbytes** scalar, maximum memory to be used.

OUTPUT *y* scalar, maximum memory to be used.

REMARKS **maxbytes** returns the value in the global scalar **__maxbytes**, which can be reset in the calling program.

maxbytes is called by **Run-Time Library** functions and applications when determining how many rows can be read from a data set in one call to **readr**.

maxbytes replaced the obsolete command **coreleft**. If **coreleft** returns a meaningful number for your operating system and if you wish to reference it, set **__maxbytes** = 0 and then call **maxbytes**.

EXAMPLE

```
y = maxbytes;
```

```
print y;

     100000000.000
```

SOURCE system.src

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	K×1 vector, abscissae.
n	scalar, highest order.
alpha	scalar, $0 \leq alpha < 1$.

OUTPUT

y	K×N matrix, evaluations of the modified Bessel or the exponentially scaled modified Bessel of the first kind of the n^{th} order.

REMARKS For the functions that permit you to specify the order, the returned matrix contains a sequence of modified or exponentially scaled modified Bessel values of different orders. For the i^{th} row of *y*:

$$y[i,.] = \begin{array}{cccc} I_\alpha(x[i]) & I_{\alpha+1}(x[i]) & \cdots & I_{\alpha+n-1}(x[i]) \end{array}$$

The remaining functions generate modified Bessels of only the specified order.

The exponentially scaled modified Bessels are related to the unscaled modifed Bessels in the following way:

```
mbesselei0(x) = exp(-x) * mbesseli0(x)
```

The use of the scaled versions of the modified Bessel can improve the numerical properties of some calculations by keeping the intermediate numbers small in size.

EXAMPLE

This example produces estimates for the "circular" response regression model (Fisher, N.I. *Statistical Analysis of Circular Data*. NY: Cambridge University Press, 1993.), where the dependent variable varies between $-\pi$ 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:

$$logL = -N \times ln(I_0(\kappa)) + \kappa \sum_{i}^{N} cos(y_i - \mu - G(X_i B))$$

To generate estimates it is necessary to maximize this function using an iterative method. **QNewton** is used here.

κ is required to be nonnegative and therefore in the example below, the exponential of this parameter is estimated instead. Also, the exponentially

scaled modified Bessel is used to improve numerical properties of the calculations.

The **arctan** function is used in $G()$ to map XB to the $[-\pi, \pi]$ 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(mbesselei0(exp(b[1])) -
    sumc(exp(b[1])*(cos(dev)-1))));
endp;

loadm data;
y0 = data[.,1];
x0 = data[.,2:cols(data)];

b0 = 2*ones(cols(x0),1);

{ b,fct,grd,ret } = QNewton(&lpr,b0);

cov = invpd(hessp(&lpr,b));

print "estimates   standard errors";
print;
print b~sqrt(diag(cov));
```

SOURCE ribesl.src

meanc

PURPOSE Computes the mean of every column of a matrix.

FORMAT *y* **= meanc(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* K×1 matrix containing the mean of every column of *x*.

EXAMPLE
```
x = meanc(rndu(2000,4));
```

```
x =  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* **= median(***x***);**

INPUT *x* N×K matrix.

OUTPUT *m* K×1 vector containing the medians of the respective columns of *x*.

EXAMPLE
```
x = { 8 4,
      6 8,
      3 7 };

y = median(x);
```

```
y =  6.0000000
     7.0000000
```

SOURCE median.src

mergeby

PURPOSE Merges two sorted files by a common variable.

FORMAT **mergeby(***infile1***,***infile2***,***outfile***,***keytyp***);**

INPUT
infile1 string, name of input file 1.

infile2 string, name of input file 2.

outfile string, name of output file.

keytyp scalar, data type of key variable.

1 numeric

2 character

REMARKS This will combine the variables in the two files to create a single large file. The following assumptions hold:

1. Both files have a single (key) variable in common and it is the first variable.
2. All of the values of the key variable are unique.
3. Each file is already sorted on the key variable.

The output file will contain the key variable in its first column.

It is not necessary for the two files to have the same number of rows. For each row for which the key variables match, a row will be created in the output file. *outfile* will contain the columns from *infile1* followed by the columns from *infile2* minus the key column from the second file.

If the inputs are null ("" or 0), the procedure will ask for them.

SOURCE `sortd.src`

mergevar

PURPOSE Accepts a list of names of global matrices, and concatenates the corresponding matrices horizontally to form a single matrix.

FORMAT *x* = **mergevar(***vnames***);**

INPUT *vnames* string or K×1 column vector containing the names of K global matrices.

OUTPUT *x* N×M matrix that contains the concatenated matrices, where M is the sum of the columns in the K matrices specified in *vnames*.

REMARKS The matrices specified in *vnames* must be globals and they must all have the same number of rows.

This function is the opposite of **makevars**.

EXAMPLE
```
let vnames = age pay sex;
x = mergevar(vnames);
```

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 of a matrix.

FORMAT *y* = **minc**(*x*);

INPUT *x* N×K matrix or sparse matrix.

OUTPUT *y* K×1 matrix containing the smallest element in each column of *x*.

REMARKS If *x* is complex, **minc** uses the complex modulus (**abs**(*x*)) to determine the smallest elements.

To find the minimum element in each row, transpose the matrix before applying the **minc** function.

To find the minimum value in the whole matrix, nest two calls to **minc**:

```
y = minc(minc(x));
```

EXAMPLE
```
x = rndn(4,2);
y = minc(x);
```

```
x =  −1.061321   −0.729026
     −0.021965    0.184246
      1.843242   −1.847015
      1.977621   −0.532307

y =  −1.061321
     −1.847015
```

SEE ALSO **maxc**, **minindc**, **maxindc**

minindc

PURPOSE Returns a column vector containing the index (i.e., row number) of the smallest element in each column of a matrix.

FORMAT *y* **= minindc(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* K×1 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);
```

$$\mathtt{x} = \begin{matrix} -5 & 6 & -4 & -1 \\ 2 & -2 & 1 & 3 \\ 6 & 0 & 1 & -7 \\ -6 & 0 & 8 & -4 \\ 7 & -4 & 8 & 3 \end{matrix}$$

$$\mathtt{y} = \begin{matrix} -6 \\ -4 \\ -4 \\ -7 \end{matrix}$$

$$\mathtt{z} = \begin{matrix} 4 \\ 5 \\ 1 \\ 3 \end{matrix}$$

SEE ALSO **maxindc**, **minc**, **maxc**

miss, missrv

PURPOSE **miss** converts specified elements in a matrix to **GAUSS**'s missing value code. **missrv** is the reverse of this, and converts missing values into specified values.

FORMAT *y* **= miss(***x*,*v***);**

y **= missrv(***x*,*v***);**

INPUT *x* N×K matrix.

v L×M matrix, E×E 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);
```

In this example, **x** must have 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**

missex

PURPOSE Converts numeric values to the missing value code according to the values given in a logical expression.

FORMAT *y* = **missex**(*x*,*e*);

INPUT
- *x* N×K matrix.
- *e* N×K 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* N×K matrix that equals *x*, but with those elements that correspond to the 1's in *e* converted to missing.

REMARKS The matrix *e* will usually be created by a logical expression. For instance, to convert all numbers between 10 and 15 in *x* to missing, the following code could be used:

```
y = missex(x, (x .>10) .and (x .<15));
```

Note that "dot" operators MUST be used in constructing the logical expressions.

For complex matrices, the missing value code is defined as a missing value entry in the real part of the matrix. For complex *x*, then, **missex** replaces elements with a ". + 0i" value.

This function is like **miss**, but is more general in that a range of values can be converted into missings.

EXAMPLE

```
x = rndu(3,2);
/* logical expression */
e = (x .> .10) .and (x .< .20);
y = missex(x,e);
```

A 3×2 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**

minv

PURPOSE Performs an element by element comparison of two matrices and returns the minimum value for each element.

FORMAT *z* = **minv**(*x*,*y*);

GLOBAL INPUT

x	N×K matrix
y	N×K matrix

OUTPUT

z	A N×K matrix whose values are the minimum of each element from the arguments *x* and *y*.

REMARKS **maxv** works for sparse matrices as well as arrays.

EXAMPLE

```
x = rndn(10,10);
y = rndn(10,10);
```

```
z = minv(x,y);
\end{vebatim}

\See       \commandname{maxv}

\endcr\CR{moment}
\index{moment#@\commandname{moment}}

\Purpose

Computes a cross-product\index{cross-product}
matrix.  This is the same as \substitute{x}$ˆ\prime$\substitute{x}.

\Format    \textcode{\substitute{y} = moment(\substitute{x},\substitute{d})

\Input
          \begin{argumentlist}
          \item[x]  N$\times$K matrix or M-dimensional array where
the last two dimensions are N$\times$K.
          \item[d]  scalar, controls handling of missing values.

          \begin{List}
           \item[0] missing values will not be checked for.
           This is the fastest option.

           \item[1] ‘‘listwise deletion\index{deletion}\index{listwise dele
           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 \substitute{x} contains missing
           values, then \textcode{moment(\substitute{x},1)} will return a s

           \item[2] ‘‘pairwise deletion\index{pairwise deletion}’’ is used.
           Any element of \substitute{x} that is missing is excluded from
           the computation of the moment matrix\index{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.
          \end{List}
          \end{argumentlist}
\Output
          \begin{argumentlist}
          \item[y]  K$\times$K matrix or M-dimensional array where
the last two dimensions are K$\times$K, the cross-product
of \substitute{x}.
          \end{argumentlist}

\Remarks

The fact that the moment
matrix is symmetric is taken into account to cut
execution time almost in half.

If \substitute{x} is an array, the result will be an array containing
the cross-products of each 2-dimensional array described by the two
trailing dimensions of \substitute{x}. In other words, for a
10$\times$4$\times$4 array \substitute{x}, the resulting array
\substitute{y} will contain the cross-products of each fo the 10
4$\times$4 arrays contained in \substitute{x}, so
\textcode{\substitute{y}[\substitute{n},.,.]=\substitute{x}[\substitute{n},
for 1$\le$n$\le$10.

If there is no missing data then \substitute{d} = 0 should be used
because it will be faster.

The \commandname{/} operator (matrix division) will automatically
form a moment matrix (performing pairwise
deletions if \textcode{trap 2} is set) and will compute the \commandname{ol
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 \commandname{/} operator is
```

used.

\Example
\begin{verbatim}
```
xx = moment(x,2);
ixx = invpd(xx);
b = ixx*missrv(x,0)'y;
```

In this example, the regression of **y** on **x** is computed. The moment matrix (**xx**) is formed using the **moment** command (with pairwise deletion, since the second parameter is 2). Then **xx** is inverted using the **invpd** function. Finally, the **ols** coefficients are computed. **missrv** is used to emulate pairwise deletion by setting missing values to 0.

momentd

PURPOSE Computes a moment ($X'X$) matrix from a **GAUSS** data set.

FORMAT *m* **= momentd(***dataset***,***vars***);**

INPUT

dataset string, name of data set.

vars K×1 character vector, names of variables

- or -

K×1 numeric vector, indices of columns.

These can be any size subset of the variables in the data set, and can be in any order. If a scalar 0 is passed, all columns of the data set will be used.

GLOBAL INPUT

__con scalar, default 1.

1 a constant term will be added.

0 no constant term will be added.

__miss scalar, default 0.

0 there are no missing values (fastest).

1 do listwise deletion; drop an observation if any missings occur in it.

2 do pairwise deletion; this is equivalent to setting missings to 0 when calculating *m*.

__row scalar, the number of rows to read per iteration of the read loop, default 0.

If 0, the number of rows will be calculated internally.

If you get an **Insufficient memory** error, or you want the rounding to be exactly the same between runs, you can set the number of rows to read before calling **momentd**.

OUTPUT *m* M×M matrix, where M = K + **__con**, the moment matrix constructed by calculating X′X where X is the data, with or without a constant vector of ones.

Error handling is controlled by the low order bit of the trap flag.

trap 0 terminate with error message

trap 1 return scalar error code in *m*

33 too many missings

34 file not found

EXAMPLE
```
z = { age, pay, sex };
m = momentd("freq",z);
```

SOURCE momentd.src

movingave

PURPOSE Computes moving average of a series.

FORMAT *y* = **movingave**(*x*,*d*);

INPUT
- *x* N×K matrix.
- *d* scalar, order of moving average.

OUTPUT
- *y* N×K matrix, filtered series. The first *d*-1 rows of *x* are set to missing values.

REMARKS **movingave** is essentially a smoothing time series filter. The moving average as performed by column and thus it treats the N×K matrix as K time series of length N.

SEE ALSO **movingaveWgt**, **movingaveExpwgt**

movingaveExpwgt

PURPOSE Computes exponentially weighted moving average of a series.

FORMAT *y* = **movingaveExpwgt**(*x*,*d*,*p*);

INPUT
- *x* N×K matrix.
- *d* scalar, order of moving average.
- *p* scalar, smoothing coefficient where 0>*p*>1.

OUTPUT
- *y* N×K matrix, filtered series. The first *d*-1 rows of *x* are set to missing values.

REMARKS **movingaveExpwgt** is smoothing time series filter using exponential weights. The moving average as performed by column and thus it treats the N×K matrix as K time series of length N.

SEE ALSO **movingaveWgt**, **movingave**

movingaveWgt

PURPOSE Computes weighted moving average of a series

FORMAT *y* = **movingaveWgt**(*x*,*d*,*w*);

INPUT

x	N×K matrix.
d	scalar, order of moving average.
w	*d*×1 vector, weights.

OUTPUT

y	N×K matrix, filtered series. The first *d*-1 rows of *x* are set to missing values.

REMARKS **movingaveWgt** is essentially a smoothing time series filter with weights. The moving average as performed by column and thus it treats the N×K matrix as K time series of length N.

SEE ALSO **movingave**, **movingaveExpwgt**

msym

PURPOSE Allows the user to set the symbol that **GAUSS** uses when missing values are converted to ASCII and vice versa.

FORMAT **msym** *str*;

INPUT

str	literal or ^string (up to 8 letters) which, if not surrounded by quotes, is forced to uppercase. This is the string to be printed for missing values. The default is '**.**'.

REMARKS The entire string will be printed out when converting to ASCII in **print** and **printfm** statements.

When converting ASCII to binary in **loadm** and **let** statements, only the first character is significant. In other words,

```
msym HAT;
```

will cause '**H**' to be converted to missing on input.

This does not affect **writer**, which outputs data in binary format.

SEE ALSO **print**, **printfm**

new

PURPOSE Erases everything in memory including the symbol table; closes all open files as well as the auxiliary output and turns the window on if it was off; also allows the size of the new symbol table and the main program space to be specified.

FORMAT **new** [[*nos*]] [[**,** *mps*]]**;**

INPUT
nos scalar, which indicates the maximum number of global symbols allowed.

mps scalar, which indicates the number of bytes of main program space to be allocated. The second argument is obsolete, and included only for backwards compatibility. Memory is dynamically allocated.

REMARKS Procedures, user-defined functions, and global matrices, strings, and string arrays are all global symbols.

This command can be used with arguments as the first statement in a program to clear the symbol table and to allocate only as much space for program code as

your program actually needs. When used in this manner, the auxiliary output will not be closed. This will allow you to open the auxiliary output from the command level and run a program without having to remove the **new** at the beginning of the program. If this command is not the first statement in your program, it will cause the program to terminate.

EXAMPLE

```
new;                /* clear global symbols. */

new 300;            /* clear global symbols,set maximum
                    ** number of global symbols to 300,
                    ** and leave program space unchanged.
                    */
```

SEE ALSO **clear**, **delete**, **output**

nextindex

PURPOSE Returns the index of the next element or subarray in an array.

FORMAT *ni* = **nextindex**(*i*,*o*);

INPUT

i	M×1 vector of indices into an array, where M<=N.
o	N×1 vector of orders of an N-dimensional array.

OUTPUT

ni	M×1 vector of indices, the index of the next element or subarray in the array corresponding to *o*.

REMARKS **nextindex** will return a scalar error code if the index cannot be incremented.

EXAMPLE

```
a = ones(2520,1);
a = areshape(a,3|4|5|6|7);
orders = getorders(a);
```

```
ind = { 2,3,5 };
ind = nextindex(ind,orders);
```

```
      2
ind = 4
      1
```

In this example, **nextindex** incremented **ind** to index the next 6×7 subarray in array **a**.

SEE ALSO **previousindex**, **loopnextindex**, **walkindex**

nextn, nextnevn

PURPOSE Returns allowable matrix dimensions for computing FFT's.

FORMAT *n* **= nextn(***n0***);**

n **= nextnevn(***n0***);**

INPUT *n0* scalar, the length of a vector or the number of rows or columns in a matrix.

OUTPUT *n* scalar, the next allowable size for the given dimension for computing an FFT or RFFT. $n \geq n0$.

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 Function	Vector Length	Matrix Rows	Matrix Columns
fftn	**nextn**	**nextn**	**nextn**
rfftn	**nextnevn**	**nextn**	**nextnevn**
rfftnp	**nextnevn**	**nextn**	**nextnevn**

EXAMPLE

```
n = nextn(456);
```

```
n = 480.00000
```

SOURCE optim.src

SEE ALSO **fftn**, **optn**, **optnevn**, **rfftn**, **rfftnp**

nextwind

PURPOSE Set 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 GRAPHIC PANELS, Section 24.3.

SOURCE pwindow.src

SEE ALSO **endwind**, **begwind**, **setwind**, **getwind**, **makewind**, **window**

null

PURPOSE Computes an orthonormal basis for the (right) null space of a matrix.

FORMAT *b* **= null(***x***);**

INPUT *x* N×M matrix.

OUTPUT *b* M×K matrix, where K is the nullity of *x*, such that:

$$x * b = 0 \qquad \text{(N×K matrix of zeros)}$$

and

$$b'b = I \qquad \text{(M×M identity matrix)}$$

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**, **_qrsl**

null1

PURPOSE Computes an orthonormal basis for the (right) null space of a matrix.

FORMAT *nu* **= null1(***x***,***dataset***);**

INPUT

x	N×M matrix.
dataset	string, the name of a data set **null1** will write.

OUTPUT

nu	scalar, the nullity of *x*.

REMARKS **null1** computes an M×K matrix *b*, where K is the nullity of *x*, such that:

$$x * b \quad = \quad 0 \qquad \text{(N×K matrix of zeros)}$$

and

$$b'b \quad = \quad I \qquad \text{(M×M identity matrix)}$$

The transpose of *b* is written to the data set named by *dataset*, unless the nullity of *x* is zero. If *nu* is zero, the data set is not written.

SOURCE null.src

GLOBALS **_qrdc**, **_qrsl**

numCombinations

PURPOSE Computes number of combinations of *n* things taken *k* at a time.

FORMAT *y* = **numCombinations**(*n*,*k*);

INPUT
n scalar.
k scalar.

OUTPUT *y* scalar, number of combinations of *n* things take *k* at a time.

EXAMPLE
```
y = numCombinations(25,5);

print y;

53130.0000
```

SEE ALSO **combinate**, **combinated**

ols

PURPOSE Computes a least squares regression.

FORMAT { *vnam*,*m*,*b*,*stb*,*vc*,*stderr*,*sigma*,*cx*,*rsq*,*resid*,*dwstat* } =
ols(*dataset*,*depvar*,*indvars***);**

INPUT

dataset string, name of data set or null string.

If *dataset* is a null string, the procedure assumes that the actual data has been passed in the next two arguments.

depvar If *dataset* contains a string:

string, name of dependent variable
- or -
scalar, index of dependent variable. If scalar 0, the last column of the data set will be used.

If *dataset* is a null string or 0:

N×1 vector, the dependent variable.

indvars If *dataset* contains a string:

K×1 character vector, names of independent variables
- or -
K×1 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:

N×K matrix, the independent variables.

GLOBAL INPUT Defaults are provided for the following global input variables, so they can be ignored unless you need control over the other options provided by this procedure.

__altnam character vector, default 0.

This can be a (K+1)×1 or (K+2)×1 character vector of alternate variable names for the output. If **__con** is 1, this must be (K+2)×1. The name of the dependent variable is the last element.

__con scalar, default 1.

1 a constant term will be added, D = K+1.

0 no constant term will be added, D = K.

A constant term will always be used in constructing the moment matrix *m*.

__miss scalar, default 0.

0 there are no missing values (fastest).

1 listwise deletion, drop any cases in which missings occur.

2 pairwise deletion, this is equivalent to setting missings to 0 when calculating *m*. The number of cases computed is equal to the total number of cases in the data set.

__output scalar, default 1.

1 print the statistics.

0 do not print statistics.

__row scalar, the number of rows to read per iteration of the read loop. Default 0.

If 0, the number of rows will be calculated internally. If you get an **Insufficient memory** error while executing **ols**, you can supply a value for **__row** that works on your system.

The answers may vary slightly due to rounding error differences when a different number of rows is read per iteration. You can use **__row** to control this if you want to get exactly the same rounding effects between several runs.

_olsres scalar, default 0.

1 compute residuals (*resid*) and Durbin-Watson statistic (*dwstat*).

0 *resid* = 0, *dwstat* = 0.

OUTPUT *vnam* (K+2)×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.

m 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	*indvars*	*depvar*
(constant)	(independent variables)	(dependent variable)

A constant term is always used in computing *m*.

b D×1 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 K×1 vector, the standardized coefficients.

vc D×D matrix, the variance-covariance matrix of estimates.

stderr D×1 vector, the standard errors of the estimated parameters.

sigma scalar, standard deviation of residual.

cx (K+1)×(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 N×1 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 N×1 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**.

EXAMPLE

```
y = { 2,
      3,
      1,
      7,
      5 };

x = { 1 3 2,
      2 3 1,
      7 1 7,
      5 3 1,
      3 5 5 };

output file = ols.out reset;
call ols(0,y,x);
output off;
```

In this example, the output from **ols** is put into a file called `ols.out` as well as being printed to the window. This example will compute a least squares regression of **y** on **x**. The return values are discarded by using a **call** statement.

```
data = "olsdat";
depvar = { score };
indvars = { region,age,marstat };
_olsres = 1;
output file = lpt1 on;
{ nam,m,b,stb,vc,std,sig,cx,rsq,resid,dbw } =
    ols(data,depvar,indvars);
output off;
```

In this example, the data set `olsdat.dat` is used to compute a regression. The dependent variable is **score**. The independent variables are: **region**, **age**, and **marstat**. The residuals and Durbin-Watson statistic will be computed. The output will be sent to the printer as well as the window and the returned values are assigned to variables.

SOURCE `ols.src`

SEE ALSO **olsqr**

olsmt

PURPOSE Computes a least squares regression.

FORMAT *oout* = **olsmt**(*oc0*,*dataset*,*depvar*,*indvars*);

INPUT *oc0* instance of an **olsmtControl** structure containing the following members:

oc0`.altnam` character vector, default 0.
This can be a (K+1)×1 or (K+2)×1 character vector of alternate variable names for the output. If *oc0*.**con** is 1, this must be (K+2)×1. The name of the dependent variable is the last element.

oc0`.con` scalar, default 1.

1 a constant term will be added, D = K+1.

0 no constant term will be added, D = K.

A constant term will always be used in constructing the moment matrix *m*.

oc0`.miss` scalar, default 0.

0 there are no missing values (fastest).

1 listwise deletion, drop any cases in which missings occur.

2 pairwise deletion, this is equivalent to setting missings to 0 when calculating *m*. The number of cases computed is equal to the total number of cases in the data set.

oc0.row — scalar, the number of rows to read per iteration of the read loop. Default 0.
If 0, the number of rows will be calculated internally. If you get an **Insufficient memory** error message while executing **olsmt**, you can supply a value for *oc0*.**row** that works on your system.
The answers may vary slightly due to rounding error differences when a different number of rows is read per iteration. You can use *oc0*.**row** to control this if you want to get exactly the same rounding effects between several runs.

oc0.vpad — scalar, default 1.
If 0, internally created variable names are not padded to the same length (e.g. "X1, X2,..., X10").
If 1, they are padded with zeros to the same length (e.g., "X01, X02,..., X10").

oc0.output — scalar, default 1.

1 print the statistics.

0 do not print statistics.

oc0.res — scalar, default 0.

1 compute residuals (*resid*) and Durbin-Watson statistic (*dwstat*).

0 *oout*.**resid** = 0, *oout*.**dwstat** = 0.

oc0.rnam — string, default "_olsmtres".
If the data is taken from a data set, a new data set will be created for the residuals, using the

name in *oc0*.**rnam**.

oc0.maxvec — scalar, default 20000.
The largest number of elements allowed in any one matrix.

oc0.fcmptol — scalar, default 1e-12.
Tolerance used to fuzz the comparison operations to allow for round off error.

dataset string, name of data set or null string.

If *dataset* is a null string, the procedure assumes that the actual data has been passed in the next two arguments.

depvar If *dataset* contains a string:

string, name of dependent variable
- or -
scalar, index of dependent variable. If scalar 0, the last column of the data set will be used.

If *dataset* is a null string or 0:

N×1 vector, the dependent variable.

indvars If *dataset* contains a string:

K×1 character vector, names of independent variables
- or -
K×1 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:

N×K matrix, the independent variables.

OUTPUT *oout* instance of an **olsmtOut** structure containing the following members:

oout.vnam — (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.

oout.m

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	*indvars*	*depvar*
constant	independent variables	dependent variables

A constant term is always used in computing *m*.

oout.b

D×1 vector, the least squares estimates of parameters

Error handling is controlled by the low order bit of the trap flag.

trap 0 terminate with error message

trap 1 return scalar error code in *b*

- **30** system singular
- **31** system underdetermined
- **32** same number of columns as rows
- **33** too many missings
- **34** file not found
- **35** no variance in an independent variable

The system can become underdetermined if you use listwise deletion and have missing values. In that case, it is possible to skip so many cases that there are fewer useable rows than columns in the data set.

oout.stb

K×1 vector, the standardized coefficients.

oout.vc

D×D matrix, the variance-covariance matrix of estimates.

oout.stderr	D×1 vector, the standard errors of the estimated parameters.
oout.sigma	scalar, standard deviation of residual.
oout.cx	(K+1)×(K+1) matrix, correlation matrix of variables with the dependent variable as the last column.
oout.rsq	scalar, R square, coefficient of determination.
oout.resid	residuals, *oout*.**resid** = *y* - *x* * *oout*.**b**. If *oc0*.**olsres** = 1, the residuals will be computed. If the data is taken from a data set, a new data set will be created for the residuals, using the name in *oc0*.**rnam**. The residuals will be saved in this data set as an N×1 column. The *oout*.**resid** return value will be a string containing the name of the new data set containing the residuals. If the data is passed in as a matrix, the *oout*.**resid** return value will be the N×1 vector of residuals.
oout.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 **olsmt**.

EXAMPLE

```
#include olsmt.sdf

struct olsmtControl oc0;
struct olsmtOut oOut;
oc0 = olsmtControlCreate;

y = { 2,
      3,
      1,
      7,
```

```
      5 };

x = { 1 3 2,
      2 3 1,
      7 1 7,
      5 3 1,
      3 5 5 };

output file = olsmt.out reset;
oOut = olsmt(oc0,0,y,x);
output off;
```

In this example, the output from **olsmt** is put into a file called `olsmt.out` as well as being printed to the window. This example will compute a least squares regression of **y** on **x**.

```
#include olsmt.sdf

struct olsmtControl oc0;
struct olsmtOut oOut;
oc0 = olsmtControlCreate;

data = "olsdat";
depvar = { score };
indvars = { region,age,marstat };
oc0.res = 1;
output file = lpt1 on;
oOut = olsmt(oc0,data,depvar,indvars);
output off;
```

In this example, the data set `olsdat.dat` is used to compute a regression. The dependent variable is **score**. The independent variables are: **region**, **age**, and **marstat**. The residuals and Durbin-Watson statistic will be computed. The output will be sent to the printer as well as the window and the returned values are assigned to variables.

SOURCE olsmt.src

SEE ALSO **olsmtControlCreate**, **olsqrmt**

olsmtControlCreate

PURPOSE Creates default **olsmtControl** structure.

INCLUDE olsmt.sdf

FORMAT *c* = **olsmtControlCreate;**

OUTPUT *c* instance of an **olsmtControl** structure with members set to default values.

SOURCE olsmt.src

SEE ALSO **olsmt**

olsqr

PURPOSE Computes OLS coefficients using QR decomposition.

FORMAT *b* = **olsqr**(*y*,*x*);

INPUT *y* N×1 vector containing dependent variable.

x N×P matrix containing independent variables.

GLOBAL INPUT **_olsqtol** scalar, the tolerance for testing if diagonal elements are approaching zero. The default value is 10^{-14}.

OUTPUT *b* P×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

SEE ALSO **ols**, **olsqr2**, **orth**, **qqr**

olsqr2

PURPOSE Computes OLS coefficients, residuals, and predicted values using the QR decomposition.

FORMAT { *b*,*r*,*p* } = **olsqr2**(*y*,*x*);

INPUT *y* N×1 vector containing dependent variable.

x N×P matrix containing independent variables.

GLOBAL INPUT **_olsqtol** scalar, the tolerance for testing if diagonal elements are approaching zero. The default value is 10^{-14}.

OUTPUT *b* P×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.

r P×1 vector of residuals. ($r = y - x * b$)

	p	P×1 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

SEE ALSO **olsqr**, **orth**, **qqr**

olsqrmt

PURPOSE Computes OLS coefficients using QR decomposition.

FORMAT *b* = **olsqrmt**(*y*,*x*,*tol*);

INPUT	*y*	N×1 vector containing dependent variable.
	x	N×P matrix containing independent variables.
	tol	scalar, the tolerance for testing if diagonal elements are approaching zero. The default value is 10^{-14}.
OUTPUT	*b*	P×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.

olsqrmt handles matrices that do not have full rank by returning zeros for the coefficients that cannot be estimated.

SOURCE olsmt.src

SEE ALSO **olsmt**, **olsqr2**

ones

PURPOSE Creates a matrix of ones.

FORMAT *y* = **ones**(*r*,*c*);

INPUT
r scalar, number of rows.
c scalar, number of columns.

OUTPUT
y *r*×*c* matrix of ones.

REMARKS Noninteger arguments will be truncated to an integer.

EXAMPLE
```
x = ones(3,2);
```

```
      1.000000   1.000000
x =   1.000000   1.000000
      1.000000   1.000000
```

SEE ALSO **zeros**, **eye**

open

PURPOSE Opens an existing **GAUSS** data file.

FORMAT **open** *fh*=*filename* [[**for** *mode*]] [[**-w32**]] [[**varindxi** [[*offs*]]]];

INPUT *filename* literal or ^string.

filename is the name of the file on the disk. The name can include a path if the directory to be used is not the current directory. This filename will automatically be given the extension `.dat`. If an extension is specified, the `.dat` will be overridden. If the file is an `.fmt` matrix file, the extension must be explicitly given. If the name of the file is to be taken from a string variable, the name of the string must be preceded by the ^ (caret) operator.

mode literal, the modes supported with the optional **for** subcommand are:

read	This is the default file opening mode and will be the one used if none is specified. Files opened in this mode cannot be written to. The pointer is set to the beginning of the file and the **writer** function is disabled for files opened in this way. This is the only mode available for matrix files (`.fmt`), which are always written in one piece with the **save** command.
append	Files opened in this mode cannot be read. The pointer will be set to the end of the file so that a subsequent write to the file with the **writer** function will add data to the end of the file without overwriting any of the existing data in the file. The **readr** function is disabled for files opened in this way. This mode is used to add additional rows to the end of a file.
update	Files opened in this mode can be read from and written to. The pointer will be set to the

beginning of the file. This mode is used to make changes in a file.

offs scalar, offset added to "index variables".

The optional **varindxi** subcommand tells **GAUSS** to create a set of global scalars that contain the index (column position) of the variables in a **GAUSS** data file. These "index variables" will have the same names as the corresponding variables in the data file but with "i" added as a prefix. They can be used inside index brackets, and with functions like **submat** to access specific columns of a matrix without having to remember the column position.

The optional *offs* argument is an offset that will be added to the index variables. This is useful if data from multiple files are concatenated horizontally in one matrix. It can be any scalar expression. The default is 0.

The index variables are useful for creating submatrices of specific variables without requiring that the positions of the variables be known. For instance, if there are two variables, **xvar** and **yvar** in the data set, the index variables will have the names **ixvar**, **iyvar**. If **xvar** is the first column in the data file, and **yvar** is the second, and if no offset, *offs*, has been specified, then **ixvar** and **iyvar** will equal 1 and 2 respectively. If an offset of 3 had been specified, then these variables would be assigned the values 4 and 5 respectively.

The **-w32** flag is an optimization for Windows. It is ignored on all other platforms. **GAUSS** 7.0 uses Windows system file write commands that support 64-bit file sizes. These commands are slower on Windows XP than the 32-bit file write commands that were used in **GAUSS** 6.0. If you include the **-w32** flag, successive writes to the file indicated by *fh* will use 32-bit Windows write commands, which will be faster on Windows XP. Note, however, that the **-w32** flag does not support 64-bit file sizes.

The **varindxi** option cannot be used with **.fmt** matrix files because no column names are stored with them.

If **varindxi** is used, **GAUSS** will ignore the **Undefined symbol** error for global symbols that start with "i". This makes it much more convenient to use index variables because they don't have to be cleared before they are accessed in the program. Clearing is

otherwise necessary because the index variables do not exist until execution time when the data file is actually opened and the names are read in from the header of the file. At compile time a statement like: **y=x[.,ixvar];** will be illegal if the compiler has never heard of **ixvar**. If **varindxi** is used, this error will be ignored for symbols beginning with "i". Any symbols that are accessed before they have been initialized with a real value will be trapped at execution time with a **Variable not initialized** error.

OUTPUT *fh* scalar, file handle.

fh is the file handle which will be used by most commands to refer to the file within **GAUSS**. This file handle is actually a scalar containing an integer value that uniquely identifies each file. This value is assigned by **GAUSS** when the **open** command is executed. If the file was not successfully opened, the file handle will be set to -1.

REMARKS The file must exist before it can be opened with the **open** command. To create a new file, see **create** or **save**.

A file can be opened simultaneously under more than one handle. See the second example following.

If the value that is in the file handle when the **open** command begins to execute matches that of an already open file, the process will be aborted and a **File already open** message will be given. This gives you some protection against opening a second file with the same handle as a currently open file. If this happens, you would no longer be able to access the first file.

It is important to set unused file handles to zero because both **open** and **create** check the value that is in a file handle to see if it matches that of an open file before they proceed with the process of opening a file. This should be done with **close** or **closeall**.

EXAMPLE
```
fname = "/data/rawdat";
open dt = ^fname for append;
if dt =\,= -1;
```

```
    print "File not found";
    end;
endif;
y = writer(dt,x);
if y /= rows(x);
    print "Disk Full";
    end;
endif;
dt = close(dt);
```

In the example above, the existing data set `/data/rawdat.dat` is opened for appending new data. The name of the file is in the string variable **fname**. In this example the file handle is tested to see if the file was opened successfully. The matrix **x** is written to this data set. The number of columns in **x** must be the same as the number of columns in the existing data set. The first row in **x** will be placed after the last row in the existing data set. The **writer** function will return the number of rows actually written. If this does not equal the number of rows that were attempted, then the disk is probably full.

```
open fin = mydata for read;
open fout = mydata for update;
do until eof(fin);
    x = readr(fin,100);
    x[.,1 3] = ln(x[.,1 3];
    call writer(fout,x);
endo;
closeall fin,fout;
```

In the above example, the same file, `mydata.dat`, is opened twice with two different file handles. It is opened for read with the handle **fin**, and it is opened for update with the handle **fout**. This will allow the file to be transformed in place without taking up the extra space necessary for a separate output file. Notice that **fin** is used as the input handle and **fout** is used as the output handle. The loop will terminate as soon as the input handle has reached the end of the file. Inside the loop the file is read into a matrix called **x** using the input handle, the data are transformed (columns 1 and 3 are replaced with their

natural logs), and the transformed data is written back out using the output handle. This type of operation works fine as long as the total number of rows and columns does not change.

The following example assumes a data file named `dat1.dat` that has the variables: **visc**, **temp**, **lub**, and **rpm**:

```
open f1 = dat1 varindxi;
dtx = readr(f1,100);
x = dtx[.,irpm ilub ivisc];
y = dtx[.,itemp];
call seekr(f1,1);
```

In this example, the data set `dat1.dat` is opened for reading (the `.dat` and the **for read** are implicit). **varindxi** is specified with no constant. Thus, index variables are created that give the positions of the variables in the data set. The first 100 rows of the data set are read into the matrix **dtx**. Then, specified variables in a specified order are assigned to the matrices **x** and **y** using the index variables. The last line uses the **seekr** function to reset the pointer to the beginning of the file.

```
open q1 = c:dat1 varindx;
open q2 = c:dat2 varindx colsf(q1);
nr = 100;
y = readr(q1,nr)~readr(q2,nr);
closeall q1,q2;
```

In this example, two data sets are opened for reading and index variables are created for each. A constant is added to the indices for the second data set (**q2**), equal to the number of variables (columns) in the first data set (**q1**). Thus, if there are three variables **x1**, **x2**, **x3** in **q1**, and three variables **y1**, **y2**, **y3** in **q2**, the index variables that were created when the files were opened would be **ix1**, **ix2**, **ix3**, **iy1**, **iy2**, **iy3**. The values of these index variables would be 1, 2, 3, 4, 5, 6, respectively. The first 100 rows of the two data sets are read in and

concatenated to produce the matrix **y**. The index variables will thus give the correct positions of the variables in **y**.

```
open fx = x.fmt;
i = 1; rf = rowsf(fx);
sampsize = round(rf*0.1);
rndsmpx = zeros(sampsize,colsf(fx));
do until i > sampsize;
    r = ceil(rndu(1,1)*rf);
    call seekr(fx,r);
    rndsmpx[i,.] = readr(fx,1);
    i = i+1;
endo;
fx = close(fx);
```

In this example, a 10% random sample of rows is drawn from the matrix file `x.fmt` and put into the matrix **rndsmpx**. Note that the extension `.fmt` must be specified explicitly in the **open** statement. The **rowsf** command is used to obtain the number of rows in `x.fmt`. This number is multiplied by 0.10 and the result is rounded to the nearest integer; this yields the desired sample size. Then random integers (**r**) in the range 1 to **rf** are generated. **seekr** is used to locate to the appropriate row in the matrix, and the row is read with **readr** and placed in the matrix **rndsmpx**. This is continued until the complete sample has been obtained.

SEE ALSO **dataopen**, **create**, **close**, **closeall**, **readr**, **writer**, **seekr**, **eof**

optn, optnevn

PURPOSE Returns optimal matrix dimensions for computing FFT's.

FORMAT *n* **= optn(***n0***);**

n = **optnevn**(*n0*);

INPUT

n0 scalar, the length of a vector or the number of rows or columns in a matrix.

OUTPUT

n scalar, the next optimal size for the given dimension for computing an FFT or RFFT. $n \geq n0$.

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 33600×1 vector can compute as much as 20% faster than a 32768×1 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.

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* **= orth(***x***);**

INPUT *x* N×K matrix.

GLOBAL INPUT **_orthtol** scalar, the tolerance for testing if diagonal elements are approaching zero. The default is 1.0e-14.

OUTPUT *y* N×L matrix such that $y'y$ **= eye(**L**)** and whose columns span the same space as the columns of *x*; L is the rank of *x*.

EXAMPLE

```
x = { 6 5 4,
      2 7 5 };

y = orth(x);
```

```
y =  -0.58123819  -0.81373347
     -0.81373347   0.58123819
```

SOURCE qqr.src

SEE ALSO **qqr**, **olsqr**

output

PURPOSE

This command makes it possible to direct the output of **print** statements to two different places simultaneously. One output device is always the window or standard output. The other can be selected by the user to be any disk file or other suitable output device such as a printer.

FORMAT

output [[**file=***filename*]] [[**on**|**off**|**reset**]];

INPUT

filename literal or ^string.
The **file=***filename* subcommand selects the file or device to which output is to be sent.
If the name of the file is to be taken from a string variable, the name of the string must be preceded by the ^ (caret) operator.
The default file name is `output.out`.

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

on	opens the auxiliary output file or device and causes the results of all **print** statements to be sent to that file or device. If the file already exists, it will be opened for appending. If the file does not already exist, it will be created.
off	closes the auxiliary output file and turns off the auxiliary output.
reset	similar to the **on** subcommand, except that it always creates a new file. If the file already exists, it will be destroyed and a new file by that name will be created. If it does not exist, it will be created.

REMARKS

After you have written to an output file you have to close the file before you can print it or edit it with the **GAUSS** editor. Use **output off**.

The selection of the auxiliary output file or device remains in effect until a new selection is made, or until you get out of **GAUSS**. Thus, if a file is named as the

output device in one program, it will remain the output device in subsequent programs until a new **file=***filename* subcommand is encountered.

The command

```
output file=filename;
```

will select the file or device but will not open it. A subsequent **output on** or **output reset** will open it and turn on the auxiliary output.

The command **output off** will close the file and turn off the auxiliary output. The filename will remain the same. A subsequent **output on** will cause the file to be opened again for appending. A subsequent **output reset** will cause the existing file to be destroyed and then recreated and will turn on the auxiliary output.

The command **output** by itself will cause the name and status (i.e., open or closed) of the current auxiliary output file to be printed to the window.

The output to the console can be turned off and on using the **screen off** and **screen on** commands. Output to the auxiliary file or device can be turned off or on using the **output off** or **output on** command. The defaults are **screen on** and **output off**.

The auxiliary file or device can be closed by an explicit **output off** statement, by an **end** statement, or by an interactive **new** statement. However, a **new** statement at the beginning of a program will not close the file. This allows programs with **new** statements in them to be run without reopening the auxiliary output file.

If a program sends data to a disk file, it will execute much faster if the window is off.

The **outwidth** command will set the line width of the output file. The default is 80.

EXAMPLE

```
output file = out1.out on;
```

This statement will open the file out1.out and will cause the results of all subsequent **print** statements to be sent to that file. If out1.out already exists, the new output will be appended.

```
output file = out2.out;
output on;
```

This is equivalent to the previous example.

```
output reset;
```

This statement will create a new output file using the current filename. If the file already exists, any data in it will be lost.

```
output file = mydata.asc reset;
screen off;
format /m1/rz 1,8;
open fp = mydata;
do until eof(fp);
    print readr(fp,200);;
endo;
fp = close(fp);
end;
```

The program above will write the contents of the **GAUSS** file mydata.dat into an ASCII file called mydata.asc. If there had been an existing file by the name of mydata.asc, it would have been overwritten.

The **/m1** parameter in the **format** statement in combination with the **;;** at the end of the **print** statement will cause one carriage return/line feed pair to be written at the beginning of each row of the output file. There will not be an extra line feed added at the end of each 200 row block.

The **end** statement above will automatically perform **output off** and **screen on**.

SEE ALSO **outwidth**, **screen**, **end**, **new**

outtyp (dataloop)

PURPOSE Specifies the precision of the output data set.

FORMAT **outtyp** *num_constant*;

INPUT *num_constant* scalar, precision of output data set.

REMARKS *num_constant* must be 2, 4, or 8, to specify integer, single precision, or double precision, respectively.

If **outtyp** is not specified, the precison of the output data set will be that of the input data set. If character data is present in the data set, the precision will be forced to double.

EXAMPLE
```
outtyp 8;
```

outwidth

PURPOSE Specifies the width of the auxiliary output.

FORMAT **outwidth** *n*;

INPUT *n* scalar, width of auxilary output.

REMARKS *n* specifies the width of the auxiliary output in columns (characters). After printing *n* characters on a line, **GAUSS** will output a line feed.

If a matrix is being printed, the line feed sequence will always be inserted between separate elements of the matrix rather than being inserted between digits of a single element.

n may be any scalar-valued expressions in the range of 2-256. Nonintegers will be truncated to an integer. If 256 is used, no additional lines will be inserted.

The default is 80 columns.

EXAMPLE
```
outwidth 132;
```

This statement will change the auxiliary output width to 132 columns.

SEE ALSO **output**, **print**

pacf

PURPOSE Computes sample partial autocorrelations.

FORMAT *rkk* **= pacf(***y*,*k*,*d***);**

INPUT

y	N×1 vector, data.
k	scalar, maximum number of partial autocorrelations to compute.
d	scalar, order of differencing.

OUTPUT

rkk	K×1 vector, sample partial autocorrelations.

EXAMPLE
```
proc pacf(y,k,d);
    local a,l,j,r,t;
    r = acf(y,k,d);
```

```
    a = zeros(k,k);
    a[1,1] = r[1];

    t = 1;
    l = 2;

    do while l le k;
        a[l,l] = (r[l]-a[l-1,1:t]*rev(r[1:l-1]))/
                 (1-a[l-1,1:t]*r[1:t]);
        j = 1;

        do while j <= t;
            a[l,j] = a[l-1,j] - a[l,l]*a[l-1,l-j];
            j = j+1;
        endo;

        t = t+1;
        l = l+1;
    endo;

    retp(diag(a));
endp;
```

SOURCE tsutil.src

packedToSp

PURPOSE Creates a sparse matrix from a packed matrix of non-zero values and row and column indices.

FORMAT *y* = **packedToSp**(*r*,*c*,*p*);

INPUT

r	scalar, rows of output matrix.
c	scalar, columns of output matrix.
p	N×3 or N×4 matrix, containing non-zero values and row and column indices.

OUTPUT

y	*r*×*c* sparse matrix.

REMARKS

If *p* is N×3, *y* will be a real sparse matrix. Otherwise, if *p* is N×4, *y* will be complex.

The format for *p* is as follows:

If *p* is N×3:

Column 1	Column 2	Column 3
non-zero values	row indices	column indices

If *p* is N×4:

Column 1	Column 2	Column 3	Column 4
real non-zero values	imaginary non-zero values	row indices	column indices

Note that **spCreate** may be faster.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **packedToSp**.

EXAMPLE

```
sparse matrix y;
p = { 1 2 4, 2 5 1, 3 8 9, 4 13 5 };
y = packedToSp(15,10,p);
```

This example creates a 15×10 sparse matrix **y**, containing the following non-zero values:

Non-zero value	Index
1	(2,4)
2	(5,1)
3	(8,9)
4	(13,5)

SEE ALSO **spCreate**, **denseToSp**

packr

PURPOSE Deletes the rows of a matrix that contain any missing values.

FORMAT *y* = **packr**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* L×K 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
```
x = miss(ceil(rndu(3,3)*10),1);
y = packr(x);
```

```
      .  9  10
x =   4  2   .
      3  4   9


y =   3  4   9
```

In this example, the matrix **x** is formed with random integers and missing values. **packr** is used to delete rows with missing values.

```
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**, **missrv**

parse

PURPOSE Parses a string, returning a character vector of tokens.

FORMAT *tok* **= parse(***str***,***delim***);**

INPUT *str* string consisting of a series of tokens and/or delimiters.

delim N×K character matrix of delimiters that might be found in *str*.

OUTPUT *tok* M×1 character vector consisting of the tokens contained in *str*. All tokens are returned; any delimiters found in *str* are ignored.

REMARKS The tokens in *str* must be 8 characters or less in size. If they are longer, the contents of *tok* is unpredictable.

SEE ALSO **token**

pause

PURPOSE Pauses for a specified number of seconds.

FORMAT **pause(***sec***);**

INPUT *sec* scalar, seconds to pause.

SOURCE pause.src

SEE ALSO **wait**

pdfn

PURPOSE Computes the standard Normal (scalar) probability density function.

FORMAT *y* **= pdfn(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* N×K matrix containing the standard Normal probability density function of *x*.

REMARKS This does not compute the joint Normal density function. Instead, the scalar Normal density function is computed element-by-element. *y* could be computed by the following **GAUSS** code:

```
y = (1/sqrt(2*pi))*exp(-(x.*x)/2);
```

EXAMPLE
```
x = rndn(2,2);
y = pdfn(x);
```

```
x =  −1.828915   0.514485
     −0.550219  −0.275229

y =  0.074915   0.349488
     0.342903   0.384115
```

pi

PURPOSE Returns the mathematical constant π.

FORMAT *y* **= pi;**

OUTPUT *y* scalar, the value of π.

EXAMPLE
```
format /rdn 16,14;
print pi;

3.14159265358979
```

pinv

PURPOSE Computes the Moore-Penrose pseudo-inverse of a matrix, using the singular value decomposition.

This pseudo-inverse is one particular type of generalized inverse.

FORMAT *y* **= pinv(***x***);**

INPUT *x* N×M matrix.

GLOBAL INPUT **_svdtol** scalar, any singular values less than **_svdtol** are treated as zero in determining the rank of the input matrix. The default value for **_svdtol** is 1.0e-13.

OUTPUT *y* M×N matrix that satisfies the 4 Moore-Penrose conditions:

$xyx = x$

$yxy = y$

xy is symmetric

yx is symmetric

GLOBAL OUTPUT **_svderr** scalar, if not all of the singular values can be computed **_svderr** will be nonzero.

EXAMPLE
```
x = { 6 5 4, 2 7 5 };
y = pinv(x);
```

```
          0.22017139   −0.16348055
y =  −0.052076467    0.13447594
     −0.015161503   0.077125906
```

SOURCE svd.src

pinvmt

PURPOSE Computes the Moore-Penrose pseudo-inverse of a matrix, using the singular value decomposition.

This pseudo-inverse is one particular type of generalized inverse.

FORMAT { *y*,*err* } = **pinvmt**(*x*,*tol*);

INPUT

x	N×M matrix.
tol	scalar, any singular values less than *tol* are treated as zero in determining the rank of the input matrix.

OUTPUT

y	M×N matrix that satisfies the 4 Moore-Penrose conditions: $xyx = x$ $yxy = y$ xy is symmetric yx is symmetric
err	scalar, if not all of the singular values can be computed *err* will be nonzero.

EXAMPLE

```
x = { 6 5 4, 2 7 5 };
tol = 1e-13;
{ y,err } = pinvmt(x,tol);
```

```
            0.22017139  -0.16348055
y =  -0.052076467   0.13447594
     -0.015161503  0.077125906


err = 0
```

SOURCE svdmt.src

polar

PURPOSE Graph data using polar coordinates.

LIBRARY pgraph

FORMAT **polar**(*radius*,*theta*);

INPUT
radius N×1 or N×M matrix. Each column contains the magnitude for a particular line.

theta N×1 or N×M matrix. Each column represents the angle values for a particular line.

SOURCE polar.src

SEE ALSO **xy**, **logx**, **logy**, **loglog**, **scale**, **xtics**, **ytics**

polychar

PURPOSE Computes the characteristic polynomial of a square matrix.

FORMAT *c* = **polychar**(*x*);

INPUT
x N×N matrix.

OUTPUT
c (N+1)×1 vector of coefficients of the N^{th} order characteristic polynomial of *x*:

$$p(x) = c[1] * x^n + c[2] * x^{(n-1)} + \ldots + c[n] * x + c[n+1];$$

REMARKS The coefficient of x^n is set to unity ($c[1]=1$).

SOURCE poly.src

SEE ALSO **polymake**, **polymult**, **polyroot**, **polyeval**

polyeval

PURPOSE Evaluates polynomials. Can either be one or more scalar polynomials or a single matrix polynomial.

FORMAT *y* = **polyeval**(*x*,*c*);

INPUT

x 1×K or N×N; that is, *x* can either represent K separate scalar values at which to evaluate the (scalar) polynomial(s), or it can represent a single N×N matrix.

c (P+1)×K or (P+1)×1 matrix of coefficients of polynomials to evaluate. If *x* is 1×K, then *c* must be (P+1)×K. If *x* is N×N, *c* must be (P+1)×1. That is, if *x* is a matrix, it can only be evaluated at a single set of coefficients.

OUTPUT

y K×1 vector (if *c* is (P+1)×K) or N×N matrix (if *c* is (P+1)×1 and *x* is N×N):

$$y = (c[1,.] * x^{p} + c[2,.] * x^{(p-1)} + \ldots + c[p+1,.])';$$

REMARKS In both the scalar and the matrix case, Horner's rule is used to do the evaluation. In the scalar case, the function **recsercp** is called (this implements an elaboration of Horner's rule).

EXAMPLE

```
x = 2;
let c = 1 1 0 1 1;
y = polyeval(x,c);
```

The result is 27. Note that this is the decimal value of the binary number 11011.

```
y = polyeval(x,1|zeros(n,1));
```

This will raise the matrix *x* to the n^{th} power (e.g: *x**x*x*x*x*... **x*).

SOURCE poly.src

SEE ALSO **polymake**, **polychar**, **polymult**, **polyroot**

polyint

PURPOSE Calculates an N^{th} order polynomial interpolation.

FORMAT *y* **= polyint(***xa*,*ya*,*x***);**

INPUT

xa	N×1 vector, *X* values.
ya	N×1 vector, *Y* values.
x	scalar, *X* value to solve for.

GLOBAL INPUT

_poldeg	scalar, the degree of polynomial required, default 6.

OUTPUT

y	result of interpolation or extrapolation.

GLOBAL OUTPUT

_polerr	scalar, interpolation error.

REMARKS Calculates an 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. Teukolsky, and W.T. Vettering. *Numerical Recipes: The Art of Scientific Computing*. NY: Cambridge Press, 1986.

polymake

PURPOSE Computes the coefficients of a polynomial given the roots.

FORMAT *c* **= polymake(***r***);**

INPUT *r* N×1 vector containing roots of the desired polynomial.

OUTPUT *c* (N+1)×1 vector containing the coefficients of the N^{th} order polynomial with roots *r*:

$$p(z) = c[1] * z^n + c[2] * z^{(n-1)} + \ldots + c[n] * z + c[n+1];$$

REMARKS The coefficient of z^n is set to unity ($c[1]$=1).

EXAMPLE
```
r = { 2, 1, 3 };
c = polymake(r);
```

```
c =  1.0000000
    -6.0000000
     11.000000
    -6.0000000
```

SOURCE `poly.src`

SEE ALSO **polychar**, **polymult**, **polyroot**, **polyeval**

polymat

PURPOSE Returns a matrix containing the powers of the elements of *x* from 1 to *p*.

FORMAT *y* = **polymat**(*x*,*p*);

INPUT
x N×K matrix.
p scalar, positive integer.

OUTPUT
y N×(*p**K) matrix containing powers of the elements of *x* from 1 to *p*. The first K columns will contain first powers, the second K columns second powers, and so on.

REMARKS To do polynomial regression use **ols**:

```
{ vnam,m,b,stb,vc,stderr,sigma,cx,rsq,resid,dwstat } =
            ols(0,y,polymat(x,p));
```

SOURCE polymat.src

polymroot

PURPOSE Computes the roots of the determinant of a matrix polynomial.

FORMAT *r* = **polymroot**(*c*);

INPUT
c (N+1)*K×K matrix of coefficients of an N^{th} order polynomial of rank K.

OUTPUT *r* K*N vector containing the roots of the determinantal equation.

REMARKS *c* is constructed of N+1 K×K coefficient matrices stacked vertically with the coefficient matrix of the t^n at the top, $t^{(n-1)}$ next, down to the t^0 matrix at the bottom.

Note that this procedure solves the scalar problem as well, that is, the one that POLYROOT solves.

EXAMPLE Solve $det(A2 * t^2 + A1 * t + A0) = 0$ where:

$$A2 = \begin{matrix} 1 & 2 \\ 2 & 1 \end{matrix}$$

$$A1 = \begin{matrix} 5 & 8 \\ 10 & 7 \end{matrix}$$

$$A0 = \begin{matrix} 3 & 4 \\ 6 & 5 \end{matrix}$$

```
a2 = { 1 2, 2 1 };
a1 = { 5 8, 10 7 };
a0 = { 3 4, 6 5 };

print polymroot(a2|a1|a0);

   -4.3027756
   -.69722436
   -2.6180340
   -.38196601
```

polymult

PURPOSE Multiplies polynomials.

FORMAT *c* = **polymult**(*c1*,*c2*);

INPUT
c1 (D1+1)×1 vector containing the coefficients of the first polynomial.
c2 (D2+1)×1 vector containing the coefficients of the second polynomial.

OUTPUT
c (D1+D2)×1 vector containing the coefficients of the product of the two polynomials.

EXAMPLE
```
c1 = { 2, 1 };
c2 = { 2, 0, 1 };
c = polymult(c1,c2);
```

```
c =
4.0000000
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 $5 * x^3 + 6 * x + 3$ would be: *c1*=5| 0| 6| 3. (Note that zeros must be explicitly given if there are powers of *x* missing.)

polyroot

PURPOSE Computes the roots of a polynomial given the coefficients.

FORMAT *y* **= polyroot(***c***);**

INPUT *c* (N+1)×1 vector of coefficients of an N^{th} order polynomial:

$$p(z) = c[1] * z^n + c[2] * z^{(n-1)} + \ldots + c[n] * z + c[n+1]$$

OUTPUT *y* N×1 vector, the roots of *c*.

REMARKS Zero leading terms will be stripped from *c*. When that occurs the order of *y* will be the order of the polynomial after the leading zeros have been stripped.

c[1] need not be normalized to unity.

SOURCE poly.src

SEE ALSO **polymake**, **polychar**, **polymult**, **polyeval**

pop

PURPOSE Provides access to a last-in, first-out stack for matrices.

FORMAT **pop** *b***; pop** *a***;**

REMARKS This is used with **gosub**, **goto**, and **return** statements with parameters. It permits passing parameters to subroutines or labels, and returning parameters from subroutines.

The **gosub** syntax allows an implicit **push** statement. This syntax is almost the same as that of a standard **gosub**, except that the matrices to be **push**'ed "into the subroutine" are in parentheses following the label name. The matrices to be **push**'ed back to the main body of the program are in parentheses following the **return** statement. The only limit on the number of matrices that can be passed to and from subroutines in this way is the amount of room on the stack.

No matrix expressions can be executed between the (implicit) **push** and the **pop**. Execution of such expressions will alter what is on the stack.

Matrices must be **pop**'ped in the reverse order that they are **push**'ed, therefore in the statements:

```
goto label(x,y,z);
  .
  .
  .
label:
    pop c;
    pop b;
    pop a;
```

c = z $b = y$ $a = x$

Note that there must be a separate **pop** statement for each matrix popped.

SEE ALSO **gosub**, **goto**, **return**

pqgwin

PURPOSE Sets the graphics viewer mode.

LIBRARY pgraph

FORMAT **pqgwin one;**

pqgwin many;

REMARKS If you call **pqgwin one**, only a single viewer will be used. If you call **pqgwin many**, a new viewer will be used for each graph.

pqgwin manual and **pqgwin auto** are supported for backwards compatibility, **manual**=**one**, **auto**=**many**.

EXAMPLE
```
pqgwin many;
```

SOURCE pgraph.src

SEE ALSO **setvwrmode**

previousindex

PURPOSE Returns the index of the previous element or subarray in an array.

FORMAT *pi* = **previousindex**(*i*,*o*);

INPUT
i M×1 vector of indices into an array, where M<=N.

o N×1 vector of orders of an N-dimensional array.

OUTPUT
pi M×1 vector of indices, the index of the previous element or subarray in the array corresponding to *o*.

REMARKS **previousindex** will return a scalar error code if the index cannot be decremented.

EXAMPLE

```
orders = {3,4,5,6,7};
a = areshape(1,orders);
orders = getorders(a);
ind = { 2,3,1 };
ind = previousindex(ind,orders);
```

```
        2
ind =   2
        5
```

In this example, **previousindex** decremented **ind** to index the previous 6×7 subarray in array **a**.

SEE ALSO **nextindex**, **loopnextindex**, **walkindex**

princomp

PURPOSE Computes principal components of a data matrix.

FORMAT { *p*,*v*,*a* } **= princomp(***x*,*j***);**

INPUT

x	N×K data matrix, N>K, full rank.
j	scalar, number of principal components to be computed (*j*<=K).

OUTPUT

p	N×J matrix of the first *j* principal components of *x* in descending order of amount of variance explained.
v	J×1 vector of fractions of variance explained.
a	J×K matrix of factor loadings, such that *x* = *p***a* +error.

REMARKS Adapted from a program written by Mico Loretan.

The algorithm is based on Theil, Henri "Principles of Econometrics." Wiley, NY, 1971, 46-56.

print

PURPOSE Prints matrices, arrays, strings and string arrays to the screen and/or auxiliary output.

FORMAT **print** [[**/flush**]] [[*/typ*]] [[*/fmted*]] [[*/mf*]] [[*/jnt*]] *list_of_expressions*[[**;**]]**;**

INPUT */typ* literal, symbol type flag.

/mat, /sa, /str Indicate which symbol types you are setting the output format for: matrices and arrays (**/mat**), string arrays (**/sa**), and/or strings (**/str**). You can specify more than one */typ* flag; the format will be set for all types indicated. If no */typ* flag is listed, **print** assumes **/mat**.

/fmted literal, enable formatting flag.

/on, /off Enable/disable formatting. When formatting is disabled, the contents of a variable are dumped to the screen in a "raw" format. **/off** is currently supported only for strings. "Raw" format for strings means that the entire string is printed, starting at the current cursor position. When formatting is enabled for strings, they are handled the same as string arrays. This shouldn't be too surprising, since a string is actually a 1×1 string array.

/mf literal, matrix format. It controls the way rows of a matrix are separated from one another. The possibilities are:

/m0 no delimiters before or after rows when printing out matrices.

/m1 or **/mb1**	print 1 carriage return/line feed pair before each row of a matrix with more than 1 row.
/m2 or **/mb2**	print 2 carriage return/line feed pairs before each row of a matrix with more than 1 row.
/m3 or **/mb3**	print "Row 1", "Row 2"...before each row of a matrix with more than one row.
/ma1	print 1 carriage return/line feed pair after each row of a matrix with more than 1 row.
/ma2	print 2 carriage return/line feed pairs after each row of a matrix with more than 1 row.
/a1	print 1 carriage return/line feed pair after each row of a matrix.
/a2	print 2 carriage return/line feed pairs after each row of a matrix.
/b1	print 1 carriage return/line feed pair before each row of a matrix.
/b2	print 2 carriage return/line feed pairs before each row of a matrix.
/b3	print "Row 1", "Row 2"... before each row of a matrix.

/jnt literal, controls justification, notation, and the trailing character.

Right-Justified

/rd	Signed decimal number in the form [[-]]####.####, where #### is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed.
/re	Signed number in the form [[-]]#.##E±###, where # is one decimal digit, ## is one or more decimal digits depending on the precision, and ### is three decimal digits. If precision is 0, the form will be [[-]]#E±### with no decimal point printed.

/ro	This will give a format like **/rd** or **/re** depending on which is most compact for the number being printed. A format like **/re** will be used only if the exponent value is less than -4 or greater than the precision. If a **/re** format is used, a decimal point will always appear. The precision signifies the number of significant digits displayed.
/rz	This will give a format like **/rd** or **/re** depending on which is most compact for the number being printed. A format like **/re** will be used only if the exponent value is less than -4 or greater than the precision. If a **/re** format is used, trailing zeros will be supressed and a decimal point will appear only if one or more digits follow it. The precision signifies the number of significant digits displayed.
Left-Justified	
/ld	Signed decimal number in the form [[-]]####.####, where #### is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed. If the number is positive, a space character will replace the leading minus sign.
/le	Signed number in the form [[-]]#.##E±###, where # is one decimal digit, ## is one or more decimal digits depending on the precision, and ### is three decimal digits. If precision is 0, the form will be [[-]]#E±### with no decimal point printed. If the number is positive, a space character will replace the leading minus sign.
/lo	This will give a format like **/ld** or **/le** depending on which is most compact for the number being printed. A format like **/le** will be used only if the exponent value is less than -4 or greater than the

precision. If a **/le** format is used, a decimal point will always appear. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.

/lz This will give a format like **/ld** or **/le** depending on which is most compact for the number being printed. A format like **/le** will be used only if the exponent value is less than -4 or greater than the precision. If a **/le** format is used, trailing zeros will be supressed and a decimal point will appear only if one or more digits follow it. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.

Trailing Character

The following characters can be added to the */jnt* parameters above to control the trailing character if any:

```
format /rdn 1,3;
```

s The number will be followed immediately by a space character. This is the default.

c The number will be followed immediately by a comma.

t The number will be followed immediately by a tab character.

n No trailing character.

The default when **GAUSS** is first started is:

```
format /m1 /ro 16,8;
```

;; Double semicolons following a **print** statement will suppress the final carriage return/line feed.

list_of_expressions any **GAUSS** expressions that produce matrices, arrays, stings, or string arrays and/or names of variables to print, separated by spaces.

REMARKS

The list of expressions MUST be separated by spaces. In **print** statements, because a space is the delimiter between expressions, NO SPACES are allowed inside expressions unless they are within index brackets, quotes, or parentheses.

The printing of special characters is accomplished by the use of the backslash (\) within double quotes. The options are:

\b	backspace (ASCII 8)
\e	escape (ASCII 27)
\f	form feed (ASCII 12)
\g	beep (ASCII 7)
\l	line feed (ASCII 10)
\r	carriage return (ASCII 13)
\t	tab (ASCII 9)
\###	the character whose ASCII value is "###" (decimal).

Thus, **\13\10** is a carriage return/line feed sequence. The first three digits will be picked up here. So if the character to follow a special character is a digit, be sure to use three digits in the escape sequence. For example: **\0074** will be interpreted as 2 characters (ASCII 7, "4")

An expression with no assignment operator is an implicit **print** statement.

If **output on** has been specified, then all subsequent **print** statements will be directed to the auxiliary output as well as the window. (See **output**.) The **locate** statement has no effect on what will be sent to the auxiliary output, so all formatting must be accomplished using tab characters or some other form of serial output.

If the name of the symbol to be printed is prefixed with a **$**, it is assumed that the symbol is a matrix of characters.

```
print $x;
```

Note that **GAUSS** makes no distinction between matrices containing character data and those containing numeric data, so it is the responsibility of the user to use functions which operate on character matrices only on those matrices containing character data.

These matrices of character strings have a maximum of 8 characters per element. A precision of 8 or more should be set when printing out character matrices or the elements will be truncated.

Complex numbers are printed with the sign of the imaginary half separating them and an "i" appended to the imaginary half. Also, the current field width setting (see **format**) refers to the width of field for each half of the number, so a complex number printed with a field of 8 will actually take (at least) 20 spaces to print.

print'ing a sparse matrix results in a table of the non-zero values contained in the sparse matrix, followed by their corresponding row and column indices, respectively.

A **print** statement by itself will cause a blank line to be printed:

```
print;
```

EXAMPLE

```
x = rndn(3,3);
format /rd 16,8;
print x;

format /re 12,2;
print x;
print /rd/m3 x;
```

```
      0.14357994      -1.39272762      -0.91942414
      0.51061645      -0.02332207      -0.02511298
     -1.54675893      -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
```

In this example, a 3×3 random matrix is printed using 3 different formats. Notice that in the last statement, the format is overridden in the **print** statement itself but the field and precision remain the same.

```
let x = AGE PAY SEX;
format /m1 8,8;
print $x;

      AGE
      PAY
      SEX
```

SEE ALSO **printfm**, **printdos**

printdos

PURPOSE Prints a string to the standard output.

FORMAT **printdos** *s*;

INPUT *s* string to be printed to the standard output.

REMARKS This function is useful for printing messages to the screen when **screen off** is in effect. The output of this function will not go to the auxiliary output.

This function was used in the past to send escape sequences to the `ansi.sys` device driver on DOS. It still works on some terminals.

EXAMPLE
```
printdos "\27[7m";  /* set for reverse video */
```

```
printdos "\27[0m";  /* set for normal text   */
```

SEE ALSO **print**, **printfm**, **screen**

printfm

PURPOSE Prints a matrix using a different format for each column of the matrix.

FORMAT *y* = **printfm**(*x*,*mask*,*fmt*);

INPUT

x	N×K matrix which is to be printed and which may contain both character and numeric data.
mask	L×M matrix, E×E conformable with *x*, containing ones and zeros, which is used to specify whether the particular row, column, or element is to be printed as a character (0) or numeric (1) value.
fmt	K×3 or 1×3 matrix where each row specifies the format for the respective column of *x*.

OUTPUT

y	scalar, 1 if the function is successful and 0 if it fails.

REMARKS The mask is applied to the matrix *x* following the rules of standard element-by-element operations. If the corresponding element of *mask* is 0, then that element of *x* is printed as a character string of up to 8 characters. If mask contains a 1, then that element of *x* is assumed to be a double precision floating point number.

The contents of *fmt* are as follows:

[K,1]	format string,	a string 8 characters maximum.
[K,2]	field width,	a number < 80.
[K,3]	precision,	a number < 17.

The format strings correspond to the **format** slash commands as follows:

```
/rdn    ''*.*lf''
/ren    ''*.*lE''
/ron    ''#*.*lG''
/rzn    ''*.*lG''

/ldn    ''- *.*lf''
/len    ''- *.*lE''
/lon    ''-# *.*lG''
/lzn    ''- *.*lG''
```

Complex numbers are printed with the sign of the imaginary half separating them and an "i" appended to the imaginary half. The field width refers to the width of field for each half of the number, so a complex number printed with a field of 8 will actually take (at least) 20 spaces to print.

If the precision = 0, the decimal point will be suppressed.

The format string can be a maximum of 8 characters and is appended to a **%** sign and passed directly to the **fprintf** function in the standard C language I/O library. The **lf**, etc., are case sensitive. If you know C, you will easily be able to use this.

If you want special characters to be printed after *x*, then include them as the last characters of the format string. For example

''*.*lf,''	right-justified decimal followed by a comma.
''-*.*s ''	left-justified string followed by a space.
''*.*lf''	right-justified decimal followed by nothing.

If you want the beginning of the field padded with zeros, then put a "**0**" before the first "*****" in the format string:

''0*.*lf''	right-justified decimal.

EXAMPLE Here is an example of **printfm** being used to print a mixed numeric and character matrix:

```
let x[4,3] =
"AGE" 5.12345564 2.23456788
"PAY" 1.23456677 1.23456789
"SEX" 1.14454345 3.44718234
"JOB" 4.11429432 8.55649341;

let mask[1,3] = 0 1 1; /* character numeric numeric */

let fmt[3,3] =
"-*.*s " 8 8   /* first column format */
"*.*lf," 10 3  /* second column format */
"*.*le " 12 4; /* third column format */

d = printfm(x,mask,fmt);
```

The output looks like this:

```
AGE           5.123,  2.2346E+00
PAY           1.235,  1.2346E+00
SEX           1.145,  3.4471E+00
JOB           4.114,  8.5564E+00
```

When the column of *x* to be printed contains all character elements, use a format string of **''*.*s''** if you want it right-justified, or **''-*.*s''** if you want it left-justified. If the column is mixed character and numeric elements, then use the correct numeric format and **printfm** will substitute a default format string for those elements in the column that are character.

Remember, the mask value controls whether an element will be printed as a number or a character string.

SEE ALSO **print**, **printdos**

printfmt

PURPOSE Prints character, numeric, or mixed matrix using a default format controlled by the functions **formatcv** and **formatnv**.

FORMAT *y* = **printfmt**(*x*,*mask*);

INPUT

x N×K matrix which is to be printed.

mask scalar, 1 if *x* is numeric or 0 if *x* is character.

- or -

1×K vector of 1's and 0's.

The corresponding column of *x* will be printed as numeric where *mask* = 1 and as character where *mask* = 0.

OUTPUT

y scalar, 1 if the function is successful and 0 if it fails.

REMARKS Default format for numeric data is: **''*.*lg '' 16 8**

Default format for character data is: **''*.*s '' 8 8**

EXAMPLE

```
x = rndn(5,4);
call 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 */
```

Procedures can be used just as if they were functions intrinsic to the language. Below are the possible variations depending on the number of items the procedure returns.

Returns 1 item:

```
y = dog(i,j,k);
```

Returns multiple items:

```
{ x,y,z } = cat(i,j,k);
```

Returns no items:

```
fish(i,j,k);
```

If the procedure does not return any items or you want to discard the returned items:

```
call dog(i,j,k);
```

Procedure definitions may not be nested.

For more details on writing procedures, see PROCEDURES AND KEYWORDS, Chapter 12.

SEE ALSO **keyword**, **call**, **endp**, **local**, **retp**

prodc

PURPOSE Computes the products of all elements in each column of a matrix.

FORMAT *y* = **prodc**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* K×1 matrix containing the products of all elements in each column of *x*.

REMARKS To find the products of the elements in each row of a matrix, transpose before applying **prodc**. If *x* is complex, use the bookkeeping transpose (**.'**).

To find the products of all of the elements in a matrix, use the **vecr** function before applying **prodc**.

EXAMPLE
```
let x[3,3] = 1 2 3
             4 5 6
             7 8 9;

y = prodc(x);
```

```
          28
y =       80
         162
```

SEE ALSO **sumc**, **meanc**, **stdc**

putarray

PURPOSE Puts a contiguous subarray into an N-dimensional array and returns the resulting array.

FORMAT *y* = **putarray**(*a*,*loc*,*src*);

INPUT

a N-dimensional array.

loc M×1 vector of indices into the array to locate the subarray of interest, where M is a value from 1 to N.

src [N-M]-dimensional array, matrix, or scalar.

OUTPUT

y N-dimensional array.

REMARKS

If *loc* is an N×1 vector, then *src* must be a scalar. If *loc* is an [N-1]×1 vector, then *src* must be a 1-dimensional array or a 1×L vector, where L is the size of the fastest moving dimension of the array. If *loc* is an [N-2]×1 vector, then *src* must be a K×L matrix, or a K×L 2-dimensional array, where K is the size of the second fastest moving dimension.

Otherwise, if *loc* is an M×1 vector, then *src* must be an [N-M]-dimensional array, whose dimensions are the same size as the corresponding dimensions of array *a*.

EXAMPLE

```
a = arrayalloc(2|3|4|5|6,0);
src = arrayinit(4|5|6,5);
loc = { 2,1 };
a = putarray(a,loc,src);
```

This example sets the contiguous 4×5×6 subarray of **a** beginning at [2,1,1,1,1] to the array **src**, in which each element is set to the specified value 5.

SEE ALSO

setarray

putf

PURPOSE

Writes the contents of a string to a file.

FORMAT

ret **= putf(***filename***,***str***,***start***,***len***,***mode***,***append***);**

INPUT

filename	string, name of output file.
str	string to be written to *filename*. All or part of *str* may be written out.
start	scalar, beginning position in *str* of output string.
len	scalar, length of output string.
mode	scalar, output mode, (0) ASCII or (1) binary.
append	scalar, file write mode, (0) overwrite or (1) append.

OUTPUT *ret* scalar, return code.

0	normal return
1	null file name
2	file open error
3	file write error
4	output string too long
5	null output string, or illegal *mode* value
6	illegal *append* value
16	(1) append specified but file did not exist; file was created (warning only)

REMARKS If *mode* is set to (1) binary, a string of length *len* will be written to *filename*. If *mode* is set to (0) ASCII, the string will be output up to length *len* or until **putf** encounters a ^Z (ASCII 26) in *str*. The ^Z will not be written to *filename*.

If *append* is set to (0) overwrite, the current contents of *filename* will be destroyed. If *append* is set to (1) append, *filename* will be created if it does not already exist.

If an error occurs, **putf** will either return an error code or terminate the program with an error message, depending on the **trap** state. If bit 2 (the 4's bit) of the trap flag is 0, **putf** will terminate with an error message. If bit 2 of the trap flag is 1, **putf** will return an error code. The value of the trap flag can be tested with **trapchk**.

SOURCE `putf.src`

SEE ALSO **getf**

putvals

PURPOSE Inserts values into a matrix or N-dimensional array.

FORMAT *y* = **putvals**(*x*,*inds*,*vals*);

INPUT

x	M×K matrix or N-dimensional array.
inds	L×D matrix of indices, specifying where the new values are to be inserted, where D is the number of dimensions in *x*.
vals	L×1 vector, new values to insert.

OUTPUT

y	M×K matrix or N-dimensional array, copy of *x* containing the new values in *vals*.

REMARKS If *x* is a vector, *inds* should be an L×1 vector. If *x* is a matrix, *inds* should be an L×2 matrix. Otherwise if *x* is an N-dimensional array, *inds* should be an L×N matrix.

putvals allows you to insert multiple values into a matrix or N-dimensional array at one time. This could also be accomplished using indexing inside a **for** loop.

EXAMPLE

```
x = { -0.8750   0.3616   0.6032  -0.3974,
       0.7644  -1.8509  -0.2703  -0.8190,
       0.7886   1.2678  -1.4998  -0.5876,
       0.6639  -0.7972   1.2713   0.1896,
       0.6303   0.7879  -0.7451  -0.5419 };

inds = { 1 1, 2 4, 3 2, 3 4, 5 3 };
v = seqa(1,1,5);
y = putvals(x,inds,v);
```

```
          1.0000    0.3616    0.6032   −0.3974
          0.7644   −1.8509   −0.2703    2.0000
y =       0.7886    3.0000   −1.4998    4.0000
          0.6639   −0.7972    1.2713    0.1896
          0.6303    0.7879    5.0000   −0.5419
```


pvCreate

PURPOSE Returns an initialized an instance of structure of type **PV**.

FORMAT *p1* **= pvCreate;**

OUTPUT *p1* an instance of structure of type **PV**

EXAMPLE
```
struct PV p1;
p1 = pvCreate;
```

SOURCE pv.src

pvGetIndex

PURPOSE Gets row indices of a matrix in a parameter vector.

FORMAT *id* **= pvGetIndex(***p1***,***nm1***);**

INPUT *p1* an instance of structure of type **PV**.

nm1 name or row number of matrix.

OUTPUT *id* K×1 vector, row indices of matrix described by *nm1* in parameter vector.

SOURCE pv.src

pvGetParNames

PURPOSE Generates names for parameter vector stored in structure of type **PV**.

INCLUDE pv.sdf

FORMAT *s* = **pvGetParNames**(*p1*);

INPUT *p1* an instance of structure of type **PV**.

OUTPUT *s* K×1 string array, names of parameters.

REMARKS If the vector in the structure of type **PV** was generated with matrix names, the parameter names will be concatenations of the matrix name with row and column numbers of the parameters in the matrix. Otherwise the names will have a generic prefix with concatenated row and column numbers.

EXAMPLE
```
#include pv.sdf
struct PV p1;

p1 = pvCreate;

x = { 1 2,
      3 4 };

mask = { 1 0,
         0 1 };
```

```
p1 = pvPackm(p1,x,"P",mask);

print pvGetParNames(p1);

P[1,1]
P[2,2]
```

SOURCE pv.src

pvGetParVector

PURPOSE Retrieves parameter vector from structure of type **PV**.

INCLUDE pv.sdf

FORMAT *p* **= pvGetParVector(***p1***);**

INPUT *p1* an instance of structure of type **PV**.

OUTPUT *p* K×1 vector, parameter vector.

REMARKS Matrices or portions of matrices (stored using a mask) are stored in the structure of type **PV** as a vector in the *p* member.

EXAMPLE

```
#include pv.sdf
struct PV p1;

p1 = pvCreate;

x = { 1 2,
      3 4 };

mask = { 1 0,
```

```
                0 1 };

p1 = pvPackm(p1,x,"X",mask);

print pvUnpack(p1,1);



1.000  2.000
3.000  4.000

print pvGetParVector(p1);

1.000
4.000
```

SOURCE pv.src

pvLength

PURPOSE Returns the length of a parameter vector.

FORMAT *n* = **pvLength**(*p1*);

INPUT *p1* an instance of structure of type **PV**.

OUTPUT *n* scalar, length of parameter vector in *p1*.

SOURCE pv.src

pvList

PURPOSE Retrieves names of packed matrices in structure of type **PV**.

FORMAT *n* = **pvList**(*p1*);

INPUT *p1* an instance of structure of type **PV**.

OUTPUT *n* K×1 string vector, names of packed matrices.

SOURCE **pv.src**

pvPack

PURPOSE Packs general matrix into a structure of type **PV** with matrix name.

INCLUDE pv.sdf

FORMAT *p1* = **pvPack**(*p1*,*x*,*nm*);

INPUT *p1* an instance of structure of type **PV**.
x M×N matrix or N-dimensional array.
nm string, name of matrix/array.

OUTPUT *p1* an instance of structure of type **PV**.

EXAMPLE
```
#include pv.sdf

y = rndn(100,1);
x = rndn(100,5);
```

```
struct PV p1;
p1 = pvCreate;
p1 = pvPack(p1,x,"Y");
p1 = pvPack(p1,y,"X");
```

These matrices can be extracted using the **pvUnpack** command:

```
y = pvUnpack(p1,"Y");
x = pvUnpack(p1,"X");
```

SOURCE **pv.src**

SEE ALSO **pvPackm**, **pvPacks**, **pvUnpack**

pvPacki

PURPOSE Packs general matrix or array into a **PV** instance with name and index.

INCLUDE `pv.sdf`

FORMAT *p1* **= pvPacki(***p1*,*x*,*nm*,*i***);**

INPUT

p1	an instance of structure of type **PV**.
x	M×N matrix or N-dimensional array.
nm	string, name of matrix or array, or null string.
i	scalar, index of matrix or array in lookup table.

OUTPUT

p1	an instance of structure of type **PV**.

EXAMPLE

```
#include pv.sdf
```

```
y = rndn(100,1);
x = rndn(100,5);
struct PV p1;
p1 = pvCreate;
p1 = pvPacki(p1,y,"Y",1);
p1 = pvPacki(p1,x,"X",2);
```


These matrices can be extracted using the **pvUnpack** command:

```
y = pvUnpack(p1,1);
x = pvUnpack(p1,2);
```

SEE ALSO **pvPack**, **pvUnpack**

pvPackm

PURPOSE Packs general matrix into a structure of type **PV** with a mask and matrix name.

INCLUDE `pv.sdf`

FORMAT *p1* = **pvPackm**(*p1*,*x*,*nm*,*mask*);

INPUT

p1	an instance of structure of type **PV**.
x	M×N matrix or N-dimensional array.
nm	string, name of matrix/array or N-dimensional array.
mask	M×N matrix, mask matrix of zeros and ones.

OUTPUT

p1	an instance of structure of type **PV**.

REMARKS The *mask* argument allows storing a selected portion of a matrix into the packed vector. The ones in *mask* indicate an element to be stored in the packed matrix.

When the matrix is unpacked (using **pvUnpack**) the elements corresponding to the zeros are restored. Elements corresponding to the ones come from the packed vector which may have been changed.

If the mask is all zeros, the matrix or array is packed with the specified elements in the second argument but no elements of the matrix or array are entered into the parameter vector. When unpacked the matrix or array in the second argment is returned without modification.

EXAMPLE

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2,
      3 4 };

mask = { 1 0,
         0 1 };

p1 = pvPackm(p1,x,"X",mask);

print pvUnpack(p1,1);

    1.000  2.000
    3.000  4.000

p1 = pvPutParVector(p1,5|6);

print pvUnpack(p1,"X");

    5.000  2.000
    3.000  6.000
```

SOURCE pv.src

pvPackmi

PURPOSE Packs general matrix or array into a **PV** instance with a mask, name, and index.

INCLUDE `pv.sdf`

FORMAT *p1* = **pvPackmi**(*p1*,*x*,*nm*,*mask*,*i*);

INPUT

p1	an instance of structure of type **PV**.
x	M×N matrix or N-dimensional array.
nm	string, matrix or array name.
mask	M×N matrix or N-dimensional array, *mask* of zeros and ones.
i	scalar, index of matrix or array in lookup table.

OUTPUT

p1	an instance of structure of type **PV**.

REMARKS The *mask* allows storing a selected portion of a matrix into the parameter vector. The ones in the *mask* matrix indicate an element to be stored in the parameter matrix. When the matrix is unpacked (using **pvUnpackm**) the elements corresponding to the zeros are restored. Elements corresponding to the ones come from the parameter vector.

If the mask is all zeros, the matrix or array is packed with the specified elements in the second argument but no elements of the matrix or array are entered into the parameter vector. When unpacked the matrix or array in the second argment is returned without modification.

EXAMPLE

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2,
```

```
      3 4 };

mask = { 1 0,
         0 1 };

p1 = pvPackmi(p1,x,"X",mask,1);

print pvUnpack(p1,1);

    1.000  2.000
    3.000  4.000

p1 = pvPutParVector(p1,5|6);

print pvUnpack(p1,1);

    5.000  2.000
    3.000  6.000
```

SEE ALSO **pvPackm**, **pvUnpack**

pvPacks

PURPOSE Packs symmetric matrix into a structure of type **PV**.

INCLUDE `pv.sdf`

FORMAT *p1* **= pvPacks(***p1*,*x*,*nm***);**

INPUT

p1	an instance of structure of type **PV**.
x	M×M symmetric matrix.
nm	string, matrix name.

OUTPUT *p1* an instance of structure of type **PV**.

REMARKS **pvPacks** does not support the packing of arrays.

EXAMPLE
```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2,
      2 1 };

p1 = pvPacks(p1,x,"A");
p1 = pvPacks(p1,eye(2),"I");
```

These matrices can be extracted using the **pvUnpack** command:

```
print pvUnpack(p1,"A");

    1.000  2.000
    2.000  1.000

print pvUnpack(p1,"I");

    1.000  0.000
    0.000  1.000
```

SOURCE pv.src

SEE ALSO **pvPacksm**, **pvUnpack**

pvPacksi

pvPacksi

PURPOSE Packs symmetric matrix into a **PV** instance with matrix name and index.

INCLUDE `pv.sdf`

FORMAT *p1* **= pvPacksi(***p1*,*x*,*nm*,*i*);

INPUT

p1	an instance of structure of type **PV**.
x	M×M symmetric matrix.
nm	string, matrix name.
i	scalar, index of matrix in lookup table.

OUTPUT

p1	an instance of structure of type **PV**.

REMARKS **pvPacksi** does not support the packing of arrays.

EXAMPLE

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2, 2 1 };

p1 = pvPacksi(p1,x,"A",1);
p1 = pvPacksi(p1,eye(2),"I",2);
```

These matrices can be extracted using the **pvUnpack** command.

```
print pvUnpack(p1,1);

    1.000  2.000
    2.000  1.000

print pvUnpack(p1,2);

    1.000  0.000
    0.000  1.000
```

SEE ALSO **pvPacks**, **pvUnpack**

pvPacksm

PURPOSE Packs symmetric matrix into a structure of type **PV** with a mask.

INCLUDE `pv.sdf`

FORMAT *p1* = **pvPacksm**(*p1*,*x*,*nm*,*mask*);

INPUT

p1	an instance of structure of type **PV**.
x	M×M symmetric matrix.
nm	string, matrix name.
mask	M×M matrix, mask matrix of zeros and ones.

OUTPUT

p1	an instance of structure of type **PV**.

REMARKS **pvPacksm** does not support the packing of arrays.

The mask allows storing a selected portion of a matrix into the packed vector. The ones in *mask* indicate an element to be stored in the packed matrix. When the matrix is unpacked (using **pvUnpack**) the elements corresponding to the zeros are restored. Elements corresponding to the ones come from the packed vector which may have been changed.

Only the lower left portion of the *mask* matrix is used, and only the lower left portion of the *x* matrix is stored in the packed vector.

If the mask is all zeros, the matrix is packed with the specified elements in the second argument but no elements of the matrix are entered into the parameter vector. When unpacked the matrix in the second argment is returned without modification.

EXAMPLE

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2 4,
      2 3 5,
      4 5 6};

mask = { 1 0 1,
           0 1 0,
         1 0 1 };

p1 = pvPacksm(p1,x,"A",mask);

print pvUnpack(p1,"A");

    1.000  2.000  4.000
    2.000  3.000  5.000
    4.000  5.000  6.000

p2 = pvGetParVector(p1);

print p2;

    1.000
    3.000
4.000
    6.000

p3 = { 10, 11, 12, 13 };
p1 = pvPutParVector(p1,p3);

print pvUnpack(p1,"A");

   10.000   2.000  12.000
    2.000  11.000   5.000
```

```
    12.000   5.000  13.000
```

SOURCE pv.src

pvPacksmi

PURPOSE Packs symmetric matrix into a **PV** instance with a mask, matrix name, and index.

INCLUDE pv.sdf

FORMAT *p1* = **pvPacksmi**(*p1*,*x*,*nm*,*mask*,*i*);

INPUT

p1	an instance of structure of type **PV**.
x	M×M symmetric matrix.
nm	string, matrix name.
mask	M×M matrix, symmetric mask matrix of zeros and ones.
i	scalar, index of matrix in lookup table.

OUTPUT

p1	an instance of structure of type **PV**.

REMARKS **pvPacksmi** does not support the packing of arrays.

The *mask* allows storing a selected portion of a matrix into the parameter vector. The ones in the *mask* matrix indicate an element to be stored in the parameter vector. When the matrix is unpacked (using **pvUnpackm**) the elements corresponding to the zeros are restored. Elements corresponding to the ones come from the parameter vector.

Only the lower left portion of the *mask* matrix is used, and only the lower left portion of the *x* matrix is stored in the packed vector.

If the mask is all zeros, the matrix is packed with the specified elements in the second argument but no elements of the matrix are entered into the parameter

vector. When unpacked the matrix in the second argment is returned without modification.

EXAMPLE

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2 4,
      2 3 5,
      4 5 6};

mask = { 1 0 1,
         0 1 0,
         1 0 1 };

p1 = pvPacksmi(p1,x,"A",mask,1);

print pvUnpack(p1,1);

1.000  2.000  4.000
2.000  3.000  5.000
4.000  5.000  6.000

p2 = pvGetParVector(p1);

print p2;

1.000
3.000
4.000
6.000

p3 = { 10, 11, 12, 13 };
p1 = pvPutParVector(p1,p3);

print pvUnpack(p1,1);
```

```
10.000  2.000  12.000
 2.000  11.000  5.000
12.000  5.000  13.000
```

SEE ALSO **pvPacksm**, **pvUnpack**

pvPutParVector

PURPOSE Inserts parameter vector into structure of type **PV**.

INCLUDE `pv.sdf`

FORMAT *p1* = **pvPutParVector**(*p1*,*p*);

INPUT

p1	an instance of structure of type **PV**.
p	K×1 vector, parameter vector.

OUTPUT

p1	an instance of structure of type **PV**.

REMARKS Matrices or portions of matrices (stored using a *mask*) are stored in the structure of type **PV** as a vector in the *p* member.

EXAMPLE

```
#include pv.sdf

struct PV p1;
p1 = pvCreate;

x = { 1 2 4,
      2 3 5,
      4 5 6};

mask = { 1 0 1,
```

```
                0 1 0,
                1 0 1 };

// packed as square matrix
p1 = pvPackm(p1,x,"A",mask);

print pvUnpack(p1,"A");

    1.000  2.000  4.000
    2.000  3.000  5.000
    4.000  5.000  6.000

p3 = { 10, 11, 12, 13, 14 };
p1 = pvPutParVector(p1,p3);

print pvUnpack(p1,"A");

   10.000   2.000  11.000
    2.000  12.000   5.000
   13.000   5.000  14.000
```

SOURCE pv.src

pvTest

PURPOSE Tests an instance of structure of type **PV** to determine if it is a proper structure of type **PV**.

FORMAT *i* **= pvTest(***p1***);**

INPUT *p1* an instance of structure of type **PV**.

OUTPUT *i* scalar, if 0, *p1* is a proper structure of type **PV**, else if 1, an improper or unitialized structure of type **PV**.

SOURCE pv.src

pvUnpack

PURPOSE Unpacks matrices stored in a structure of type **PV**.

FORMAT *x* = **pvUnpack**(*p1*,*m*);

INPUT
p1 an instance of structure of type **PV**.
m string, name of matrix, or integer, index of matrix.

OUTPUT
x M×N general matrix or M×M symmetric matrix or N-dimensional array.

SOURCE pv.src

QNewton

PURPOSE Optimizes a function using the BFGS descent algorithm.

FORMAT { *x*,*f*,*g*,*ret* } = **QNewton**(&*fct*,*start*);

INPUT
&*fct* pointer to a procedure that computes the function to be minimized. This procedure must have one input argument, a vector of parameter values, and one output argument, the value of the function evaluated at the input vector of parameter values.
start K×1 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 K×1 vector of parameter values, and a single output argument, a K×1 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 K×1 vector, labels for parameters.

_qn_PrintResults scalar, if 1, results are printed.

OUTPUT

x K×1 vector, coefficients at the minimum of the function.

f scalar, value of function at minimum.

g K×1 vector, gradient at the minimum of the function.

ret scalar, return code.

- **0** normal convergence
- **1** forced termination
- **2** max iterations exceeded
- **3** function calculation failed
- **4** gradient calculation failed
- **5** step length calculation failed
- **6** function cannot be evaluated at initial parameter values

REMARKS

If you are running in terminal mode, **GAUSS** will not see any input until you press ENTER. Pressing C on the keyboard will terminate iterations, and pressing P will toggle iteration output.

To reset global variables for this function to their default values, call **QNewtonSet**.

EXAMPLE

This example computes maximum likelihood coefficients and standard errors for a Tobit model:

```
/*
**  qnewton.e  -  a Tobit model
*/

z = loadd("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;
print "coefficients  standard errors";
print b~sqrt(diag(invpd(h)));

output off;

/*
** log-likelihood proc
*/

proc lpr(b);
    local s,m,u;
    s = b[4];
    if s <= 1e-4;
        retp(error(0));
    endif;
    m = z[.,2:4]*b[1:3,.];
    u = z[.,1] ./= 0;
    retp(-sumc(u.*lnpdfn2(z[.,1]-m,s) +
                    (1-u).*(ln(cdfnc(m/sqrt(s))))));
```

```
endp;
```

produces:

```
Tobit Model
coefficients  standard errors

     0.010417884      0.080220019
     -0.20805753      0.094551107
    -0.099749592      0.080006676
      0.65223067      0.099827309
```

SOURCE qnewton.src

QNewtonmt

PURPOSE Minimize an arbitrary function.

INCLUDE qnewtonmt.sdf

FORMAT *out* = **QNewtonmt**(&*fct*,*par*,*data*,*c*);

INPUT

&*fct* pointer to a procedure that computes the function to be minimized. This procedure must have two input arguments, an instance of a **PV** structure containing the parameters, and a **DS** structure containing data, if any. And, one output argument, the value of the function evaluated at the input vector of parameter values.

par an instance of a **PV** structure. The *par* instance is passed to the user-provided procedure pointed to by &*fct*. *par* is constructed using the **pvPack** functions.

data an array of instances of a **DS** structure. This array is passed to the user-provided pointed by **&***fct* to be used in the objective function. **QNewtonmt** does not look at this structure. Each instance contains the the following members which can be set in whatever way that is convenient for computing the objective function:

data[i].dataMatrix	N×K matrix, data matrix.
data[i].dataArray	N×K×L. . . array, data array.
data[i].vnames	string array, variable names (optional).
data[i].dsname	string, data name (optional).
data[i].type	scalar, type of data (optional).

c an instance of a **QNewtonmtControl** structure. Normally an instance is initialized by calling **QNewtonmtControlCreate** and members of this instance can be set to other values by the user. For an instance named *c*, the members are:

c.CovType	scalar, if 1, ML covariance matrix, else if 2, QML covariance matrix is computed. Default is 0, no covariance matrix.
c.GradProc	scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. Default = ., i.e., no gradient procedure has been provided.
c.MaxIters	scalar, maximum number of iterations. Default = 1e+5.
c.MaxTries	scalar, maximum number of attemps in random search. Default = 100.
c.relGradTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisifed **QNewtonmt** exits the iterations.
c.randRadius	scalar, If zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = .001.
c.output	scalar, if nonzero, results are printed. Default = 0.

c.PrintIters — scalar, if nonzero, prints iteration information. Default = 0.

c.disableKey — scalar, if nonzero, keyboard input disabled

OUTPUT *out* — an instance of an **QNewtonmtOut** structure. For an instance named out, the members are:

out.par — instance of a **PV** structure containing the parameter estimates will be placed in the member matrix *out*.**par**.

out.fct — scalar, function evaluated at *x*.

out.retcode — scalar, return code:

0 normal convergence.
1 forced exit.
2 maximum number of iterations exceeded.
3 function calculation failed.
4 gradient calculation failed.
5 Hessian calculation failed.
6 line search failed.
7 error with constraints.
8 function complex.

out.moment — K×K matrix, covariance matrix of parameters, if *c*.**covType** > 0.

out.hessian — K×K matrix, matrix of second derivatives of objective function with respect to parameters.

REMARKS There is one required user-provided procedure, the one computing the objective function to be minimized, and another optional functions, the gradient of the objective function.

These functions have one input argument that is an instance of type struct **PV** and a second argument that is an instance of type struct **DS**. On input to the call to **QNewtonmt**, the first argument contains starting values for the parameters and the second argument any required data. The data are passed in a separate argument because the structure in the first argument will be copied as it is passed through procedure calls which would be very costly if it contained large

data matrices. Since **QNewtonmt** makes no changes to the second argument it will be passed by pointer thus saving time because its contents aren't copied.

The **PV** structures are set up using the **PV** pack procedures, **pvPack**, **pvPackm**, **pvPacks**, and **pvPacksm**. These procedures allow for setting up a parameter vector in a variety of ways.

For example, we might have the following objective function for fitting a nonlinear curve to data:


```
proc Micherlitz(struct PV par1, struct DS data1);
    local p0,e,s2,x,y;
    p0 = pvUnpack(par1,"parameters");
    y = data1.dataMatrix[.,1];
    x = data1.dataMatrix[.,2];
    e = y - p0[1] - p0[2]*exp(-p0[3] * x);
    retp(-lnpdfmvn(e,e'e/rows(e));
endp;
```

In this example the dependent and independent variables are passed to the procedure as the first and second columns of a data matrix stored in a single **DS** structure. Alternatively these two columns of data can be entered into a vector of **DS** structures one for each column of data:

If the objective function is the negative of a proper log-likelihood, and if *c*.**covType** is set to 1, the covariance matrix of the parameters is computed and returned in *out*.**moment**, and standard errors, t-statistics and probabilities are printed if *c*.**output** = 1.

If the objective function returns the negative of a vector of log-likelihoods, and if *c*.**covType** is set to 2, the quasi-maximum likelihood (QML) covariance matrix of the parameters is computed.

EXAMPLE The following is a complete example for estimating the parameters of the Micherlitz equation in data on the parameters and where an optional gradient procedure has been provided.

```
#include QNewtonmt.sdf

struct DS d0;
d0 = dsCreate;

y =   3.183|
      3.059|
      2.871|
      2.622|
      2.541|
      2.184|
      2.110|
      2.075|
      2.018|
      1.903|
      1.770|
      1.762|
      1.550;

x = seqa(1,1,13);
d0.dataMatrix = y~x;

struct QNewtonmtControl c0;
c0 = QNewtonmtControlCreate;
c0.output = 1;  /* print results */
c0.covType = 1; /* compute moment matrix */
                /* of parameters */

struct PV par1;
par1 = pvCreate;
par1 = pvPack(par1,1|1|0,"parameters");

struct QNewtonmt out1;
out1 = QNewtonmt(&Micherlitz,par1,d0,c0);
```

SOURCE qnewtonmt.src

SEE ALSO **QNewtonmtControlCreate**, **QNewtonmtOutCreate**

QNewtonmtControlCreate

PURPOSE Creates default **QNewtonmtControl** structure.

INCLUDE qnewtonmt.sdf

FORMAT *c* = **QNewtonmtControlCreate;**

OUTPUT *c* instance of **QNewtonmtControl** structure with members set to default values.

SOURCE qnewtonmt.src

SEE ALSO **QNewtonmt**

QNewtonmtOutCreate

PURPOSE Creates default **QNewtonmtOut** structure.

FORMAT *c* = **QNewtonmtOutCreate;**

OUTPUT *c* instance of **QNewtonmtOut** structure with members set to default values.

SOURCE qnewtonmt.src

SEE ALSO **QNewtonmt**

QNewtonSet

PURPOSE Resets global variables used by **QNewton** to default values.

FORMAT **QNewtonSet;**

SOURCE qnewton.src

QProg

PURPOSE Solves the quadratic programming problem.

FORMAT { *x*,*u1*,*u2*,*u3*,*u4*,*u5* } = **QProg(***start*,*q*,*r*,*a*,*b*,*c*,*d*,*bnds***);**

INPUT

start	K×1 vector, start values.
q	K×K matrix, symmetric model matrix.
r	K×1 vector, model constant vector.
a	M×K matrix, equality constraint coefficient matrix, or scalar 0, no equality constraints.
b	M×1 vector, equality constraint constant vector, or scalar 0, will be expanded to M×1 vector of zeros.
c	N×K matrix, inequality constraint coefficient matrix, or scalar 0, no inequality constraints.
d	N×1 vector, inequality constraint constant vector, or scalar 0, will be expanded to N×1 vector of zeros.
bnds	K×2 matrix, bounds on *x*, the first column contains the lower bounds on *x*, and the second column the upper bounds. If scalar 0, the bounds for all elements will default to ±1e200.

GLOBAL INPUT

_qprog_maxit scalar, maximum number of iterations. Default = 1000.

OUTPUT

x K×1 vector, coefficients at the minimum of the function.

u1 M×1 vector, Lagrangian coefficients of equality constraints.

u2 N×1 vector, Lagrangian coefficients of inequality constraints.

u3 K×1 vector, Lagrangian coefficients of lower bounds.

u4 K×1 vector, Lagrangian coefficients of upper bounds.

ret scalar, return code.

- **0** successful termination
- **1** max iterations exceeded
- **2** machine accuracy is insufficient to maintain decreasing function values
- **3** model matrices not conformable
- **<0** active constraints inconsistent

REMARKS **QProg** solves the standard quadratic programming problem:

$$\min \frac{1}{2} x'Qx - x'R$$

subject to constraints,

$$\begin{aligned} Ax &= B \\ Cx &\geq D \end{aligned}$$

and bounds,

$$x_{low} \leq x \leq x_{up}$$

SOURCE qprog.src

QProgmt

PURPOSE Solves the quadratic programming problem.

INCLUDE `qprogmt.sdf`

FORMAT *qOut* = **QProgmt**(*qIn*);

INPUT *qIn* instance of a **qprogMTIn** structure containing the following members:

qIn.start	K×1 vector, start values.
qIn.q	K×K matrix, symmetric model matrix.
qIn.r	K×1 vector, model constant vector.
qIn.a	M×K matrix, equality constraint coefficient matrix, or scalar 0, no equality constraints.
qIn.b	M×1 vector, equality constraint constant vector, or scalar 0, will be expanded to M×1 vector of zeros.
qIn.c	N×K matrix, inequality constraint coefficient matrix, or scalar 0, no inequality constraints.
qIn.d	N×1 vector, inequality constraint constant vector, or scalar 0, will be expanded to N×1 vector of zeros.
qIn.bounds	K×2 matrix, bounds on *qOut*.**x**, the first column contains the lower bounds on *qOut*.**x**, and the second column the upper bounds. If scalar 0, the bounds for all elements will default to ±1e200.
maxit	scalar, maximum number of iterations. Default = 1000.

OUTPUT *qOut* instance of a **qprogMTOut** structure containing the following members:

qOut.x — K×1 vector, coefficients at the minimum of the function.

qOut.lagrange — instance of a **qprogMTLagrange** structure containing the following members:

- *qOut*.lagrange.lineq M×1 vector, Lagrangian coefficients of equality constraints.
- *qOut*.lagrange.linineq N×1 vector, Lagrangian coefficients of inequality constraints.
- *qOut*.lagrange.bounds K×2 matrix, Lagrangian coefficients of bounds, the first column contains the lower bounds and the second the upper bounds.

qOut.ret — scalar, return code.

- **0** successful termination
- **1** max iterations exceeded
- **2** machine accuracy is insufficient to maintain decreasing function values
- **3** model matrices not conformable
- **<0** active constraints inconsistent

REMARKS **QProgmt** solves the standard quadratic programming problem:

$$\min \frac{1}{2} x' Q x - x' R$$

subject to constraints,

$$Ax = B$$

$$Cx \geq D$$

and bounds,

$$x_{low} \leq x \leq x_{up}$$

SOURCE qprogmt.src

SEE ALSO **QProgmtInCreate**

QProgmtInCreate

PURPOSE Creates an instance of a structure of type **QProgmtInCreate** with the **maxit** member set to a default value.

INCLUDE qprogmt.sdf

FORMAT *s* = **QProgmtInCreate;**

OUTPUT *s* instance of structure of type **QProgmtInCreate**.

SOURCE qprogmt.src

SEE ALSO **QProgmt**

qqr

PURPOSE Computes the orthogonal-triangular (QR) decomposition of a matrix X, such that:

$$X = Q_1 R$$

FORMAT { *q1*,*r* } = **qqr**(*x*);

INPUT
x N×P matrix.

OUTPUT
q1 N×K unitary matrix, K = min(N,P).
r K×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

$$Q = [\ Q_1\ Q_2\ ]$$

where Q_1 has P columns, then

$$X = Q_1 R$$

is the QR decomposition of X. If X has linearly independent columns, R is also the Cholesky factorization of the moment matrix of X, i.e., of $X'X$.

If you want only the R matrix, see the function **qr**. Not computing Q_1 can produce significant improvements in computing time and memory usage.

An unpivoted R matrix can also be generated using **cholup**:

```
r = cholup(zeros(cols(x),cols(x)),x);
```

For linear equation or least squares problems, which require Q_2 for computing residuals and residual sums of squares, see **olsqr** and **qtyr**.

For most problems an explicit copy of Q_1 or Q_2 is not required. Instead one of the following, $Q'Y$, QY, $Q_1'Y$, Q_1Y, $Q_2'Y$, or Q_2Y, for some Y, is required. These cases are all handled by **qtyr** and **qyr**. These functions are available because Q and Q_1 are typically very large matrices while their products with Y are more manageable.

If N<P, the factorization assumes the form:

$$Q'X = [\ R_1\ R_2\ ]$$

where R_1 is a P×P upper triangular matrix and R_2 is P×(N-P). Thus Q is a P×P matrix and R is a P×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* } = **qqre**(*x*);

INPUT

x	N×P matrix.

OUTPUT

q1	N×K unitary matrix, K = min(N,P).
r	K×P upper triangular matrix.
e	P×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

$$Q = [\ Q_1\ Q_2\ ]$$

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:

$$X[., E] = [\ X_1\ X_2\ ]$$

where X_1 is N×M and X_2 is N×(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×M and R_{12} is M×(P-M). Then

$$A = R_{11}^{-1} R_{12}$$

and

$$X_2 = X_1 A$$

that is, A is an M×(P-N) matrix defining the linear combinations of X_2 with respect to X_1.

If N<P, the factorization assumes the form:

$$Q'X = [\ R_1\ R_2\ ]$$

where R_1 is a P×P upper triangular matrix and R_2 is P×(N-P). Thus Q is a P×P matrix and R is a P×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

$$b = \texttt{qrsol}(Q'Y, R1)|\texttt{zeros(N-P,1);}$$

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* } = **qqrep**(*x*,*pvt*);

INPUT

x	N×P matrix.
pvt	P×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

q1	N×K unitary matrix, K = min(N,P).
r	K×P upper triangular matrix.
e	P×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

$$Q = [\ Q_1\ Q_2\ ]$$

where Q_1 has P columns, then

$$X[., E] = Q_1 R$$

is the QR decomposition of $X[., E]$.

qqrep allows you to control the pivoting. For example, suppose that X is a data set with a column of ones in the first column. If there are linear dependencies among the columns of X, the column of ones for the constant may get pivoted away. This column can be forced to be included among the linearly independent columns using *pvt*.

If you want only the R matrix, see **qrep**. Not computing Q_1 can produce significant improvements in computing time and memory usage.

SOURCE qqr.src

SEE ALSO **qqr**, **qre**, **olsqr**

qr

PURPOSE Computes the orthogonal-triangular (QR) decomposition of a matrix *X*, such that:

$$X = Q_1 R$$

FORMAT *r* **= qr(***x***);**

INPUT *x* N×P matrix.

OUTPUT *r* K×P 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

$$Q = [\ Q_1\ Q_2\ ]$$

where Q_1 has P columns, then

$$X = Q_1 R$$

is the QR decomposition of X. If X has linearly independent columns, R is also the Cholesky factorization of the moment matrix of X, i.e., of $X'X$.

qr does not return the Q_1 matrix because in most cases it is not required and can be very large. If you need the Q_1 matrix, see the function **qqr**. If you need the entire Q matrix, call **qyr** with Y set to a conformable identity matrix.

For most problems $Q'Y$, $Q_1'Y$, or QY, Q_1Y, for some Y, are required. For these cases see **qtyr** and **qyr**.

For linear equation or least squares problems, which require Q_2 for computing residuals and residual sums of squares, see **olsqr**.

If N<P, the factorization assumes the form:

$$Q'X = [\ R_1\ R_2\ ]$$

where R_1 is a P×P upper triangular matrix and R_2 is P×(N-P). Thus Q is a P×P matrix and R is a P×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**

qre

PURPOSE Computes the orthogonal-triangular (QR) decomposition of a matrix X, such that:

$$X[., E] = Q_1 R$$

FORMAT { *r*,*e* } = **qre**(*x*);

INPUT
x N×P matrix.

OUTPUT
r K×P upper triangular matrix, K = min(N,P).
e P×1 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

$$Q = [\ Q_1\ Q_2\ ]$$

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 $Q'Y$, $Q_1'Y$, or QY, Q_1Y, for some Y, are required. For these cases see **qtyre** and **qyre**.

If X has rank P, then the columns of X will not be permuted. If X has rank M<P, then the M linearly independent columns are permuted to the front of X by E. Partition the permuted X in the following way:

$$X[., E] = [\ X_1\ X_2\ ]$$

where X_1 is N×M and X_2 is N×(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×M and R_{12} is M×(P-M). Then

$$A = R_{11}^{-1} R_{12}$$

and

$$X_2 = X_1 A$$

that is, A is an M×(P-N) matrix defining the linear combinations of X_2 with respect to X_1.

If N<P the factorization assumes the form:

$$Q'X = [\ R_1\ R_2\ ]$$

where R_1 is a P×P upper triangular matrix and R_2 is P×(N-P). Thus Q is a P×P matrix and R is a P×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

$$b = \texttt{qrsol(}Q'Y\texttt{,}R1\texttt{)|zeros(N-P,1);}$$

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* } = **qrep**(*x*,*pvt*);

INPUT *x* N×P matrix.

pvt P×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

r K×P upper triangular matrix, K = min(N,P).

e P×1 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

$$Q = [\ Q_1\ Q_2\ ]$$

where Q_1 has P columns, then

$$X[., E] = Q_1 R$$

is the QR decomposition of $X[., E]$.

qrep does not return the Q_1 matrix because in most cases it is not required and

can be very large. If you need the Q_1 matrix, see the function **qqrep**. If you need the entire Q matrix, call **qyrep** with Y set to a conformable identity matrix. For most problems $Q'Y$, $Q_1'Y$, or QY, Q_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* **= qrsol(***b***,***R***);**

INPUT

b	P×L matrix.
R	P×P upper triangular matrix.

OUTPUT

x	P×L matrix.

REMARKS **qrsol** applies a backsolve to $Rx = b$ to solve for x. Generally R will be the R matrix from a QR factorization. **qrsol** may be used, however, in any situation where R is upper triangular.

SOURCE qrsol.src

SEE ALSO **qqr**, **qr**, **qtyr**, **qrtsol**

qrtsol

PURPOSE Computes the solution of $R'x = b$ where R is an upper triangular matrix.

FORMAT x **= qrtsol(**b**,**R**);**

INPUT
b	P×L matrix.
R	P×P upper triangular matrix.

OUTPUT
x	P×L matrix.

REMARKS **qrtsol** applies a forward solve to $R'x = b$ to solve for x. Generally R will be the R matrix from a QR factorization. **qrtsol** may be used, however, in any situation where R is upper triangular. If R is lower triangular, transpose before calling **qrtsol**.

If R is not transposed, use **qrsol**.

SOURCE qrsol.src

SEE ALSO **qqr**, **qr**, **qtyr**, **qrsol**

qtyr

PURPOSE Computes the orthogonal-triangular (QR) decomposition of a matrix X and returns $Q'Y$ and R.

FORMAT { qty**,**r } **= qtyr(**y**,**x**);**

INPUT
y	N×L matrix.

	x	N×P matrix.
OUTPUT	*qty*	N×L unitary matrix.
	r	K×P 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

$$Q = [\ Q_1\ Q_2\ ]$$

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 N×L 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:

$$H = Q_2'Y = qty[P + 1 : N, .]$$

Suppose that X is an N×K data set of independent variables, and Y is an N×1 vector of dependent variables. Then it can be shown that

$$b = R^{-1}G$$

and

$$s_j = \sum_{i=1}^{N-P} H_{i,j}, \ \ j = 1, 2, ...L$$

where b is a P×L matrix of least squares coefficients and s is a 1×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* } = **qtyre**(*y*,*x*);

INPUT

y	N×L matrix.
x	N×P matrix.

OUTPUT

qty	N×L unitary matrix.
r	K×P upper triangular matrix, K = min(N,P).
e	P×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

$$Q = [\ Q_1\ Q_2\ ]$$

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:

$$X[., E] = [\ X_1\ X_2\ ]$$

where X_1 is N×M and X_2 is N×(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×M and R_{12} is M×(P-M). Then

$$A = R_{11}^{-1} R_{12}$$

and

$$X_2 = X_1 A$$

that is, A is an M×(P-N) matrix defining the linear combinations of X_2 with respect to X_1.

For most problems Q or Q_1 is not it is required. Rather, we require $Q'Y$ or $Q_1'Y$ where Y is an N×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. $Q_1'Y$ 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×K data set of independent variables and Y is an N×1 vector of dependent variables. Suppose further that X contains linearly dependent columns, i.e., X has rank M<P. Then define

$$C = Q_1'Y[1 : M, .]$$

$$A = R[1 : M, 1 : M]$$

and the vector (or matrix of L>1) of least squares coefficients of the reduced, linearly independent problem is the solution of

$$Ab = C$$

To solve for b use **qrsol**:

```
b = qrsol(C,A);
```

If N<P, the factorization assumes the form:

$$Q'X[., E] = [\ R_1\ R_2\ ]$$

where R_1 is a P×P upper triangular matrix and R_2 is P×(N-P). Thus Q is a P×P matrix and R is a P×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

$$b = \texttt{qrsol}(Q'Y,R1)|\texttt{zeros(N-P,1);}$$

SOURCE qtyr.src

SEE ALSO **qqr**, **qre**, **qtyr**

qtyrep

PURPOSE Computes the orthogonal-triangular (QR) decomposition of a matrix X using a pivot vector and returns $Q'Y$ and R.

FORMAT { *qty*,*r*,*e* } = **qtyrep**(*y*,*x*,*pvt*);

INPUT

y	N×L matrix.
x	N×P matrix.
pvt	P×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×L unitary matrix.
r	K×P upper triangular matrix, K = min(N,P).
e	P×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

$$Q = [\ Q_1\ Q_2\ ]$$

where Q_1 has P columns, then

$$X[., E] = Q_1 R$$

is the QR decomposition of $X[., E]$.

qtyrep allows you to control the pivoting. For example, suppose that X is a data set with a column of ones in the first column. If there are linear dependencies among the columns of X, the column of ones for the constant may get pivoted away. This column can be forced to be included among the linearly independent columns using *pvt*.

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 =           0.0000000   −0.95346259    4.7673129
              0.0000000     0.0000000    3.1622777

             1.0000000
e =          2.0000000
             3.0000000
```

SOURCE qtyr.src

SEE ALSO **qrep**, **qtyre**

quantile

PURPOSE Computes quantiles from data in a matrix, given specified probabilities.

FORMAT *y* = **quantile**(*x*,*e*)

INPUT
x N×K matrix of data.
e L×1 vector, **quantile** levels or probabilities.

OUTPUT
y L×K 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

 1.9687       2.0899      1.8576      2.0545
```

SOURCE quantile.src

quantiled

PURPOSE Computes quantiles from data in a data set, given specified probabilities.

FORMAT *y* **= quantiled(***dataset***,***e***,***var***);**

INPUT *dataset* string, data set name, or N×M matrix of data.

	e	L×1 vector, quantile levels or probabilities.
	var	K×1 vector or scalar zero. If K×1, character vector of labels selected for analysis, or numeric vector of column numbers in data set of variables selected for analysis. If scalar zero, all columns are selected. If dataset is a matrix *var* cannot be a character vector.
OUTPUT	*y*	L×K matrix, quantiles.

REMARKS **quantiled** will not succeed if N*minc(e) is less than 1, or N*maxc(e) is greater than N - 1. In other words, to produce a **quantile** for a level of .001, the input matrix must have more than 1000 rows.

Example:

```
y = quantiled("tobit",e,0);

print "medians";
print y[2,.];
print;
print "95 percentiles";
print y[1,.];
print y[3,.];
```

produces:

```
medians

0.0000     1.0000    -0.0021    -0.1228

95 percentiles

-1.1198     1.0000    -1.8139    -2.3143

2.3066     1.0000     1.4590     1.6954
```

SOURCE quantile.src

qyr

PURPOSE Computes the orthogonal-triangular (QR) decomposition of a matrix *X* and returns *QY* and *R*.

FORMAT { *qy*,*r* } = **qyr**(*y*,*x*);

INPUT

y	N×L matrix.
x	N×P matrix.

OUTPUT

qy	N×L unitary matrix.
r	K×P 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

$$Q = [\ Q_1\ Q_2\ ]$$

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 N×L 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);
```

```
qy =  4.6288991   9.0506281
     -3.6692823  -7.8788202
      3.1795692   1.0051489

r =  -7.3484692  -4.6268140
      0.0000000   10.468648
```

SOURCE qyr.src

SEE ALSO **qqr**, **qyre**, **qyrep**, **olsqr**

qyre

PURPOSE Computes the orthogonal-triangular (QR) decomposition of a matrix X and returns QY and R.

FORMAT { *qy*,*r*,*e* } = **qyre**(*y*,*x*);

INPUT *y* N×L matrix.

	x	N×P matrix.
OUTPUT	*qy*	N×L unitary matrix.
	r	K×P upper triangular matrix, K = min(N,P).
	e	P×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

$$Q = [\ Q_1\ Q_2\ ]$$

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×L matrix, which will be a much smaller matrix.

If either $Q'Y$ or $Q_1'Y$ are required, see **qtyre**.

IfN <P, the factorization assumes the form:

$$Q'X[., E] = [\ R_1\ R_2\ ]$$

where R_1 is a P×P upper triangular matrix and R_2 is P×(N-P). Thus Q is a P×P matrix and R is a P×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);
```

```
qy =  -0.59422765   -3.0456088
      -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**

qyrep

PURPOSE Computes the orthogonal-triangular (QR) decomposition of a matrix X using a pivot vector and returns QY and R.

FORMAT { *qy*,*r*,*e* } **= qyrep(***y*,*x*,*pvt***);**

INPUT

y	N×L matrix.
x	N×P matrix.

pvt P×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

qy N×L unitary matrix.

r K×P upper triangular matrix, K = min(N,P).

e P×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

$$Q = [\ Q_1\ Q_2\ ]$$

where Q_1 has P columns, then

$$X[., E] = Q_1 R$$

is the QR decomposition of $X[., E]$.

qyrep allows you to control the pivoting. For example, suppose that X is a data set with a column of ones in the first column. If there are linear dependencies among the columns of X, the column of ones for the constant may get pivoted

away. This column can be forced to be included among the linearly independent columns using *pvt*.

For most problems Q or Q_1 is not what is required. Since Q can be a very large matrix, **qyrep** has been provided for the calculation of QY, where Y is some N×L matrix, which will be a much smaller matrix.

If either $Q'Y$ or $Q_1'Y$ are required, see **qtyrep**.

If N<P, the factorization assumes the form:

$$Q'X[.,E] = [\ R_1\ R_2\ ]$$

where R_1 is a P×P upper triangular matrix and R_2 is P×(N-P). Thus Q is a P×P matrix and R is a P×N matrix containing R_1 and R_2.

SOURCE qyr.src

SEE ALSO **qr**, **qqrep**, **qrep**, **qtyrep**

rank

PURPOSE Computes the rank of a matrix, using the singular value decomposition.

FORMAT *k* **= rank(***x***);**

INPUT *x* N×P matrix.

GLOBAL INPUT

_svdtol scalar, the tolerance used in determining if any of the singular values are effectively 0. The default value is $10e^{-13}$. This can be changed before calling the procedure.

OUTPUT *k* an estimate of the rank of *x*. This equals the number of singular values of *x* that exceed a prespecified tolerance in absolute value.

GLOBAL OUTPUT **_svderr** scalar, if not all of the singular values can be computed **_svderr** will be nonzero.

SOURCE svd.src

rankindx

PURPOSE Returns the vector of ranks of a vector.

FORMAT *y* = **rankindx**(*x*,*flag*);

INPUT *x* N×1 vector.

flag scalar, 1 for numeric data or 0 for character data.

OUTPUT *y* N×1 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 = 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**(*f1*,*r*);

INPUT
f1 scalar, file handle of an open file.
r scalar, number of rows to read.

OUTPUT
y N×K matrix, the data read from the file.

REMARKS The first time a **readr** statement is encountered, the first *r* rows will be read. The next time it is encountered, the next *r* rows will be read in, and so on. If the end of the data set is reached before *r* rows can be read, then only those rows remaining will be read.

After the last row has been read, the pointer is placed immediately after the end of the file. An attempt to read the file in these circumstances will cause an error message.

To move the pointer to a specific place in the file use **seekr**.

EXAMPLE
```
open dt = dat1.dat;
m = 0;

do until eof(dt);
    x = readr(dt,400);
    m = m+moment(x,0);
endo;

dt = close(dt);
```

This code reads data from a data set 400 rows at a time. The moment matrix for each set of rows is computed and added to the sum of the previous moment

matrices. The result is the moment matrix for the entire data set. **eof(dt)** returns 1 when the end of the data set is encountered.

SEE ALSO **open**, **create**, **writer**, **seekr**, **eof**

real

PURPOSE Returns the real part of *x*.

FORMAT *zr* **= real(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *zr* N×K matrix or N-dimensional array, the real part of *x*.

REMARKS If *x* is not complex, *zr* will be equal to *x*.

EXAMPLE
```
x = { 1 11,
      7i 3,
      2+i 1 };

zr = real(x);
```

```
          1.0000000   11.000000
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

x	N×1 vector to be recoded (changed).
e	N×K matrix of 1's and 0's.
v	K×1 vector containing the new values to be assigned to the recoded variable.

OUTPUT

y	N×1 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;

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
      2
v =   3
      4
```

```
      20
       3
y =    2
      63
       1
```

SOURCE datatran.src

SEE ALSO **code**, **substute**

recode (dataloop)

PURPOSE Changes the value of a variable with different values based on a set of logical expressions.

FORMAT
```
recode [#] [$] var with
        val_1 for expression_1,
        val_2 for expression_2,
            .
            .
            .
        val_n for expression_n;
```

INPUT

var literal, the new variable name.

val scalar, value to be used if corresponding expression is TRUE.

expression logical scalar-returning expression that returns nonzero TRUE or zero FALSE.

REMARKS If '**$**' is specified, the variable will be considered a character variable. If '**#**' is specified, the variable will be considered numeric. If neither is specified, the type of the variable will be left unchanged.

The logical expressions must be mutually exclusive, that is only one may return TRUE for a given row (observation).

If none of the expressions is TRUE for a given row (observation), its value will remain unchanged.

Any variables referenced must already exist, either as elements of the source data set, as **extern**'s, or as the result of a previous **make**, **vector**, or **code** statement.

EXAMPLE
```
recode age with
    1 for age < 21,
```

```
    2 for age >= 21 and age < 35,
    3 for age >= 35 and age < 50,
    4 for age >= 50 and age < 65,
    5 for age >= 65;


recode $ sex with
    "MALE" for sex =\,= 1,
    "FEMALE" for sex =\,= 0;


recode # sex with
    1 for sex $=\,= "MALE",
    0 for sex $=\,= "FEMALE";
```

SEE ALSO **code (dataloop)**

recserar

PURPOSE Computes a vector of autoregressive recursive series.

FORMAT *y* **= recserar(***x*,*y0*,*a***);**

INPUT

x	N×K matrix
y0	P×K matrix.
a	P×K matrix.

OUTPUT

y	N×K matrix containing the series.

REMARKS **recserar** is particularly useful in dealing with time series.

Typically, the result would be thought of as K vectors of length N.

y0 contains the first P values of each of these vectors (thus, these are prespecified). The remaining elements are constructed by computing a 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:

$$y[t,.] = x[t,.] + a[1,.] * y[t-1,.] + \ldots + a[P,.] * y[t-P,.], t = P+1, \ldots, N$$

and

$$y[t,.] = y0[t,.], t = 1, \ldots, P$$

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;
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] = rho[1,1]*y[1,t-1] + x[t,.]*b + e[1,t]
y[2,t] = 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**

recsercp

PURPOSE Computes a recursive series involving products. Can be used to compute cumulative products, to evaluate polynomials using Horner's rule, and to convert from base *b* representations of numbers to decimal representations among other things.

FORMAT *y* **= recsercp(***x*,*z***);**

INPUT
x N×K or 1×K matrix
z N×K or 1×K matrix.

OUTPUT
y N×K matrix in which each column is a series generated by a recursion of the form:

$$\begin{aligned} y(1) &= x(1) + z(1) \\ y(t) &= y(t-1) * x(t) + z(t), t = 2, ...N \end{aligned}$$

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(*x***,0)** 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 **x** is a scalar and **c** is an (N+1)×1 vector, the result **p** will contain the value of the polynomial whose coefficients are given in **c**. That is:

```
p = c[1,.].*x^n + c[2,.].*x^(n-1) + ... + c[n+1,.];
```

Note that both **x** and **c** could contain more than 1 column, and then this code would evaluate the entire set of polynomials at the same time. Note also that if **x** = 2, and if **c** contains the digits of the binary representation of a number, then **p** will be the decimal representation of that number.

SEE ALSO **recserar**, **recserrc**

recserrc

PURPOSE Computes a recursive series involving division.

FORMAT *y* **= recserrc(***x***,***z***);**

INPUT

x	1×K or K×1 vector.
z	N×K matrix.

OUTPUT *y* N×K matrix in which each column is a series generated by a recursion of the form:

$$
\begin{aligned}
y[1] &= x \bmod z[1], \quad x = \text{trunc}(x/z[1]) \\
y[2] &= x \bmod z[2], \quad x = \text{trunc}(x/z[2]) \\
y[3] &= x \bmod z[3], \quad x = \text{trunc}(x/z[3]) \\
&\vdots \\
y[n] &= x \bmod z[n]
\end{aligned}
$$

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**

rerun

PURPOSE Displays the most recently created graphics file.

LIBRARY pgraph

FORMAT **rerun;**

REMARKS **rerun** is used by the **endwind** function.

SOURCE pcart.src

GLOBALS **_pcmdlin**, **_pnotify**, **_psilent**, **_ptek**, **_pzoom**

reshape

PURPOSE Reshapes a matrix.

FORMAT *y* = **reshape**(*x*,*r*,*c*);

INPUT

x	N×K matrix.
r	scalar, new row dimension.
c	scalar, new column dimension.

OUTPUT

y	*r*×*c* matrix created from the elements of *x*.

REMARKS Matrices are stored in row major order.

The first *c* elements are put into the first row of *y*, the second in the second row, and so on. If there are more elements in *x* than in *y*, the remaining elements are discarded. If there are not enough elements in *x* to fill *y*, then when **reshape** runs out of elements, it goes back to the first element of *x* and starts getting additional elements from there.

EXAMPLE

```
y = reshape(x,2,6);
```

$$\text{If } x = \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix} \text{ then } y = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{matrix}$$

$$\text{If } x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \text{ then } y = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 & 2 & 3 \end{matrix}$$

$$\text{If } x = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{matrix} \text{ then } y = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{matrix}$$

$$\text{If } x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix} \text{ then } y = \begin{matrix} 1 & 2 & 3 & 4 & 1 & 2 \\ 3 & 4 & 1 & 2 & 3 & 4 \end{matrix}$$

$$\text{If } x = 1 \text{ then } y = \begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$$

SEE ALSO **submat**, **vec**

retp

PURPOSE Returns from a procedure or keyword.

FORMAT **retp;**

retp(*x*,*y*,...**);**

REMARKS For more details, see PROCEDURES AND KEYWORDS, Chapter 12.

In a **retp** statement 0-1023 items may be returned. The items may be expressions. Items are separated by commas.

It is legal to return with no arguments, as long as the procedure is defined to return 0 arguments.

SEE ALSO **proc**, **keyword**, **endp**

return

PURPOSE Returns from a subroutine.

FORMAT **return;**

return(*x*,*y*,...**);**

REMARKS The number of items that may be returned from a subroutine in a **return** statement is limited only by stack space. The items may be expressions. Items are separated by commas.

It is legal to return with no arguments and therefore return nothing.

SEE ALSO **gosub**, **pop**

rev

PURPOSE Reverses the order of the rows in a matrix.

FORMAT *y* **= rev(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* N×K 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 = round(rndn(5,3)*10);
y = 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**

rfft

PURPOSE Computes a real 1- or 2-D Fast Fourier transform.

FORMAT *y* **= rfft(***x***);**

INPUT *x* N×K real matrix.

OUTPUT *y* L×M 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 = rfft(x);
```

```
y =  6.0000000   1.0000000
     1.5000000  −2.5000000
```

SEE ALSO **rffti**, **fft**, **ffti**, **fftm**, **fftmi**

rffti

PURPOSE Computes inverse real 1- or 2-D Fast Fourier transform.

FORMAT *y* **= rffti(***x***);**

INPUT *x* N×K matrix.

OUTPUT *y* L×M real matrix, where L and M are the smallest prime factor products greater than or equal to N and K.

REMARKS It is up to the user to guarantee that the input will return a real result. If in doubt, use **ffti**.

EXAMPLE
```
x = { 6 1, 1.5 -2.5 };
```

```
y = rffti(x);
```

```
y =  6.0000000  9.0000000
     8.0000000  1.0000000
```

SEE ALSO **rfft**, **fft**, **ffti**, **fftm**, **fftmi**

rfftip

PURPOSE Computes an inverse real 1- or 2-D FFT. Takes a packed format FFT as input.

FORMAT *y* **= rfftip(***x***);**

INPUT *x* N×K matrix or K-length vector.

OUTPUT *y* L×M real matrix or M-length vector.

REMARKS **rfftip** assumes that its input is of the same form as that output by **rfftp** and **rfftnp**.

rfftip uses the Temperton prime factor FFT algorithm. This algorithm can compute the inverse FFT of any vector or matrix whose dimensions can be expressed as the product of selected prime number factors. **GAUSS** implements the Temperton algorithm for any integer power of 2, 3, and 5, and one factor of 7. Thus, **rfftip** can handle any matrix whose dimensions can be expressed as:

$$2^p \times 3^q \times 5^r \times 7^s, \quad \begin{array}{l} p,q,r \geq 0 \\ s=0 \text{ or } 1 \end{array}$$

If a dimension of *x* does not meet this requirement, it will be padded with zeros to the next allowable size before the inverse FFT is computed. Note that

rfftip assumes the length (for vectors) or column dimension (for matrices) of *x* is K-1 rather than K, since the last element or column does not hold FFT information, but the Nyquist frequencies.

The sizes of *x* and *y* are related as follows: L will be the smallest prime factor product greater than or equal to N, and M will be twice the smallest prime factor product greater than or equal to K-1. This takes into account the fact that *x* contains both positive and negative frequencies in the row dimension (matrices only), but only positive frequencies, and those only in the first K-1 elements or columns, in the length or column dimension.

It is up to the user to guarantee that the input will return a real result. If in doubt, use **ffti**. Note, however, that **ffti** expects a full FFT, including negative frequency information, for input.

Do not pass **rfftip** the output from **rfft** or **rfftn**–it will return incorrect results. Use **rffti** with those routines.

SEE ALSO **fft**, **ffti**, **fftm**, **fftmi**, **fftn**, **rfft**, **rffti**, **rfftn**, **rfftnp**, **rfftp**

rfftn

PURPOSE Computes a real 1- or 2-D FFT.

FORMAT *y* **= rfftn(***x***);**

INPUT *x* N×K real matrix.

OUTPUT *y* L×M 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:

$$2^p \times 3^q \times 5^r \times 7^s, \quad \begin{array}{ll} p,q,r \geq 0 & \text{for rows of matrix} \\ p>0,\ q,r \geq 0 & \text{for columns of matrix} \\ p>0,\ q,r \geq 0 & \text{for length of vector} \\ s=0 \text{ or } 1 & \text{for all dimensions} \end{array}$$

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 33600×1 vector can compute as much as 20 percent faster than a 32768×1 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. (You can use these to see the dimensions to which **rfftn** would pad a matrix or vector.)

rfftn scales the computed FFT by 1/(L*M).

SEE ALSO **fft**, **ffti**, **fftm**, **fftmi**, **fftn**, **rfft**, **rffti**, **rfftip**, **rfftnp**, **rfftp**

rfftnp

PURPOSE Computes a real 1- or 2-D FFT. Returns the results in a packed format.

FORMAT *y* = **rfftnp**(*x*);

INPUT *x* N×K real matrix or K-length real vector.

OUTPUT *y* L×(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:

$$2^p \times 3^q \times 5^r \times 7^s, \quad \begin{array}{ll} p,q,r \geq 0 & \text{for rows of matrix} \\ p>0,\ q,r \geq 0 & \text{for columns of matrix} \\ p>0,\ q,r \geq 0 & \text{for length of vector} \\ s=0 \text{ or } 1 & \text{for all dimensions} \end{array}$$

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 a single factor. For example, even though it is bigger, a 33600×1 vector can compute as much as 20 percent faster than a 32768×1 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**, **fftmi**, **fftn**, **rfft**, **rffti**, **rfftip**, **rfftn**, **rfftp**

rfftp

PURPOSE Computes a real 1- or 2-D FFT. Returns the results in a packed format.

FORMAT *y* = **rfftp**(*x*);

INPUT *x* N×K real matrix or K-length real vector.

OUTPUT *y* L×(M/2+1) matrix or (M/2+1)-length vector, where L and M are the smallest powers of 2 greater than or equal to N and K, respectively.

REMARKS If a dimension of *x* is not a power of 2, it will be padded with zeros to the next allowable size before the FFT is computed.

For 1-D FFT's, **rfftp** returns the positive frequencies in ascending order in the first M/2 elements, and the Nyquist frequency in the last element. For 2-D

FFT's, **rfftp** returns the positive and negative frequencies for the row dimension, and for the column dimension, it returns the positive frequencies in ascending order in the first M/2 columns, and the Nyquist frequencies in the last column. Usually the FFT of a real function is calculated to find the power density spectrum or to perform filtering on the waveform. In both these cases only the positive frequencies are required. (See also **rfft** and **rfftn** for routines that return the negative frequencies as well.)

rfftp scales the computed FFT by 1/(L*M).

rfftp uses the Temperton FFT algorithm.

SEE ALSO **fft**, **ffti**, **fftm**, **fftmi**, **fftn**, **rfft**, **rffti**, **rfftip**, **rfftn**, **rfftnp**

rndbeta

PURPOSE Computes pseudo-random numbers with beta distribution.

FORMAT *x* = **rndbeta**(*r*,*c*,*a*,*b*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
a	M×N matrix, E×E conformable with *r*×*c* resulting matrix, shape parameters for beta distribution.
b	K×L matrix, E×E conformable with *r*×*c* resulting matrix, shape parameters for beta distribution.

OUTPUT

x	*r*×*c* matrix, beta distributed pseudo-random numbers.

REMARKS The properties of the pseudo-random numbers in *x* are:

$$
\begin{aligned}
E(x) &= a/(a+b) \\
Var(x) &= a*b/((a+b+1)*(a+b)^2) \\
x &> 0 \\
x &< 1 \\
a &> 0 \\
b &> 0
\end{aligned}
$$

SOURCE **random.src**

rndcon, rndmult, rndseed

PURPOSE Resets the parameters of the linear congruential random number generator that is the basis for **rndu**, **rndi** and **rndn**.

FORMAT **rndcon** *c*;

rndmult *a*;

rndseed *seed*;

INPUT

c	scalar, constant for the random number generator.
a	scalar, multiplier for the random number generator.
seed	scalar, initial seed for the random number generator.

Parameter default values and ranges:

seed	time(0)	$0 < seed < 2^{32}$
a	1664525	$0 < a < 2^{32}$
c	1013904223	$0 <= c < 2^{32}$

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:

$$new\_seed = (((a * seed) \ \% \ 2^{32}) + c) \ \% \ 2^{32}$$

(where **%** is the mod operator). Then a number between 0 and 1 is created by dividing the new seed by 2^{32}:

$$x = new\_seed \ / \ 2^{32}$$

rndcon resets *c*.

rndmult resets *a*.

rndseed resets *seed*. This is the initial seed for the generator. The default is that **GAUSS** uses the clock to generate an initial seed when **GAUSS** is invoked.

GAUSS goes to the clock to seed the generator only when it is first started up. Therefore, if **GAUSS** is allowed to run for a long time, and if large numbers of random numbers are generated, there is a possibility of recycling (that is, the sequence of "random numbers" will repeat itself). However, the generator used has an extremely long cycle, so that should not usually be a problem.

The parameters set by these commands remain in effect until new commands are encountered, or until **GAUSS** is restarted.

SEE ALSO **rndu**, **rndn**, **rndi**, **rndLCi**, **rndKMi**

rndgam

PURPOSE Computes pseudo-random numbers with gamma distribution.

FORMAT *x* = **rndgam(***r*,*c*,*alpha***);**

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
alpha	M×N matrix, E×E conformable with *r*×*c* resulting matrix, shape parameters for gamma distribution.

OUTPUT

x	*r*×*c* 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 * 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`

rndi

PURPOSE Returns a matrix of random integers, $0 <= y < 2^{32}$.

FORMAT *y* **= rndi(***r***,***c***);**

INPUT
r scalar, row dimension.
c scalar, column dimension.

OUTPUT
y *r*×*c* matrix of random integers between 0 and 2^{32}-1, inclusive.

REMARKS *r* and *c* will be truncated to integers if necessary.

This generator is automatically seeded using the system clock when **GAUSS** first starts. However, that can be overridden using the **rndseed** statement.

Each seed is generated from the preceding seed, using the formula

new_seed = (((*a* * *seed*) % 2^{32})+*c*) % 2^{32}

where **%** is the mod operator. The new seeds are the values returned. The multiplicative constant and the additive constant may be changed using **rndmult** and **rndcon** respectively.

SEE ALSO **rndu**, **rndn**, **rndcon**, **rndmult**

rndKMbeta

PURPOSE Computes beta pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndKMbeta**(*r*,*c*,*a*,*b*,*state*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
a	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, first shape argument for beta distribution.
b	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, second shape argument for beta distribution.
state	scalar or 500×1 vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **500×1 vector case:** *state* = the state vector returned from a previous call to one of the **rndKM** random number functions.

OUTPUT

x	*r*×*c* matrix, beta distributed random numbers.
newstate	500×1 vector, the updated state.

REMARKS The properties of the pseudo-random numbers in *x* are:

$$E(x) = \frac{a}{a+b}, Var(x) = \frac{(a*b)}{(a+b+1)*(a+b)^2}$$

$$0 < x < 1, a > 0, b > 0$$

r and *c* will be truncated to integers if necessary.

SOURCE `randkm.src`

TECHNICAL NOTES **rndKMbeta** uses the recur-with-carry KISS+Monster algorithm described in the **rndKMi** Technical Notes.

rndKMgam

PURPOSE Computes Gamma pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndKMgam**(*r*,*c*,*alpha*,*state*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
alpha	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, shape argument for gamma distribution.
state	scalar or 500×1 vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **500×1 vector case:** *state* = the state vector returned from a previous call to one of the **rndKM** random number functions.

OUTPUT

x $r \times c$ matrix, gamma distributed random numbers.

newstate 500×1 vector, the updated state.

REMARKS

The properties of the pseudo-random numbers in *x* are:

$$E(x) = alpha, Var(x) = alpha$$

$$x > 0, alpha > 0$$

To generate **gamma(***alpha***,***theta***)** pseudo-random numbers where *theta* is a scale parameter, multiply the result of **rndKMgam** by *theta*.

Thus

```
z = theta * rndgam(1,1,alpha);
```

has the properties

$$E(z) = alpha * theta, Var(z) = alpha * theta^2$$

$$z > 0, alpha > 0, theta > 0$$

r and *c* will be truncated to integers if necessary.

SOURCE

randkm.src

TECHNICAL NOTES

rndKMgam uses the recur-with-carry KISS+Monster algorithm described in the **rndKMi** Technical Notes.

rndKMi

PURPOSE Returns a matrix of random integers, $0 <= y < 2^{32}$, and the state of the random number generator.

FORMAT { *y*,*newstate* } = **rndKMi**(*r*,*c*,*state*);

INPUT

r	scalar, row dimension.
c	scalar, column dimension.
state	scalar or 500×1 vector. **Scalar case:** *state* = starting seed value. If -1, **GAUSS** computes the starting seed based on the system clock. **500×1 vector case:** *state* = the state vector returned from a previous call to one of the **rndKM** random number generators.

OUTPUT

y	*r*×*c* matrix of random integers between 0 and 2^{32} - 1, inclusive.
newstate	500×1 vector, the updated state.

REMARKS *r* and *c* will be truncated to integers if necessary.

EXAMPLE This example generates two thousand vectors of random integers, each with one million elements. The state of the random number generator after each iteration is used as an input to the next generation of random numbers.

```
state = 13;
n = 2000;
k = 1000000;
c = 0;
min = 2^32+1;
max = -1;
```

```
do while c < n;
    { y,state } = rndKMi(k,1,state);
    min = minc(min | minc(y));
    max = maxc(max | maxc(y));
    c = c + k;
endo;

print "min " min;
print "max " max;
```


SEE ALSO **rndKMn**, **rndKMu**

TECHNICAL NOTES **rndKMi** generates random integers using a KISS+Monster algorithm developed by George Marsaglia. KISS initializes the sequence used in the recur-with-carry Monster random number generator. For more information on this generator see http://www.Aptech.com/random.

rndKMn

PURPOSE Returns a matrix of standard normal (pseudo) random variables and the state of the random number generator.

FORMAT { *y*,*newstate* } = **rndKMn**(*r*,*c*,*state*);

INPUT

r scalar, row dimension.

c scalar, column dimension.

state scalar or 500×1 vector.

Scalar case:

state = starting seed value. If -1, **GAUSS** computes the starting seed based on the system clock.

500×1 vector case:

state = the state vector returned from a previous call to one of the **rndKM** random number generators.

OUTPUT

y *r*×*c* matrix of standard normal random numbers.

newstate 500×1 vector, the updated state.

REMARKS

r and *c* will be truncated to integers if necessary.

EXAMPLE

This example generates two thousand vectors of standard normal random numbers, each with one million elements. The state of the random number generator after each iteration is used as an input to the next generation of random numbers.

```
state = 13;
n = 2000;
k = 1000000;
c = 0;
submean = {};

do while c < n;
    { y,state } = rndKMn(k,1,state);
    submean = submean | meanc(y);
    c = c + k;
endo;

mean = meanc(submean);
print mean;
```

SEE ALSO

rndKMu, **rndKMi**

TECHNICAL NOTES

rndKMn calls the uniform random number generator that is the basis for **rndKMu** multiple times for each normal random number generated. This is the recur-with-carry KISS+Monster algorithm described in the **rndKMi** Technical Notes. Potential normal random numbers are filtered using the fast acceptance-rejection algorithm proposed by Kinderman, A.J. and J.G. Ramage,

"Computer Generation of Normal Random Numbers," *Journal of the American Statistical Association*, December 1976, Volume 71, Number 356, pp. 893-896.

rndKMnb

PURPOSE Computes negative binomial pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndKMnb**(*r*,*c*,*k*,*p*,*state*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
k	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, "event" argument for negative binomial distribution.
p	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, "probability" argument for negative binomial distribution.
state	scalar or 500×1 vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **500×1 vector case:** *state* = the state vector returned from a previous call to one of the **rndKM** random number functions.

OUTPUT

x	*r*×*c* matrix, negative binomial distributed random numbers.
newstate	500×1 vector, the updated state.

REMARKS The properties of the pseudo-random numbers in *x* are:

$$E(x) = \frac{k * p}{(1-p)}, Var(x) = \frac{k * p}{(1-p)^2}$$

$$x = 0, 1, \ldots, k > 0, 0 < p < 1$$

r and *c* will be truncated to integers if necessary.

SOURCE `randkm.src`

TECHNICAL NOTES **rndKMnb** uses the recur-with-carry KISS+Monster algorithm described in the **rndKMi** Technical Notes.

rndKMp

PURPOSE Computes Poisson pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndKMp**(*r*,*c*,*lambda*,*state*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
lambda	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, shape argument for Poisson distribution.
state	scalar or 500×1 vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **500×1 vector case:** *state* = the state vector returned from a previous call to one of the **rndKM** random number functions.

OUTPUT

x	*r*×*c* matrix, Poisson distributed random numbers.
newstate	500×1 vector, the updated state.

REMARKS The properties of the pseudo-random numbers in *x* are:

$$E(x) = lambda, Var(x) = lambda$$

$$x = 0, 1, \ldots, lambda > 0$$

r and *c* will be truncated to integers if necessary.

SOURCE `randkm.src`

TECHNICAL NOTES **rndKMp** uses the recur-with-carry KISS+Monster algorithm described in the **rndKMi** Technical Notes.

rndKMu

PURPOSE Returns a matrix of uniform (pseudo) random variables and the state of the random number generator.

FORMAT { *y*,*newstate* } = **rndKMu**(*r*,*c*,*state*);

INPUT

r scalar, row dimension.

c scalar, column dimension.

state scalar, 2×1 vector, or 500×1 vector.

Scalar case:

state = starting seed value. If -1, **GAUSS** computes the starting seed based on the system clock.

2×1 vector case:

[1] the starting seed, uses the system clock if -1

[2] 0 for 0 <= *y* < 1
1 for 0 <= *y* <= 1

500×1 vector case:

state = the state vector returned from a previous call to one of the **rndKM** random number generators.

OUTPUT

y — *r*×*c* matrix of uniform random numbers, 0 <= *y* < 1.

newstate — 500×1 vector, the updated state.

REMARKS

r and *c* will be truncated to integers if necessary.

EXAMPLE

This example generates two thousand vectors of uniform random numbers, each with one million elements. The state of the random number generator after each iteration is used as an input to the next generation of random numbers.

```
state = 13;
n = 2000;
k = 1000000;
c = 0;
submean = {};

do while c < n;
    { y,state } = rndKMu(k,1,state);
    submean = submean | meanc(y);
    c = c + k;
endo;

mean = meanc(submean);
print 0.5-mean;
```

SEE ALSO

rndKMn, **rndKMi**

TECHNICAL NOTES

rndKMu uses the recur-with-carry KISS-Monster algorithm described in the **rndKMi** Technical Notes. Random integer seeds from 0 to 2^{32}-1 are generated. Each integer is divided by 2^{32} or 2^{32}-1.

rndKMvm

PURPOSE Computes von Mises pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndKMvm**(*r*,*c*,*m*,*k*,*state*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
m	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, means for vm distribution.
k	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, shape argument for vm distribution.
state	scalar or 500×1 vector. **Scalar case:** *state* = starting seed value only. If -1, **GAUSS** computes the starting seed based on the system clock. **500×1 vector case:** *state* = the state vector returned from a previous call to one of the **rndKM** random number functions.

OUTPUT

x	*r*×*c* matrix, von Mises distributed random numbers.
newstate	500×1 vector, the updated state.

REMARKS *r* and *c* will be truncated to integers if necessary.

SOURCE randkm.src

TECHNICAL NOTES **rndKMvm** uses the recur-with-carry KISS+Monster algorithm described in the **rndKMi** Technical Notes.

rndLCbeta

PURPOSE Computes beta pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndLCbeta**(*r*,*c*,*a*,*b*,*state*);

INPUT

r scalar, number of rows of resulting matrix.

c scalar, number of columns of resulting matrix.

a *r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, first shape argument for beta distribution.

b *r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, second shape argument for beta distribution.

state scalar, or 3×1 vector, or 4×1 vector.

Scalar case:

state = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with **rndcon** and **rndmult**.

If *state* = -1, **GAUSS** computes the starting seed based on the system clock.

3×1 vector case:

[1] the starting seed, uses the system clock if -1

[2] the multiplicative constant

[3] the additive constant

4×1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

OUTPUT

x *r*×*c* matrix, beta distributed random numbers.

newstate 4×1 vector:

[1] the updated seed

[2] the multiplicative constant
[3] the additive constant
[4] the original initialization seed

REMARKS The properties of the pseudo-random numbers in *x* are:

$$E(x) = \frac{a}{a+b}, Var(x) = \frac{(a*b)}{(a+b+1)*(a+b)^2}$$

$$0 < x < 1, a > 0, b > 0$$

r and *c* will be truncated to integers if necessary.

SOURCE `randlc.src`

TECHNICAL NOTES This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = (((a * seed) \ \% \ 2^{32}) + c) \ \% \ 2^{32}$$

where **%** is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant.

rndLCgam

PURPOSE Computes Gamma pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndLCgam**(*r*,*c*,*alpha*,*state*);

INPUT *r* scalar, number of rows of resulting matrix.

c scalar, number of columns of resulting matrix.

alpha *r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, shape argument for gamma distribution.

state scalar, or 3×1 vector, or 4×1 vector.

Scalar case:

state = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with **rndcon** and **rndmult**.

If *state* = -1, **GAUSS** computes the starting seed based on the system clock.

3×1 vector case:

[1] the starting seed, uses the system clock if -1

[2] the multiplicative constant

[3] the additive constant

4×1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

OUTPUT *x* *r*×*c* matrix, gamma distributed random numbers.

newstate 4×1 vector:

[1] the updated seed

[2] the multiplicative constant

[3] the additive constant

[4] the original initialization seed

REMARKS The properties of the pseudo-random numbers in *x* are:

$$E(x) = alpha, Var(x) = alpha$$

$$x > 0, alpha > 0$$

To generate **gamma(***alpha***,***theta***)** pseudo-random numbers where *theta* is a scale parameter, multiply the result of **rndLCgam** by *theta*.

Thus

```
z = theta * rndgam(1,1,alpha);
```

has the properties

$$E(z) = alpha * theta, Var(z) = alpha * theta^2$$

$$z > 0, alpha > 0, theta > 0$$

r and *c* will be truncated to integers if necessary.

SOURCE `randlc.src`

TECHNICAL NOTES This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = (((a * seed) \ \% \ 2^{32}) + c) \ \% \ 2^{32}$$

where **%** is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant.

rndLCi

PURPOSE Returns a matrix of random integers, $0 <= y < 2^{32}$, and the state of the random number generator.

FORMAT { *y*,*newstate* } = **rndLCi**(*r*,*c*,*state*);

INPUT

r scalar, row dimension.

c scalar, column dimension.

state scalar, or 3×1 vector, or 4×1 vector.

Scalar case:

state = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with **rndcon** and **rndmult**.

If *state*<0, **GAUSS** computes the starting seed based on the system clock.

3×1 vector case:

[1] the starting seed, uses the system clock if < 0

[2] the multiplicative constant

[3] the additive constant

4×1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

OUTPUT

y *r*×*c* matrix of random integers between 0 and 2^{32} - 1, inclusive.

newstate 4×1 vector:

[1] the updated seed

[2] the multiplicative constant

[3] the additive constant

[4] the original initialization seed

REMARKS *r* and *c* will be truncated to integers if necessary.

Each seed is generated from the preceding seed, using the formula

$$new\_seed = (((a * seed) \ \% \ 2^{32}) + c) \ \% \ 2^{32}$$

where **%** is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant. The new seeds are the values returned.

EXAMPLE

```
state = 13;
n = 2000000000;
k = 1000000;
c = 0;
min = 2^32+1;
max = -1;

do while c < n;
    { y,state } = rndLCi(k,1,state);
    min = minc(min | minc(y));
    max = maxc(max | maxc(y));
    c = c + k;
endo;

print "min " min;
print "max " max;
```

SEE ALSO **rndLCn**, **rndLCu**, **rndcon**, **rndmult**

rndLCn

PURPOSE Returns a matrix of standard normal (pseudo) random variables and the state of the random number generator.

FORMAT { *y*,*newstate* } = **rndLCn**(*r*,*c*,*state*);

INPUT

r	scalar, row dimension.
c	scalar, column dimension.
state	scalar, or 3×1 vector, or 4×1 vector.
	Scalar case:

state = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with **rndcon** and **rndmult**.

If *state*<0, **GAUSS** computes the starting seed based on the system clock.

3×1 vector case:

[1] the starting seed, uses the system clock if < 0

[2] the multiplicative constant

[3] the additive constant

4×1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

OUTPUT

y *r*×*c* matrix of standard normal random numbers.

newstate 4×1 vector:

[1] the updated seed

[2] the multiplicative constant

[3] the additive constant

[4] the original initialization seed

REMARKS *r* and *c* will be truncated to integers if necessary.

EXAMPLE

```
state = 13;
n = 2000000000;
k = 1000000;
c = 0;
submean = {};

do while c < n;
    { y,state } = rndLCn(k,1,state);
    submean = submean | meanc(y);
    c = c + k;
endo;
```

```
mean = meanc(submean);
print mean;
```

SEE ALSO **rndLCu**, **rndLCi**, **rndcon**, **rndmult**

TECHNICAL NOTES The normal random number generator is based on the uniform random number generator, using the fast acceptance-rejection algorithm proposed by Kinderman, A.J. and J.G. Ramage, "Computer Generation of Normal Random Numbers," *Journal of the American Statistical Association*, December 1976, Volume 71, Number 356, pp. 893-896. This algorithm calls the linear congruential uniform random number generator multiple times for each normal random number generated. See **rndLCu** for a description of the uniform random number generator algorithm.

rndLCnb

PURPOSE Computes negative binomial pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndLCnb**(*r*,*c*,*k*,*p*,*state*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
k	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, "event" argument for negative binomial distribution.
p	*r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, "probability" argument for negative binomial distribution.
state	scalar, or 3×1 vector, or 4×1 vector. **Scalar case:** *state* = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with **rndcon** and **rndmult**.

If *state* = -1, **GAUSS** computes the starting seed based on the system clock.

3×1 vector case:

[1] the starting seed, uses the system clock if -1

[2] the multiplicative constant

[3] the additive constant

4×1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

OUTPUT

x *r*×*c* matrix, negative binomial distributed random numbers.

newstate 4×1 vector:

[1] the updated seed

[2] the multiplicative constant

[3] the additive constant

[4] the original initialization seed

REMARKS The properties of the pseudo-random numbers in *x* are:

$$E(x) = \frac{k * p}{(1-p)}, Var(x) = \frac{k * p}{(1-p)^2}$$

$$x = 0, 1, \ldots, k > 0, 0 < p < 1$$

r and *c* will be truncated to integers if necessary.

SOURCE `randlc.src`

TECHNICAL NOTES This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = (((a * seed) \ \% \ 2^{32}) + c) \ \% \ 2^{32}$$

where **%** is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant.

rndLCp

PURPOSE Computes Poisson pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndLCp**(*r*,*c*,*lambda*,*state*);

INPUT

r scalar, number of rows of resulting matrix.

c scalar, number of columns of resulting matrix.

lambda *r*×*c* matrix, or *r*×1 vector, or 1×*c* vector, or scalar, shape argument for Poisson distribution.

state scalar, or 3×1 vector, or 4×1 vector.

Scalar case:

state = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with **rndcon** and **rndmult**.

If *state* = -1, **GAUSS** computes the starting seed based on the system clock.

3×1 vector case:

[1] the starting seed, uses the system clock if -1

[2] the multiplicative constant

[3] the additive constant

4×1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

OUTPUT *x* *r*×*c* matrix, Poisson distributed random numbers.

newstate 4×1 vector:

[1] the updated seed
[2] the multiplicative constant
[3] the additive constant
[4] the original initialization seed

REMARKS The properties of the pseudo-random numbers in *x* are:

$$E(x) = lambda, Var(x) = lambda$$

$$x = 0, 1, \ldots, lambda > 0$$

r and *c* will be truncated to integers if necessary.

SOURCE `randlc.src`

TECHNICAL NOTES This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = (((a * seed) \ \% \ 2^{32}) + c) \ \% \ 2^{32}$$

where **%** is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant.

rndLCu

PURPOSE Returns a matrix of uniform (pseudo) random variables and the state of the random number generator.

FORMAT { *y*,*newstate* } = **rndLCu**(*r*,*c*,*state*);

INPUT

r scalar, row dimension.

c scalar, column dimension.

state scalar, or 3×1 vector, or 4×1 vector.

Scalar case:

state = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with **rndcon** and **rndmult**.

If *state*<0, **GAUSS** computes the starting seed based on the system clock.

3×1 vector case:

[1] the starting seed, uses the system clock if < 0

[2] the multiplicative constant

[3] the additive constant

4×1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

OUTPUT

y *r*×*c* matrix of uniform random numbers, 0<=*y*<1.

newstate 4×1 vector:

[1] the updated seed

[2] the multiplicative constant

[3] the additive constant

[4] the original initialization seed

REMARKS

r and *c* will be truncated to integers if necessary.

Each seed is generated from the preceding seed, using the formula

$$new\_seed = (((a * seed) \ \% \ 2^{32}) + c) \ \% \ 2^{32}$$

where **%** is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant. A number between 0 and 1 is created by dividing *new_seed* by 2^{32}.

EXAMPLE

```
state = 13;
n = 2000000000;
k = 1000000;
c = 0;
submean = {};

do while c < n;
    { y,state } = rndLCu(k,1,state);
    submean = submean | meanc(y);
    c = c + k;
endo;

mean = meanc(submean);
print 0.5-mean;
```

SEE ALSO **rndLCn**, **rndLCi**, **rndcon**, **rndmult**

TECHNICAL NOTES This function uses a linear congruential method, discussed in Kennedy, W. J. Jr., and J. E. Gentle, *Statistical Computing*, Marcel Dekker, Inc., 1980, pp. 136-147.

rndLCvm

PURPOSE Computes von Mises pseudo-random numbers.

FORMAT { *x*,*newstate* } = **rndLCvm**(*r*,*c*,*m*,*k*,*state*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
m	$r \times c$ matrix, or $r \times 1$ vector, or $1 \times c$ vector, or scalar, means for vm distribution.
k	$r \times c$ matrix, or $r \times 1$ vector, or $1 \times c$ vector, or scalar, shape argument for vm distribution.

state scalar, or 3×1 vector, or 4×1 vector.

Scalar case:

state = starting seed value only. System default values are used for the additive and multiplicative constants.

The defaults are 1013904223, and 1664525, respectively. These may be changed with **rndcon** and **rndmult**.

If *state* = -1, **GAUSS** computes the starting seed based on the system clock.

3×1 vector case:

[1] the starting seed, uses the system clock if -1

[2] the multiplicative constant

[3] the additive constant

4×1 vector case:

state = the state vector returned from a previous call to one of the **rndLC** random number generators.

OUTPUT

x *r*×*c* matrix, von Mises distributed random numbers.

newstate 4×1 vector:

[1] the updated seed

[2] the multiplicative constant

[3] the additive constant

[4] the original initialization seed

REMARKS

r and *c* will be truncated to integers if necessary.

SOURCE

randlc.src

TECHNICAL NOTES

This function uses a linear congruential method, discussed in Kennedy, W.J. Jr., and J.E. Gentle, *Statistical Computing*, Marcel Dekker, Inc. 1980, pp. 136-147. Each seed is generated from the preceding seed using the formula

$$new\_seed = (((a * seed) \ \% \ 2^{32}) + c) \ \% \ 2^{32}$$

where **%** is the mod operator and where *a* is the multiplicative constant and *c* is the additive constant.

rndn

PURPOSE Creates a matrix of standard Normal (pseudo) random numbers.

FORMAT *y* = **rndn**(*r*,*c*);

INPUT
r scalar, row dimension.
c scalar, column dimension.

OUTPUT
y *r*×*c* 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. The other parameters of the uniform generator can be changed using **rndcon**, **rndmod**, and **rndmult**.

EXAMPLE
```
x = rndn(8100,1);
m = meanc(x);
s = stdc(x);
```

m = 0.002810

s = 0.997087

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**, **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." *Journal of the American Statistical Association*. Vol. 71 No. 356, Dec. 1976, 893-96.

rndnb

PURPOSE

Computes pseudo-random numbers with negative binomial distribution.

FORMAT

x = **rndnb**(*r*,*c*,*k*,*p*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
k	M×N matrix, E×E conformable with *r*×*c* resulting matrix, "event" parameters for negative binomial distribution.
p	K×L matrix, E×E conformable with *r*×*c* resulting matrix, "probability" parameters for negative binomial distribution.

OUTPUT

x	*r*×*c* matrix, negative binomial distributed pseudo-random numbers.

REMARKS

The properties of the pseudo-random numbers in *x* are:

$$\begin{aligned} E(x) &= k * p/(1-p) \\ Var(x) &= k * p/(1-p)^2 \\ x &= 0, 1, 2, \cdots, k \\ k &> 0 \\ p &> 0 \\ p &< 1 \end{aligned}$$

SOURCE

random.src

rndp

PURPOSE Computes pseudo-random numbers with Poisson distribution.

FORMAT *x* = **rndp**(*r*,*c*,*lambda*);

INPUT

r	scalar, number of rows of resulting matrix.
c	scalar, number of columns of resulting matrix.
lambda	M×N matrix, E×E conformable with *r*×*c* resulting matrix, shape parameters for Poisson distribution.

OUTPUT

x	*r*×*c* matrix, Poisson distributed pseudo-random numbers.

REMARKS The properties of the pseudo-random numbers in *x* are:

$$\begin{aligned} E(x) &= lambda \\ Var(x) &= lambda \\ x &= 0, 1, 2, \ldots \\ lambda &> 0 \end{aligned}$$

SOURCE random.src

rndu

PURPOSE Creates a matrix of uniform (pseudo) random variables.

FORMAT *y* = **rndu**(*r*,*c*);

INPUT

r	scalar, row dimension.
c	scalar, column dimension.

OUTPUT

y	$r \times c$ 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.

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.

rndvm

PURPOSE Computes von Mises pseudo-random numbers.

FORMAT *x* = **rndvm**(*r*,*c*,*m*,*k*);

INPUT
- *r* scalar, number of rows of resulting matrix.
- *c* scalar, number of columns of resulting matrix.
- *m* N×K matrix, E×E conformable with *r*×*c*, means for von Mises distribution.
- *k* L×M matrix, E×E conformable with *r*×*c*, shape argument for von Mises distribution.

OUTPUT
- *x* *r*×*c* matrix, von Mises distributed random numbers.

SOURCE `random.src`

rotater

PURPOSE Rotates the rows of a matrix.

FORMAT *y* = **rotater**(*x*,*r*);

INPUT
- *x* N×K matrix to be rotated.
- *r* N×1 or 1×1 matrix specifying the amount of rotation.

OUTPUT
- *y* N×K rotated matrix.

REMARKS The rotation is performed horizontally within each row of the matrix. A positive rotation value will cause the elements to move to the right. A negative rotation

value will cause the elements to move to the left. In either case, the elements that are pushed off the end of the row will wrap around to the opposite end of the same row.

If the rotation value is greater than or equal to the number of columns in *x*, then the rotation value will be calculated using (*r* **% cols(***x***)**).

EXAMPLE
```
y = rotater(x,r);
```

If $x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix}$ and $r = \begin{matrix} 1 \\ -1 \end{matrix}$ Then $y = \begin{matrix} 3 & 1 & 2 \\ 5 & 6 & 4 \end{matrix}$

If $x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{matrix}$ and $r = \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix}$ Then $y = \begin{matrix} 1 & 2 & 3 \\ 6 & 4 & 5 \\ 8 & 9 & 7 \\ 10 & 11 & 12 \end{matrix}$

SEE ALSO **shiftr**

round

PURPOSE Round to the nearest integer.

FORMAT *y* **= round(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array containing the rounded elements of *x*.

EXAMPLE
```
let x = { 77.68   -14.10,
```

```
                4.73   -158.88 };
y = round(x);
```

$$y = \begin{matrix} 78.00 & -14.00 \\ 5.00 & -159.00 \end{matrix}$$

SEE ALSO **trunc**, **floor**, **ceil**

rows

PURPOSE Returns the number of rows in a matrix.

FORMAT *y* **= rows(***x***);**

INPUT *x* N×K matrix or sparse matrix.

OUTPUT *y* scalar, number of rows in the specified matrix.

REMARKS If *x* is an empty matrix, **rows(***x***)** and **cols(***x***)** return 0.

EXAMPLE
```
x = ones(3,5);
y = rows(x);
```

$$x = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{matrix}$$

$$y = 3$$

SEE ALSO **cols**, **show**

rowsf

PURPOSE Returns the number of rows in a **GAUSS** data set (`.dat`) file or **GAUSS** matrix (`.fmt`) file.

FORMAT *y* **= rowsf(***f***);**

INPUT *f* file handle of an open file.

OUTPUT *y* scalar, number of rows in the specified file.

EXAMPLE
```
open fp = myfile;
r = rowsf(fp);
c = colsf(fp);
```

SEE ALSO **colsf**, **open**, **typef**

rref

PURPOSE Computes the reduced row echelon form of a matrix.

FORMAT *y* **= rref(***x***);**

INPUT *x* M×N matrix.

OUTPUT *y* M×N 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);
```

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

SOURCE rref.src

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.prg;`	No additional search will be made if the file is not found.
`run myprog.prg;`	The directories listed in **src_path** will be searched for `myprog.prg` if the file is not found in the current directory.

Programs can also be run by typing the filename on the OS command line when

starting **GAUSS**.

EXAMPLE **Example 1**

```
run myprog.prg;
```

Example 2

```
name = "myprog.prg";
run ^name;
```

SEE ALSO **#include**

satostrC

PURPOSE Copies from one string array to another using a C language format specifier string for each element.

FORMAT *y* **= satostrC(***sa***,***fmt***);**

INPUT *sa* N×M string array.

fmt 1×1, 1×M, or M×1 format specifier for each element copy.

OUTPUT *y* N×M formatted string array.

SOURCE strfns.src

SEE ALSO **strcombine**

save

PURPOSE Saves matrices, strings, or procedures to a disk file.

FORMAT **save** [[*vflag*]] [[**path**=*path*]] *x*, [[*lpath*=]]*y*;

INPUT *vflag* version flag.

-v89	not supported
-v92	supported on UNIX, Windows
-v96	supported on all platforms

See also FILE I/O, Chapter 20, for details on the various versions. The default format can be specified in `gauss.cfg` by setting the **dat_fmt_version** configuration variable. If **dat_fmt_version** is not set, the default is **v96**.

path literal or ^string, a default path to use for this and subsequent **save**'s.

x a symbol name, the name of the file the symbol will be saved in is the same as this with the proper extension added for the type of the symbol.

lpath literal or ^string, a local path and filename to be used for a particular symbol. This path will override the path previously set and the filename will override the name of the symbol being saved. The extension cannot be overridden.

y the symbol to be saved to *lpath*.

REMARKS **save** can be used to save matrices, strings, procedures, and functions. Procedures and functions must be compiled and resident in memory before they can be **save**'d.

The following extensions will be given to files that are **save**'d:

matrix	.fmt
string	.fst
procedure	.fcg
function	.fcg
keyword	.fcg

If the **path=** subcommand is used with **save**, the path string will be remembered until changed in a subsequent command. This path will be used whenever none is specified. The **save** path can be overridden in any particular **save** by specifying an explicit path and filename.

EXAMPLE

```
spath = "/gauss";
save path = ^spath x,y,z;
```

Save **x**, **y**, and **z** using /gauss as the path. This path will be used for the next **save** if none is specified.

```
svp = "/gauss/data";
save path = ^svp n, k, /gauss/quad1=quad;
```

n and **k** will be saved using /gauss/data as the **save** path, **quad** will be saved in /gauss with the name quad1.fmt. On platforms that use the backslash as the path separator, the double backslash is required inside double quotes to produce a backslash because it is the escape character in quoted strings. It is not required when specifying literals.

```
save path=/procs;
```

Change **save** path to /procs.

```
save path = /miscdata;
save /data/mydata1 = x, y, hisdata = z;
```

In the above program:

x would be saved in `/data/mydata1.fmt`
y would be saved in `/miscdata/y.fmt`
z would be saved in `/miscdata/hisdata.fmt`

SEE ALSO **datasave**, **load**, **saveall**, **saved**

saveall

PURPOSE Saves the current state of the machine to a compiled file. All procedures, global matrices and strings will be saved.

FORMAT **saveall** *fname*;

INPUT *fname* literal or ^string, the path and filename of the compiled file to be created.

REMARKS The file extension will be `.gcg`.

A file will be created containing all your matrices, strings, and procedures. No main code segment will be saved. This just means it will be a `.gcg` file with no main program code (see **compile**). The rest of the contents of memory will be saved, including all global matrices, strings, functions and procedures. Local variables are not saved. This can be used inside a program to take a snapshot of the state of your global variables and procedures. To reload the compiled image, use **run** or **use**.

```
library pgraph;
external proc xy,logx,logy,loglog,hist;
saveall pgraph;
```

This would create a file called `pgraph.gcg`, containing all the procedures,

strings and matrices needed to run **Publication Quality Graphics** programs. Other programs could be compiled very quickly with the following statement at the top of each:

```
use pgraph;
```

SEE ALSO **compile**, **run**, **use**

saved

PURPOSE Writes a matrix in memory to a **GAUSS** data set on disk.

FORMAT *y* **= saved(***x*,*dataset*,*vnames***);**

INPUT

x	N×K matrix to save in `.dat` file.
dataset	string, name of data set.
vnames	string or K×1 character vector, names for the columns of the data set.

OUTPUT

y	scalar, 1 if successful, otherwise 0.

REMARKS If *dataset* is null or 0, the data set name will be `temp.dat`.

If *vnames* is a null or 0, the variable names will begin with "X" and be numbered 1-K.

If *vnames* is a string or has fewer elements than *x* has columns, it will be expanded as explained under **create**.

The output data type is double precision.

EXAMPLE

```
x = rndn(100,3);
dataset = "mydata";
```

```
vnames = { height, weight, age };
if not saved(x,dataset,vnames);
    errorlog "Write error";
    end;
endif;
```

SOURCE saveload.src

SEE ALSO **loadd**, **writer**, **create**

savestruct

PURPOSE Saves a matrix of structures to a file on the disk.

FORMAT *retcode* **= saveStruct(***instance***,***file_name***);**

INPUT *instance* M×N matrix, instances of a structure.

file_name string, name of file on disk to contain matrix of structures.

OUTPUT *retcode* scalar, 0 if successful, otherwise 1.

REMARKS The file on the disk will be given a .fsr extension

EXAMPLE

```
#include ds.sdf

struct DS p0;
p0 = reshape(dsCreate,2,3);
retc = saveStruct(p2,"p2");
```

savewind

PURPOSE Save the current graphic panel configuration to a file.

LIBRARY pgraph

FORMAT *err* = **savewind**(*filename*);

INPUT *filename* string, name of file.

OUTPUT *err* scalar, 0 if successful, 1 if graphic panel matrix is invalid. Note that the file is written in either case.

REMARKS See the discussion on using graphics panels in GRAPHIC PANELS, Section 24.3.

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**(*x*,*y*);

INPUT *x* matrix, the X axis data.

y matrix, the Y axis data.

REMARKS *x* and *y* must each have at least 2 elements. Only the minimum and maximum values are necessary.

This routine fixes the scaling for all subsequent graphs until **graphset** is called. This also clears **xtics** and **ytics** whenever it is called.

If either of the arguments is a scalar missing, the main graphics function will set the scaling for that axis using the actual data.

If an argument has 2 elements, the first will be used for the minimum and the last will be used for the maximum.

If an argument has 2 elements, and contains a missing value, that end of the axis will be scaled from the data by the main graphics function.

If you want direct control over the axes endpoints and tick marks, use **xtics** or **ytics**. If **xtics** or **ytics** have been called after **scale**, they will override **scale**.

SOURCE pscale.src

SEE ALSO **xtics**, **ytics**, **ztics**, **scale3d**

scale3d

PURPOSE Fixes the scaling for subsequent graphs. The axes endpoints and increments are computed as a best guess based on the data passed to it.

LIBRARY pgraph

FORMAT **scale3d(***x*,*y*,*z***);**

INPUT
x matrix, the X axis data.
y matrix, the Y axis data.

z matrix, the Z axis data.

REMARKS *x*, *y* and *z* must each have at least 2 elements. Only the minimum and maximum values are necessary.

This routine fixes the scaling for all subsequent graphs until **graphset** is called. This also clears **xtics**, **ytics** and **ztics** whenever it is called.

If any of the arguments is a scalar missing, the main graphics function will set the scaling for that axis using the actual data.

If an argument has 2 elements, the first will be used for the minimum and the last will be used for the maximum.

If an argument has 2 elements, and contains a missing value, that end of the axis will be scaled from the data by the main graphics function.

If you want direct control over the axes endpoints and tick marks, use **xtics**, **ytics**, or **ztics**. If one of these functions have been called, they will override **scale3d**.

SOURCE pscale.src

SEE ALSO **scale**, **xtics**, **ytics**, **ztics**

scalerr

PURPOSE Tests for a scalar error code.

FORMAT *y* = **scalerr**(*c*);

INPUT *c* N×K matrix or sparse matrix or N-dimensional array, generally the return argument of a function or procedure call.

OUTPUT *y* scalar or [N-2]-dimensional array, 0 if the argument is not a scalar error code, or the value of the error code as an integer if the argument is an error code.

REMARKS Error codes in **GAUSS** are NaN's (Not A Number). These are not just scalar integer values. They are special floating point encodings that the math chip recognizes as not representing a valid number. See also **error**.

scalerr can be used to test for either those error codes that are predefined in **GAUSS** or an error code that the user has defined using **error**.

If *c* is an N-dimensional array, *y* will be an [N-2]-dimensional array, where each element corresponds to a 2-dimensional array described by the last two dimensions of *c*. For each 2-dimensional array in *c* that does not contain a scalar error code, its corresponding element in *y* will be set to zero. For each 2-dimensional array in *c* that does contain a scalar error code, its corresponding element in *y* will be set to the value of that error code as an integer. In other words, if *c* is a 5×5×10×10 array, *y* will be a 5×5 array, in which each element corresponds to a 10×10 array in *c* and contains either a zero or the integer value of a scalar error code.

If *c* is an empty matrix, **scalerr** will return 65535.

Certain functions will either return an error code or terminate a program with an error message, depending on the trap state. The **trap** command is used to set the trap state. The error code that will be returned will appear to most commands as a missing value code, but the **scalerr** function can distinguish between missing values and error codes and will return the value of the error code.

Following are some of the functions that are affected by the trap state:

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**

EXAMPLE

```
trap 1;
cm = invpd(x);
trap 0;
if scalerr(cm);
    cm = inv(x);
endif;
```

In this example **invpd** will return a scalar error code if the matrix **x** is not positive definite. If **scalerr** returns with a nonzero value, the program will use the **inv** function, which is slower, to compute the inverse. Since the trap state has been turned off, if **inv** fails, the program will terminate with a **Matrix singular** error message.

SEE ALSO **error**, **trap**, **trapchk**

scalinfnanmiss

PURPOSE Returns true if the argument is a scalar infinity, NaN, or missing value.

FORMAT *y* = **scalinfnanmiss**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* scalar, 1 if *x* is a scalar, infinity, NaN, or missing value, else 0.

SEE ALSO **isinfnanmiss**, **ismiss**, **scalmiss**

scalmiss

PURPOSE Tests to see if its argument is a scalar missing value.

FORMAT *y* = **scalmiss**(*x*);

INPUT *x* N×K 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, and a 0 for *x* = 1 + .i.

EXAMPLE
```
clear s;
do until eof(fp);
    y = readr(fp,nr);
    y = packr(y);
    if scalmiss(y);
        continue;
    endif;
```

```
    s = s+sumc(y);
endo;
```

In this example the **packr** function will return a scalar missing if every row of its argument contains missing values, otherwise it will return a matrix that contains no missing values. **scalmiss** is used here to test for a scalar missing returned from **packr**. If the test returns true, then the sum step will be skipped for that iteration of the read loop because there were no rows left after the rows containing missings were packed out.

schtoc

PURPOSE Reduces any 2×2 blocks on the diagional of the real Schur matrix returned from **schur**. The transformation matrix is also updated.

FORMAT { *schc*,*transc* } = **schtoc**(*sch*,*trans*);

INPUT

sch	real N×N matrix in Real Schur form, i.e., upper triangular except for possibly 2×2 blocks on the diagonal.
trans	real N×N matrix, the associated transformation matrix.

OUTPUT

schc	N×N matrix, possibly complex, strictly upper triangular. The diagonal entries are the eigenvalues.
transc	N×N matrix, possibly complex, the associated transformation matrix.

REMARKS Other than checking that the inputs are strictly real matrices, no other checks are made. If the input matrix *sch* is already upper triangular, it is not changed. Small off-diagonal elements are considered to be zero. See the source code for the test used.

EXAMPLE

```
{ schc, transc } = schtoc(schur(a));
```

This example calculates the complex Schur form for a real matrix **a**.

SOURCE schtoc.src

SEE ALSO **schur**

schur

PURPOSE Computes the Schur form of a square matrix.

FORMAT { *s*,*z* } = **schur**(*x*)

INPUT *x* K×K matrix.

OUTPUT *s* K×K matrix, Schur form.

z K×K 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 2×2 submatrices which correspond to complex eigenvalues of *x*. If *x* has no complex eigenvalues, *s* will be strictly upper triangular. To convert *s* to the complex Schur form, use the **Run-Time Library** function **schtoc**.

x is first reduced to upper Hessenberg form using orthogonal similiarity transformations, then reduced to Schur form through a sequence of QR decompositions.

schur uses the ORTRAN, ORTHES and HQR2 functions from EISPACK.

z is an orthogonal matrix that transforms *x* into *s* and vice versa. Thus

$$s = z'xz$$

and since z is orthogonal,

$$x = zsz'$$

EXAMPLE

```
let x[3,3] = 1 2 3
             4 5 6
             7 8 9;
{ s, z } = schur(x);
```

```
          16.11684397    4.89897949    0.00000000
s =       -0.00000000   -1.11684397   -0.00000000
           0.00000000    0.00000000   -0.00000000

           0.23197069    0.88290596    0.40824829
z =        0.52532209    0.23952042   -0.81649658
           0.81867350   -0.40386512    0.40824829
```

SEE ALSO **hess**

screen

PURPOSE Controls output to the screen.

FORMAT **screen on;**

screen off;

screen;

REMARKS When this is **on**, the results of all print statements will be directed to the window. When this is **off**, print statements will not be sent to the window. This

is independent of the statement **output on**, which will cause the results of all print statements to be routed to the current auxiliary output file.

If you are sending a lot of output to the auxiliary output file on a disk drive, turning the window off will speed things up.

The **end** statement will automatically perform **output off** and **screen on**.

screen with no arguments will print "Screen is on" or "Screen is off" on the console.

EXAMPLE

```
output file = mydata.asc reset;
screen off;
format /m1/rz 1,8;
open fp = mydata;
do until eof(fp);
    print readr(fp,200);;
endo;
fp = close(fp);
end;
```

The program above will write the contents of the **GAUSS** file `mydata.dat` into an ASCII file called `mydata.asc`. If `mydata.asc` already exists, it will be overwritten.

Turning the window off will speed up execution. The **end** statement above will automatically perform **output off** and **screen on**.

SEE ALSO **output**, **end**, **new**

searchsourcepath

PURPOSE Searches the source path and (if specified) the `src` subdirectory of the **GAUSS** installation directory for a specified file.

FORMAT *fpath* = **searchsourcepath**(*fname*,*srcdir*);

INPUT
fname string, name of file to search for.

srcdir scalar, one of the following:

0 do not search in the `src` subdirectory of the **GAUSS** installation directory.

1 search the `src` subdirectory first.

2 search the `src` subdirectory last.

OUTPUT *fpath* string, the path of *fname*, or null string if *fname* is not found.

REMARKS The source path is set by the **src_path** configuration variable in your **GAUSS** configuration file, `gauss.cfg`.

seekr

PURPOSE Moves the pointer in a `.dat` or `.fmt` file to a particular row.

FORMAT *y* = **seekr**(*fh*,*r*);

INPUT
fh scalar, file handle of an open file.

r scalar, the row number to which the pointer is to be moved.

OUTPUT *y* scalar, the row number to which the pointer has been moved.

REMARKS If *r* = -1, the current row number will be returned.

If *r* = 0, the pointer will be moved to the end of the file, just past the end of the last row.

rowsf returns the number of rows in a file.

```
seekr(fh,0) == rowsf(fh) + 1;
```

Do NOT try to seek beyond the end of a file.

SEE ALSO **open**, **readr**, **rowsf**

select (dataloop)

PURPOSE Selects specific rows (observations) in a data loop based on a logical expression.

FORMAT **select** *logical_expression*;

REMARKS Selects only those rows for which *logical_expression* is TRUE. Any variables referenced must already exist, either as elements of the source data set, as **extern**'s, or as the result of a previous **make**, **vector**, or **code** statement.

EXAMPLE
```
select age > 40 AND sex $=\,= 'MALE';
```

SEE ALSO **delete (dataloop)**

selif

PURPOSE Selects rows from a matrix. Those selected are the rows for which there is a 1 in the corresponding row of *e*.

FORMAT *y* = **selif**(*x*,*e*);

INPUT
x N×K matrix or string array.
e N×1 vector of 1's and 0's.

OUTPUT *y* M×K matrix or string array consisting of the rows of *x* for which there is a 1 in the corresponding row of *e*.

REMARKS The argument *e* will usually be generated by a logical expression using "dot" operators.

y will be a scalar missing if no rows are selected.

EXAMPLE
```
y = selif(x,x[.,2] .gt 100);
```

This example selects all rows of **x** in which the second column is greater than 100.

```
let x[3,3] = 0 10 20
             30 40 50
             60 70 80;

e = (x[.,1] .gt 0) .and (x[.,3] .lt 100);
y = selif(x,e);
```

The resulting matrix **y** is:

```
30   40   50
60   70   80
```

All rows for which the element in column 1 is greater than 0 and the element in column 3 is less than 100 are placed into the matrix **y**.

SEE ALSO **delif**, **scalmiss**

seqa, seqm

PURPOSE **seqa** creates an additive sequence. **seqm** creates a multiplicative sequence.

FORMAT *y* = **seqa**(*start*,*inc*,*n*);

y = **seqm**(*start*,*inc*,*n*);

INPUT

start	scalar specifying the first element.
inc	scalar specifying increment.
n	scalar specifying the number of elements in the sequence.

OUTPUT

y	$n \times 1$ vector containing the specified sequence.

REMARKS For **seqa**, *y* will contain a first element equal to *start*, the second equal to *start*+*inc*, and the last equal to *start*+*inc**(*n*-1).

For instance,

```
seqa(1,1,10);
```

will create a column vector containing the numbers 1, 2, . . . 10.

For **seqm**, *y* will contain a first element equal to *start*, the second equal to *start***inc*, and the last equal to $start*inc^{n-1}$.

For instance,

```
seqm(10,10,10);
```

will create a column vector containing the numbers 10, 100, . . . 10^{10}.

EXAMPLE

```
a = seqa(2,2,10)';
m = 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 N×1 (column) vectors.

SEE ALSO **recserar**, **recsercp**

setarray

PURPOSE Sets a contiguous subarray of an N-dimensional array.

FORMAT **setarray** *a*,*loc*,*src*;

INPUT

a	N-dimensional array.
loc	M×1 vector of indices into the array to locate the subarray of interest, where M is a value from 1 to N.
src	[N-M]-dimensional array, matrix, or scalar.

REMARKS **setarray** resets the specified subarray of *a* in place, without making a copy of the entire array. Therefore, it is faster than **putarray**.

If *loc* is an N×1 vector, then src must be a scalar. If *loc* is an [N-1]×1 vector, then *src* must be a 1-dimensional array or a 1×L vector, where L is the size of the fastest moving dimension of the array. If *loc* is an [N-2]×1 vector, then *src* must be a K×L matrix, or a K×L 2-dimensional array, where K is the size of the second fastest moving dimension.

Otherwise, if *loc* is an M×1 vector, then *src* must be an [N-M]-dimensional array, whose dimensions are the same size as the corresponding dimensions of array *a*.

EXAMPLE

```
a = arrayalloc(2|3|4|5|6,0);
src = arrayinit(4|5|6,5);
loc = { 2,1 };
setarray a,loc,src;
```

This example sets the contiguous 4×5×6 subarray of a beginning at [2,1,1,1,1] to the array *src*, in which each element is set to the specified value 5.

SEE ALSO **putarray**

setdif

PURPOSE Returns the unique elements in one vector that are not present in a second vector.

FORMAT *y* **= setdif(***v1*,*v2*,*typ***);**

INPUT

v1	N×1 vector.	
v2	M×1 vector.	
typ	scalar, type of data.	
	0	character, case sensitive.
	1	numeric.
	2	character, case insensitive.

OUTPUT

y	L×1 vector containing all unique values that are in *v1* and are not in *v2*, sorted in ascending order.

REMARKS Place smaller vector first for fastest operation.

When there are a lot of duplicates, it is faster to remove them first with unique before calling this function.

EXAMPLE

```
let v1 = mary jane linda john;
let v2 = mary sally;
typ = 0;
y = setdif(v1,v2,typ);
```

```
      JANE
y =   JOHN
     LINDA
```

SOURCE setdif.src

SEE ALSO **setdifsa**

setdifsa

PURPOSE Returns the unique elements in one string vector that are not present in a second string vector.

FORMAT *sy* **= setdifsa(***sv1***,***sv2***);**

INPUT

sv1	N×1 or 1×N string vector.
sv2	M×1 or 1×M string vector.

OUTPUT

sy	L×1 vector containing all unique values that are in *sv1* and are not in *sv2*, sorted in ascending order.

REMARKS Place smaller vector first for fastest operation.

When there are a lot of duplicates it is faster to remove them first with **unique** before calling this function.

EXAMPLE
```
string sv1 = { "mary", "jane", "linda", "john" };
string sv2 = { "mary", "sally" };

sy = setdifsa(sv1,sv2);
sy =jane
    john
    linda
```

SOURCE setdif.src

SEE ALSO **setdif**

setvars

PURPOSE Reads the variable names from a data set header and creates global matrices with the same names.

FORMAT *nvec* **= setvars(***dataset***);**

INPUT *dataset* string, the name of the **GAUSS** data set. Do not use a file extension.

OUTPUT *nvec* N×1 character vector, containing the variable names defined in the data set.

REMARKS **setvars** is designed to be used interactively.

EXAMPLE
```
nvec = setvars("freq");
```

SOURCE vars.src

SEE ALSO **makevars**

setvwrmode

PURPOSE Sets the graphics viewer mode.

LIBRARY pgraph

FORMAT *oldmode* = **setvwrmode(***mode***);**

INPUT *mode* string, new mode or null string.

one	Use only one viewer.
many	Use a new viewer for each graph.

OUTPUT *oldmode* string, previous *mode*.

REMARKS If *mode* is a null string, the current *mode* will be returned with no changes made.

If "**one**" is set, the viewer executable will be `vwr.exe`.

EXAMPLE
```
oldmode = setvwrmode("one");
call setvwrmode(oldmode);
```

SOURCE pgraph.src

SEE ALSO **pqgwin**

setwind

PURPOSE Sets the current graphic panel to a previously created graphic panel number.

LIBRARY pgraph

FORMAT **setwind**(*n*);

INPUT *n* scalar, graphic panel number.

REMARKS This function selects the specified graphic panel to be the current graphic panel. This is the graphic panel in which the next graph will be drawn.

See the discussion on using graphic panels in GRAPHICS PANELS, Section 24.3.

SOURCE pwindow.src

SEE ALSO **begwind**, **endwind**, **getwind**, **nextwind**, **makewind**, **window**

shell

PURPOSE Executes an operating system command.

FORMAT **shell** [[*s*]];

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

REMARKS **shell** lets you run shell commands and programs from inside **GAUSS**. If a command is specified, it is executed; when it finishes, you automatically return to **GAUSS**. If no command is specified, the shell is executed and control passes to it, so you can issue commands interactively. You have to type **exit** to get back to **GAUSS** in that case.

If you specify a command in a string variable, precede it with the ^ (caret).

EXAMPLE
```
comstr = "ls ./src";
shell ^comstr;
```

This lists the contents of the `./src` subdirectory, then returns to **GAUSS**.

```
shell cmp n1.fmt n1.fmt.old;
```

This compares the matrix file `n1.fmt` to an older version of itself, `n1.fmt.old`, to see if it has changed. When **cmp** finishes, control is returned to **GAUSS**.

```
shell;
```

This executes an interactive shell. The OS prompt will appear and OS commands or other programs can be executed. To return to **GAUSS**, type **exit**.

SEE ALSO **exec**,

shiftr

PURPOSE Shifts the rows of a matrix.

FORMAT *y* = **shiftr**(*x*,*s*,*f*);

INPUT

x	N×K matrix to be shifted.
s	scalar or N×1 vector specifying the amount of shift.
f	scalar or N×1 vector specifying the value to fill in.

OUTPUT

y	N×K 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 = shiftr(x,s,f);
```

$$\text{If } x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix} \quad \text{and } s = \begin{matrix} 1 \\ -1 \end{matrix} \quad \text{and } f = \begin{matrix} 99 \\ 999 \end{matrix}$$

$$\text{Then } y = \begin{matrix} 99 & 1 \\ 4 & 999 \end{matrix}$$

$$\text{If } x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \quad \text{and } s = \begin{matrix} 0 \\ 1 \\ 2 \end{matrix} \quad \text{and } f = 0$$

$$\text{Then } y = \begin{matrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 7 \end{matrix}$$

SEE ALSO **rotater**

show

PURPOSE Displays the global symbol table.

FORMAT **show** [[-*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

symbol the name of the symbol to be shown. If the last character is an asterisk (*****), all symbols beginning with the supplied characters will be shown.

REMARKS If there are no arguments, the entire symbol table will be displayed.

show is directed to the auxiliary output if it is open.

Here is an example listing with an explanation of the columns. Note that **show** does not display the column titles shown here:

```
Memory used     Name    Cplx  Type          References      Info
  128 bytes     a             MATRIX                        4,4
  672 bytes     add           KEYWORD       global refs     0=1
  192 bytes     area          FUNCTION      local refs      1=1
  256 bytes     c          C  MATRIX                        4,4
  296 bytes     p1            PROCEDURE     local refs      1=1
  384 bytes     p2            PROCEDURE     global refs     0=1
    8 bytes     ps1           STRUCT        sdat *
   16 bytes     s             STRING                        8 char
  312 bytes     s1            STRUCT        sdat            1,1
   40 bytes     sa            STRING ARRAY                  3,1
   56 bytes     sm            SPARSE MATRIX                15,15
 2104 bytes     token         PROCEDURE     local refs      2=1
  216 bytes     y             ARRAY         3 dims          2,3,4

672 bytes program space used
12 global symbols, 2000 maximum, 12 shown
0 active locals, 2000 maximum
1 active structure
```

The '`Memory used`' column gives the amount of memory used by each item.

The '`Name`' column gives the name of each symbol.

The '`Cplx`' column contains a '`C`' if the symbol is a complex matrix.

The '`Type`' column specifies the type of the symbol. It can be `ARRAY`, `FUNCTION`, `KEYWORD`, `MATRIX`, `PROCEDURE`, `STRING`, `STRING ARRAY`, or

STRUCT.

If the symbol is a procedure, keyword or function, the 'References' column will show if it makes any global references. If it makes only local references, the procedure or function can be saved to disk in an .fcg file with the **save** command. If the function or procedure makes any global references, it cannot be saved in an .fcg file.

If the symbol is a structure, the 'References' column will contain the structure type. A structure pointer is indicated by a * following the structure type.

The 'Info' column depends on the type of the symbol. If the symbol is a procedure or a function, it gives the number of values that the function or procedure returns and the number of arguments that need to be passed to it when it is called. If the symbol is a matrix, sparse matrix, string array or array of structures, then the 'Info' column gives the number of rows and columns. If the symbol is a string, then it gives the number of characters in the string. If the symbol is an N-dimensional array, then it gives the orders of each dimension. As follows:

Rets=Args	if procedure, keyword, or function
Row,Col	if matrix, sparse matrix, string array, or structure
Length	if string
OrdN,. . . ,Ord2,Ord1	if array, where N is the slowest moving dimension of the array, and Ord is the order (or size) of a dimension

If the symbol is an array of structures, the 'Info' column will display the size of the array. A scalar structure instance is treated as a 1×1 array of structures. If the symbol is a structure pointer, the 'Info' column will be blank.

The program space is the area of space reserved for all nonprocedure, nonfunction program code. The maximum program space can be controlled by the **new** command.

The maximum number of global and local symbols is controlled by the **maxglobals** and **maxlocals** configuration variables in gauss.cfg.

EXAMPLE

```
show -fpg eig*;
```

This command will show all functions and procedures that have global references and begin with **eig**.

```
show -m;
```

This command will show all matrices.

SEE ALSO **new**, **delete**

sin

PURPOSE Returns the sine of its argument.

FORMAT *y* **= sin(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array containing the sine of *x*.

REMARKS For real data, *x* should contain angles measured in radians.

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

EXAMPLE
```
let x = { 0, .5, 1, 1.5 };
y = sin(x);
```

```
y =
0.00000000
0.47942554
0.84147098
0.99749499
```

SEE ALSO **atan**, **cos**, **sinh**, **pi**

singleindex

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

FORMAT *si* **= singleindex(***i*,*o***);**

INPUT

i	N×1 vector of indices into an N-dimensional array.
o	N×1 vector of orders of an N-dimensional array.

OUTPUT

si	scalar, index of corresponding element in 1-dimensional array or vector.

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

EXAMPLE

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

a = arrayalloc(orders,0);
ai = { 2,1,3 };
setarray a, ai, 49;
v = vecr(a);
vi = singleindex(ai,orders);

print "ai = " ai;
print "vi = " vi;
print "getarray(a,ai) = " getarray(a,ai);
print "v[vi] = " v[vi];
```

produces:

```
ai =
       2.0000000
       1.0000000
       3.0000000
vi =        15.000000
getarray(a,ai) =        49.000000
v[vi] =        49.000000
```

This example allocates a 3-dimensional array **a** and sets the element corresponding to the index vector ai to 49. It then creates a vector, **v**, with the same data. The element in the array **a** that is indexed by **ai** corresponds to the element of the vector **v** that is indexed by **vi**.

SEE ALSO **arrayindex**

sinh

PURPOSE Computes the hyperbolic sine.

FORMAT *y* = **sinh**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* N×K 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);
```

```
x =  -1.570796
     -0.785398
      0.000000
      0.785398
      1.570796
      3.141593

y =  -2.301299
     -0.868671
      0.000000
      0.868671
      2.301299
     11.548739
```


SOURCE `trig.src`

sleep

PURPOSE Sleeps for a specified number of seconds.

FORMAT *unslept* **= sleep(***secs***);**

INPUT *secs* scalar, number of seconds to sleep.

OUTPUT *unslept* scalar, number of seconds not slept.

REMARKS *secs* does not have to be an integer. If your system does not permit sleeping for a fractional number of seconds, *secs* will be rounded to the nearest integer, with a minimum value of 1.

If a program sleeps for the full number of *secs* specified, **sleep** returns 0; otherwise, if the program is awakened early (e.g., by a signal), **sleep** returns the amount of time not slept.

A program may sleep for longer than *secs* seconds, due to system scheduling.

solpd

PURPOSE Solves a set of positive definite linear equations.

FORMAT *x* = **solpd(***b*,*A*);

INPUT

b	N×K matrix or M-dimensional array where the last two dimensions are N×K.
A	N×N symmetric positive definite matrix or M-dimensional array where the N×N 2-dimensional arrays described by the last two dimensions are symmetric and positive definite.

OUTPUT

x	N×K matrix or M-dimensional array where the last two dimensions are N×K, the solutions for the system of equations, $Ax = b$.

REMARKS *b* can have more than one column. If so, the system of equations is solved for each column, i.e., *A***x*[.,*i*] = *b*[.,*i*].

This function uses the Cholesky decomposition to solve the system directly. Therefore it is more efficient than using **inv(***A***)****b*.

If *b* and *A* are M-dimensional arrays, the sizes of their corresponding M-2 leading dimensions must be the same. The resulting array will contain the solutions for the system of equations given by each of the corresponding 2-dimensional arrays described by the two trailing dimensions of *b* and *A*. In other words, for a 10×4×2 array *b* and a 10×4×4 array *A*, the resulting array *x* will contain the solutions for each of the 10 corresponding 4×2 arrays contained in *b* and 4×4 arrays contained in *A*. Therefore, $A[n, ., .] * x[n, ., .] = b[n, ., .]$, for $1 <= n <= 10$.

solpd does not check to see that the matrix *A* is symmetric. **solpd** will look only at the upper half of the matrix including the principal diagonal.

If the *A* matrix is not positive definite:

trap 1 return scalar error code 30.
trap 0 terminate with an error message.

One obvious use for this function is to solve for least squares coefficients. The effect of this function is thus similar to that of the **/** operator.

If **X** is a matrix of independent variables, and **Y** is a vector containing the dependent variable, then the following code will compute the least squares coefficients of the regression of **Y** on **X**:

```
b = solpd(X'Y,X'X);
```


EXAMPLE

```
n = 5; format /lo 16,8;
A = rndn(n,n);
A = A'A;
x = rndn(n,1);
b = A*x;
x2 = solpd(b,A);
print "     X            solpd(b,A)       Difference";
print x~x2~x-x2;
```

produces:

```
     X            solpd(b,A)       Difference

 0.32547881       0.32547881      -4.9960036e-16
 1.5190182        1.5190182       -1.7763568e-15
 0.88099266       0.88099266       1.5543122e-15
 1.8192784        1.8192784       -2.2204460e-16
-0.060848175     -0.060848175     -1.4710455e-15
```

SEE ALSO **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		N×K matrix.
c		scalar specifying one column of *x* to sort on.

OUTPUT

y		N×K matrix equal to *x* and sorted on the column *c*.

REMARKS These functions will sort the rows of a matrix with respect to a specified column. That is, they will sort the elements of a column and will arrange all rows of the matrix in the same order as the sorted column.

sortc assumes that the column to sort on is numeric. **sortcc** assumes that the column to sort on contains character data.

The matrix may contain both character and numeric data, but the sort column must be all of one type. Missing values will sort as if their value is below $-\infty$.

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


      1  3  2
y =   3  4  8
      4  7  3
```

SEE ALSO **rev**

sortd

PURPOSE Sorts a data file on disk with respect to a specified variable.

FORMAT **sortd(***infile***,***outfile***,***keyvar***,***keytyp***);**

INPUT

infile	string, name of input file.
outfile	string, name of output file, must be different.
keyvar	string, name of key variable.
keytyp	scalar, type of key variable.

- **1** numeric key, ascending order.
- **2** character key, ascending order.
- **-1** numeric key, descending order.
- **-2** character key, descending order.

REMARKS The data set *infile* will be sorted on the variable *keyvar*, and will be placed in *outfile*.

If the inputs are null ("" or 0), the procedure will ask for them.

SOURCE sortd.src

SEE ALSO **sortmc**, **sortc**, **sortcc**, **sorthc**, **sorthcc**

sorthc, sorthcc

PURPOSE Sorts a matrix of numeric or character data, or a string array.

FORMAT *y* = **sorthc**(*x*,*c*);

y = **sorthcc**(*x*,*c*);

INPUT

x	N×K matrix or string array.
c	scalar specifying one column of *x* to sort on.

OUTPUT

y	N×K matrix or string array equal to *x* and sorted on the column *c*.

REMARKS These functions will sort the rows of a matrix or string array with respect to a specified column. That is, they will sort the elements of a column and will arrange all rows of the object in the same order as the sorted column.

sorthc assumes that the column to sort on is numeric. **sorthcc** assumes that the column to sort on contains character data.

If *x* is a matrix, it may contain both character and numeric data, but the sort column must be all of one type. Missing values will sort as if their value is below $-\infty$.

The sort is in ascending order. This function uses the heap sort algorithm.

If you need to obtain the matrix sorted in descending order, you can use:

```
rev(sorthc(x,c))
```

EXAMPLE

```
let x[3,3]= 4 7 3
```

```
                1 3 2
                3 4 8;
y = sorthc(x,1);
```

```
        4  7  3
x =     1  3  2
        3  4  8
```

```
        1  3  2
y =     3  4  8
        4  7  3
```


SEE ALSO **sortc**, **rev**

sortind, sortindc

PURPOSE Returns the sorted index of *x*.

FORMAT *ind* **= sortind(***x***);**

ind **= sortindc(***x***);**

INPUT *x* N×1 column vector.

OUTPUT *ind* N×1 vector representing sorted index of *x*.

REMARKS **sortind** assumes that *x* contains numeric data. **sortindc** assumes that *x* contains character data.

This function can be used to sort several matrices in the same way that some other reference matrix is sorted. To do this, create the index of the reference matrix, then use **submat** to rearrange the other matrices in the same way.

EXAMPLE

```
let x = 3 8 2 5 1 6 9
ind = sortind(x);
y = x[ind];
```

```
      3
      8
      2
x =   5
      1
      6
      9
```

```
        5
        3
        1
ind =   4
        6
        2
        7
```

```
      1
      2
      3
y =   5
      6
      8
      9
```

sortmc

PURPOSE Sorts a matrix on multiple columns.

FORMAT `y = sortmc(x,v);`

INPUT
- *x* N×K matrix to be sorted.
- *v* L×1 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* N×K sorted matrix.

SOURCE `sortmc.src`

SEE ALSO **sortd**, **sortc**, **sortcc**, **sorthc**, **sorthcc**

sortr, sortrc

PURPOSE Sorts rows of a matrix of numeric or character data.

FORMAT `y = sortr(x,r);`

`y = sortrc(x,r);`

INPUT
- *x* N×K matrix.
- *r* scalar, row of *x* on which to sort.

OUTPUT
- *y* N×K matrix equal to *x* and sorted on row *r*.

REMARKS These functions sort the columns of a matrix with respect to a specified row. That is, they sort the elements of a row and arrange all rows of the matrix in the same order as the sorted column.

sortr assumes the row on which to sort is numeric. **sortrc** assumes that the row on which to sort contains character data.

The matrix may contain both character and numeric data, but the sort row must be all of one type. Missing values will sort as if their value is below $-\infty$.

The sort will be in left to right ascending order. This function uses the Quicksort algorithm. If you need to obtain the matrix sorted left to right in descending order (i.e., ascending right to left), use

```
rev(sortr(x,r)')'
```

EXAMPLE

```
let x = { 4 7 3,
          1 3 2,
          3 4 8 };

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

```
      3  4  7
y =   2  1  1
      8  3  3
```

spCreate

PURPOSE Creates a sparse matrix from vectors of non-zero values, row indices, and column indices.

FORMAT *y* **= spCreate(***r*,*c*,*vals*,*rinds*,*cinds***);**

INPUT

r	scalar, rows of output matrix.
c	scalar, columns of output matrix.
vals	N×1 vector, non-zero values.
rinds	N×1 vector, row indices of corresponding non-zero values.
cinds	N×1 vector, column indices of corresponding non-zero values.

OUTPUT *y* *r*×*c* sparse matrix.

REMARKS Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spCreate**.

EXAMPLE
```
sparse matrix y;
vals = { 1,2,3,4 };
rinds = { 2,5,8,13 };
cinds = { 4,1,9,5 };

y = spCreate(15,10,vals,rinds,cinds);
```

This example creates a 15×10 sparse matrix **y**, containing the following non-zero values:

Non-zero value	Index
1	(2,4)
2	(5,1)
3	(8,9)
4	(13,5)

SEE ALSO **packedToSp**, **denseToSp**, **spEye**

spDenseSubmat

PURPOSE Returns a dense submatrix of a sparse matrix.

FORMAT *y* = **spDenseSubmat**(*x*,*rinds*,*cinds*);

INPUT
x	M×N sparse matrix.
rinds	K×1 vector, row indices.
cinds	L×1 vector, column indices.

OUTPUT *y* K×L dense matrix, the intersection of *rinds* and *cinds*.

REMARKS If *rinds* or *cinds* are scalar zeros, all rows or columns will be returned.

EXAMPLE
```
sparse matrix y;
x = { 0  0  0 10,
      0  2  0  0,
      0  0  0  0,
      5  0  0  0,
      0  0  0  3 };

y = denseToSp(x,0);
d = spDenseSubmat(y,0,1|3|4);
```

```
          0  0  10
          0  0   0
d =       0  0   0
          5  0   0
          0  0   3
```

SEE ALSO **spSubmat**

spDiagRvMat

PURPOSE Inserts submatrices along the diagonal of a sparse matrix.

FORMAT *y* **= spDiagRvMat(***x*,*inds*,*size*,*a***);**

INPUT

x	M×N sparse matrix.
inds	K×2 vector or scalar 0, row and column indices into *x* at which to place the corresponding submatrices in *a*.
size	K×2 vector or scalar 0, sizes of the corresponding submatrices in *a*.
a	K×L×P array, containing the submatrices to insert into *x*.

OUTPUT *y* M×N sparse matrix, a copy of *x* containing the specified insertions.

REMARKS Each row of *ind* must contain the row and column indices, respectively, that form the starting point for the insertion of the corresponding submatrix in *a*. If *ind* is a scalar 0, the starting point for the insertion of each submatrix will be one row and one column past the ending point of the previous insertion. The first insertion will begin at the [1,1] element.

Each row of *size* must contain the number of rows and columns in the corresponding submatrix in *a*. This allows you to insert submatrices of different sizes $L_i \times P_i$ by inserting them into the planes of an array that is K×MAX(L)×MAX(P) and padding the submatrices with zeros to MAX(L)×MAX(P). For each plane in *a*, **spDiagRvMat** extracts the submatrix *a*[*i*,1:*size*[*i*,1],1:*size*[*i*,2]] and inserts that into *x* at the location indicated by the corresponding row of *inds*. If *size* is a scalar 0, then each L×P plane of *a* is inserted into *x* as is.

EXAMPLE
```
declare sparse matrix x,y;
x = spEye(10);
sx1 = { 2 3, 5 8 };
sx2 = { 8 2 3 4, 7 9 5 6, 3 2 8 4 };
sx3 = { 4 7 2, 6 5 3 };
sx4 = { 9, 3 };
a = arrayinit(4|3|4,0);
a[1,1:2,1:2] = sx1;
a[2,.,.] = sx2;
a[3,1:2,1:3] = sx3;
a[4,1:2,1] = sx4;
inds = 0;
siz = { 2 2, 3 4, 2 3, 2 1 };
y = spDiagRvMat(x,inds,siz,a);

dx = spToDense(x);
dy = spToDense(y);
```

dx =

1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1

dy =

2	3	0	0	0	0	0	0	0	0
5	8	0	0	0	0	0	0	0	0
0	0	8	2	3	4	0	0	0	0
0	0	7	9	5	6	0	0	0	0
0	0	3	2	8	4	0	0	0	0
0	0	0	0	0	1	4	7	2	0
0	0	0	0	0	0	6	5	3	0
0	0	0	0	0	0	0	1	0	9
0	0	0	0	0	0	0	0	1	3
0	0	0	0	0	0	0	0	0	1

spEye

PURPOSE Creates a sparse identity matrix.

FORMAT *y* **= spEye(***n***);**

INPUT *n* scalar, order of identity matrix.

OUTPUT *y* $n \times n$ sparse identity matrix.

REMARKS Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spEye**.

EXAMPLE
```
sparse matrix y;
y = spEye(3);
d = spDenseSubmat(y,0,0);
```

```
          1.0000000   0.0000000   0.0000000
d =       0.0000000   1.0000000   0.0000000
          0.0000000   0.0000000   1.0000000
```

SEE ALSO **spCreate**, **spOnes**, **denseToSp**

spGetNZE

PURPOSE Returns the non-zero values in a sparse matrix, as well as their corresponding row and column indices.

FORMAT { *vals*,*rowinds*,*colinds* } = **spNumNZE**(*x*);

INPUT
x M×N sparse matrix.

OUTPUT
vals N×1 vector, non-zero values in *x*.
rinds N×1 vector, row indices of corresponding non-zero values.
cinds N×1 vector, column indices of corresponding non-zero values.

EXAMPLE
```
sparse matrix y;
x = { 0  0  0 10,
      0  2  0  0,
      0  0  0  0,
      5  0  0  0,
```

```
      0  0  0  3 };

y = denseToSp(x,0);
{ v,r,c } = spGetNZE(y);
```

```
v =
10
 2
 5
 3
```

```
r =
1
2
4
5
```

```
c =
4
2
1
4
```

SEE ALSO **spNumNZE**

spline

PURPOSE Computes a two-dimensional interpolatory spline.

FORMAT { *u*,*v*,*w* } = **spline**(*x*,*y*,*z*,*sigma*,*g*);

INPUT

x	1×K vector, x-abscissae (x-axis values).
y	N×1 vector, y-abscissae (y-axis values).

	z	K×N matrix, ordinates (z-axis values).
	sigma	scalar, tension factor.
	g	scalar, grid size factor.
OUTPUT	*u*	1×(K**g*) vector, x-abscissae, regularly spaced.
	v	(N**g*)×1 vector, y-abscissae, regularly spaced.
	w	(K**g*)×(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`

spNumNZE

PURPOSE Returns the number of non-zero elements in a sparse matrix.

FORMAT *n* = **spNumNZE**(*x*);

INPUT	*x*	M×N sparse matrix.
OUTPUT	*n*	scalar, the number of non-zero elements in *x*.

EXAMPLE
```
sparse matrix y;
x = { 0  0  0 10,
      0  2  0  0,
      0  0  0  0,
      5  0  0  0,
      0  0  0  3 };

y = denseToSp(x,0);
n = spNumNZE(y);
```

```
n = 4
```

SEE ALSO **spGetNZE**

spOnes

PURPOSE Generates a sparse matrix containing only ones and zeros

FORMAT *y* = **spOnes**(*r*,*c*,*rinds*,*cinds*);

INPUT

r	scalar, rows of output matrix.
c	scalar, columns of output matrix.
rinds	N×1 vector, row indices of ones.
cinds	N×1 vector, column indices of ones.

OUTPUT

y	*r*×*c* sparse matrix of ones.

REMARKS Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spOnes**.

EXAMPLE
```
sparse matrix y;
```

```
rinds = { 1, 3, 5 };
cinds = { 2, 1, 3 };
y = spOnes(5,4,rinds,cinds);
d = spDenseSubmat(y,0,0);
```

```
          0.0000000   1.0000000   0.0000000   0.0000000
          0.0000000   0.0000000   0.0000000   0.0000000
d =       1.0000000   0.0000000   0.0000000   0.0000000
          0.0000000   0.0000000   0.0000000   0.0000000
          0.0000000   0.0000000   1.0000000   0.0000000
```

SEE ALSO **spCreate**, **spEye**, **spZeros**, **denseToSp**

SpreadsheetReadM

PURPOSE Reads and writes Excel files.

FORMAT *xlsmat* **= SpreadsheetReadM(***file***,***range***,***sheet***);**

INPUT

file	string, name of `.xls` *file*.
range	string, *range* to read or write; e.g., "a1:b20".
sheet	scalar, *sheet* number.

OUTPUT

xlsmat	matrix of numbers read from Excel.

PORTABILITY **Windows** only

REMARKS If the read functions fail, they will return a scalar error code which can be decoded with **scalerr**. If the write function fails, it returns a non-zero error number.

SEE ALSO **scalerr**, **error**

SpreadsheetReadSA

PURPOSE Reads and writes Excel files.

FORMAT *xlssa* = **SpreadsheetReadSA(***file*,*range*,*sheet*);

INPUT
file	string, name of `.xls` *file*.
range	string, *range* to read or write; e.g., "a1:b20".
sheet	scalar, *sheet* number.

OUTPUT
xlssa	string array read from Excel.

PORTABILITY **Windows** only

REMARKS If the read functions fail, they will return a scalar error code which can be decoded with **scalerr**. If the write function fails, it returns a non-zero error number.

SEE ALSO **scalerr**, **error**

SpreadsheetWrite

PURPOSE Reads and writes Excel files.

FORMAT *xlsret* = **SpreadsheetWrite(***data*,*file*,*range*,*sheet*);

INPUT
data	matrix, string or string array, *data* to write.
file	string, name of `.xls` file.
range	string, *range* to read or write; e.g., "a1:b20".

sheet scalar, *sheet* number.

OUTPUT *xlsret* success code, 0 if successful, else error code.

PORTABILITY **Windows** only

REMARKS If the read functions fail, they will return a scalar error code which can be decoded with **scalerr**. If the write function fails, it returns a non-zero error number.

SEE ALSO **scalerr**, **error**

spScale

PURPOSE Scales a sparse matrix.

FORMAT { *a r s* } = **spScale**(*x*);

INPUT *x* M×N sparse matrix.

OUTPUT *a* M×N scaled sparse matrix.

r M×1 vector, row scale factors.

s N×1 vector, column scale factors.

REMARKS **spScale** scales the elements of the matrix by powers of 10 so that they are all within (-10,10).

EXAMPLE
```
x = { 25  -12   0,
        3    0 -11,
        8 -100   0 };

declare sparse matrix sm, smsc;
```

```
sm = denseToSp(x,0);

{ smsc, r, c } = spScale(sm);
```

```
      25   -12     0
sm =   3     0   -11
       8  -100     0
```

```
        2.5  -1.2   0.0
smsc =  0.3   0.0  -1.1
       0.08  -1.0   0.0
```

```
      0.1
r =   0.1
     0.01
```

```
      1.0
c =   1.0
      1.0
```

spSubmat

PURPOSE Returns a sparse submatrix of a sparse matrix.

FORMAT *y* = **spSubmat**(*x*,*rinds*,*cinds*);

INPUT

	x	M×N sparse matrix.
	rinds	K×1 vector, row indices.
	cinds	L×1 vector, column indices.

OUTPUT *s* K×L sparse matrix, the intersection of *rinds* and *cinds*.

REMARKS If *rinds* or *cinds* are scalar zeros, all rows or columns will be returned.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spSubmat**.

EXAMPLE
```
sparse matrix y;
sparse matrix z;
x = { 0  0  0 10,
      0  2  0  0,
      0  0  0  0,
      5  0  0  0,
      0  0  0  3 };

y = denseToSp(x,0);
z = spSubmat(y,1|3|4,0);
d = spDenseSubmat(z,0,0);
```

```
          0  0  0  10
d =       0  0  0   0
          5  0  0   0
```

SEE ALSO **spDenseSubmat**

spToDense

PURPOSE Converts a sparse matrix to a dense matrix.

FORMAT *y* = **spToDense**(*x*);

INPUT *x* M×N sparse matrix.

OUTPUT *y* M×N dense matrix.

REMARKS A dense matrix is just a normal format matrix.

EXAMPLE
```
sparse matrix y;
y = spEye(4);
d = spToDense(y);
```

$$\mathtt{d} = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

SEE ALSO **spDenseSubmat**, **denseToSp**

spTrTDense

PURPOSE Multiplies a sparse matrix transposed by a dense matrix.

FORMAT *y* **= spTrTDense(***s*,*d***);**

INPUT *s* N×M sparse matrix.

d N×L dense matrix.

OUTPUT *y* M×L dense matrix, the result of $s'd$.

REMARKS This may also be accomplished by the following code:

$$y = s'*d$$

However, **spTrTDense** will be more efficient.

SEE ALSO **spTScalar**

spTScalar

PURPOSE Multiplies a sparse matrix by a scalar.

FORMAT *y* = **spTScalar**(*s*,*scal*,*rinds*,*cinds*);

INPUT

s	N×M sparse matrix.
scal	scalar.
rinds	K×1 vector of row indices.
cinds	L×1 vector of column indices.

OUTPUT

y	K×L sparse matrix.

REMARKS Only the elements of *s* specified by *rinds* and *cinds* will be multiplied by *scal*. All other elements will be unchanged in the result.

To select all rows or all columns, input a scalar 0 for *rinds* or *cinds*.

Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spTScalar**.

EXAMPLE
```
sparse matrix y;
x = { 3  0  2  1,
      0  4  0  0,
      5  0  0  3,
      0  1  2  0 };

rinds = 0;
cinds = { 2,4 };
y = spTScalar(x,10,rinds,cinds);
d = spDenseSubmat(y,0,0);
```

$$\mathtt{d} = \begin{matrix} 3 & 0 & 2 & 10 \\ 0 & 40 & 0 & 0 \\ 5 & 0 & 0 & 30 \\ 0 & 10 & 2 & 0 \end{matrix}$$

SEE ALSO **spTrTDense**

spZeros

PURPOSE Creates a sparse matrix containing no non-zero values.

FORMAT *y* **= spZeros(***r***,***c***);**

INPUT

r	scalar, rows of output matrix.
c	scalar, columns of output matrix.

OUTPUT

y	*r*×*c* sparse matrix.

REMARKS Since sparse matrices are strongly typed in **GAUSS**, *y* must be defined as a sparse matrix before the call to **spZeros**.

EXAMPLE
```
sparse matrix y;
y = spZeros(4,3);
d = spDenseSubmat(y,0,0);
```

$$\mathtt{d} = \begin{matrix} 0.0000000 & 0.0000000 & 0.0000000 \\ 0.0000000 & 0.0000000 & 0.0000000 \\ 0.0000000 & 0.0000000 & 0.0000000 \\ 0.0000000 & 0.0000000 & 0.0000000 \end{matrix}$$

SEE ALSO **spOnes**, **spEye**, **createSp**

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 K×1 vector of start values.

GLOBAL INPUT

_sqp_A M×K matrix, linear equality constraint coefficients.

_sqp_B M×1 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 K×1 unknown parameter vector.

_sqp_EqProc scalar, pointer to a procedure that computes the nonlinear equality constraints. For example, the statement:

```
_sqp_EqProc = &eqproc;
```

tells **sqpSolve** that nonlinear equality constraints are to be placed on the parameters and where the procedure computing them is to be found. The procedure must have one input argument, the K×1 vector of parameters, and one output argument, the R×1 vector of computed constraints that are to be equal to zero. For example, suppose that you wish to place the following constraint:

```
p[1] * p[2] = p[3]
```

The procedure for this is:

```
proc eqproc(p);
    retp(p[1]*p[2]-p[3]);
endp;
```

_sqp_C M×K matrix, linear inequality constraint coefficients.

_sqp_D M×1 vector, linear inequality constraint constants.

These globals are used to specify linear inequality constraints of the following type:

```
_sqp_C * X >= _sqp_D
```

where *X* is the K×1 unknown parameter vector.

_sqp_IneqProc scalar, pointer to a procedure that computes the nonlinear inequality constraints. For example the statement:

```
_sqp_EqProc = &ineqproc;
```

tells **sqpSolve** that nonlinear equality constraints are to be placed on the parameters and where the procedure computing them is to be found. The procedure must have one input argument, the K×1 vector of parameters, and one output argument, the R×1 vector of computed constraints that are to be equal to zero. For example, suppose that you wish to place the following constraint:

```
p[1] * p[2] >= p[3]
```

The procedure for this is:

```
proc ineqproc(p);
   retp(p[1]*[2]-p[3]);
endp;
```

_sqp_Bounds K×2 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 1×2 matrix may be used. Default is:

[1] -1e256 [2] 1e256

_sqp_GradProc scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. For example, the statement:

```
_sqp_GradProc = &gradproc;
```

tells **sqpSolve** that a gradient procedure exists and where to find it. The user-provided procedure has two input arguments, a K×1 vector of parameter values and an N×P matrix of data. The procedure returns a single output argument, an N×K 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 P×1 vector of parameter values and an N×K data matrix. The procedure returns a single output argument, the P×P symmetric matrix of second order derivatives of the function evaluated at the parameter values.

_sqp_MaxIters scalar, maximum number of iterations. Default = 1e+5. Termination can be forced by pressing C on the keyboard.

_sqp_DirTol scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisifed, **sqpSolve** will exit the iterations.

_sqp_ParNames K×1 character vector, parameter names.

_sqp_PrintIters scalar, if nonzero, prints iteration information. Default = 0. Can be toggled during iterations by pressing P on the keyboard.

_sqp_FeasibleTest scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off.

_sqp_RandRadius scalar, if zero, no random search is attempted. If nonzero it is the radius of random search which is invoked whenever the usual line search fails. Default = .01.

__output scalar, if nonzero, results are printed. Default = 0.

OUTPUT

x K×1 vector of parameters at minimum.

f scalar, function evaluated at *x*.

lagr vector, created using **vput**. Contains the Lagrangean for the constraints. They may be extracted with the **vread** command using the following strings:

''lineq''	Lagrangeans of linear equality constraints,
''nlineq''	Lagrangeans of nonlinear equality constraints
''linineq''	Lagrangeans of linear inequality constraints
''nlinineq''	Lagrangeans of nonlinear inequality constraints
''bounds''	Lagrangeans of bounds

Whenever a constraint is active, its associated Lagrangean will be nonzero.

retcode return code:

0	normal convergence
1	forced exit
2	maximum number of iterations exceeded
3	function calculation failed
4	gradient calculation failed
5	Hessian calculation failed
6	line search failed
7	error with constraints

REMARKS

Pressing C on the keyboard will terminate iterations, and pressing P will toggle iteration output.

sqpSolve is recursive, that is, it can call itself with another function and set of global variables,

EXAMPLE

```
sqpSolveSet;

proc fct(x);
    retp( (x[1] + 3*x[2] + x[3])^2 + 4*(x[1] - x[2])^2 );
endp;

proc ineqp(x);
    retp(6*x[2] + 4*x[3] - x[1]^3 - 3);
endp;

proc eqp(x);
    retp(1-sumc(x));
endp;

_sqp_Bounds = { 0 1e256 };

start = { .1, .7, .2 };

_sqp_IneqProc = &ineqp;
_sqp_EqProc = &eqp;

{ x,f,lagr,ret } = sqpSolve( &fct,start );
```

SOURCE sqpsolve.src

sqpSolveMT

PURPOSE Solves the nonlinear programming problem.

INCLUDE sqpsolvemt.sdf

FORMAT *out1* **= sqpSolveMT(&***fct*,*par1*,*data1*,*c1*);

INPUT **&***fct* pointer to a procedure that computes the function to be minimized.

This procedure must have two input arguments, an instance of structure of type **PV** and an instance of a structure of type **DS**, and one output argument, either a 1×1 scalar or an N×1 vector of function values evaluated at the parameters stored in the **PV** instance using data stored in the **DS** instance.

par1 an instance of structure of type **PV**. The *par1* instance is passed to the user-provided procedure pointed to by **&***fct*. *par1* is constructed using the "**pack**" functions.

data1 an array of instances of a **DS** structure. This array is passed to the user-provided pointed by **&***fct* to be used in the objective function. **sqpSolveMT** does not look at this structure. Each instance contains the the following members which can be set in whatever way that is convenient for computing the objective function:

data1[*i*].dataMatrix N×K matrix, data matrix.

data1[*i*].dataArray N×K×L.. array, data array.

data1[*i*].vnames string array, variable names (optional).

data1[*i*].dsname string, data name (optional).

data1[*i*].type scalar, type of data (optional).

c1 an instance of an **sqpSolveMTControl** structure. Normally an instance is initialized by calling **sqpSolveMTControlCreate** and members of this instance can be set to other values by the user. For an instance named *c1*, the members are:

c1.A M×K matrix, linear equality constraint coefficients: *c1*.**A** * *p* = *c1*.**B** where *p* is a vector of the parameters.

c1.B M×1 vector, linear equality constraint constants: *c1*.**A** * *p* = *c1*.**B** where *p* is a vector of the parameters.

c1.C M×K matrix, linear inequality constraint coefficients: *c1*.**C** * *p* >= *c1*.**D** where *p* is a vector of the parameters.

c1.D M×1 vector, linear inequality constraint constants: *c1*.**C** * *p* >= *c1*.**D** where *p* is a vector of the parameters.

c1.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has one input argument, a structure of type **SQPdata**, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = ., i.e., no equality procedure.
c1.weights	vector, weights for objective function returning a vector. Default = 1.
c1.ineqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has one input argument, a structure of type **SQPdata**, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = ., i.e., no inequality procedure.
c1.bounds	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = -1e256 1e256 .
c1.covType	scalar, if 2, QML covariance matrix, else if 0, no covariance matrix is computed, else ML covariance matrix is computed.
c1.gradProc	scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. Default = ., i.e., no gradient procedure has been provided.
c1.hessProc	scalar, pointer to a procedure that computes the Hessian, i.e., the matrix of second order partial derivatives of the function with respect to the parameters. Default = ., i.e., no Hessian procedure has been provided.
c1.maxIters	scalar, maximum number of iterations. Default = 1e+5.
c1.dirTol	scalar, convergence tolerance for gradient of

estimated coefficients. Default = 1e-5. When this criterion has been satisfied **SQPSolve** exits the iterations.

c1.feasibleTest scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.

c1.randRadius scalar, If zero, no random search is attempted. If nonzero, it is the radius of random search which is invoked whenever the usual line search fails. Default = .01.

c1.output scalar, if nonzero, results are printed. Default = 0.

c1.printIters scalar, if nonzero, prints iteration information. Default = 0.

OUTPUT *out1* an instance of an **sqpSolveMTout** structure. For an instance named *out1*, the members are:

out1.par an instance of structure of type **PV** containing the parameter estimates will be placed in the member matrix *out1*.**par**.

out1.fct scalar, function evaluated at *x*.

out1.lagr an instance of a **SQPLagrange** structure containing the Lagrangeans for the constraints. The members are:

out1.lagr.lineq M×1 vector, Lagrangeans of linear equality constraints.

out1.lagr.nlineq N×1 vector, Lagrangeans of nonlinear equality constraints.

out1.lagr.linineq P×1 vector, Lagrangeans of linear inequality constraints.

out1.lagr.nlininequ Q×1 vector, Lagrangeans of nonlinear inequality constraints.

out1.lagr.bounds K×2 matrix, Lagrangeans of bounds.

Whenever a constraint is active, its associated Lagrangean will be nonzero. For any constraint that is inactive throughout the iterations as well as at convergence, the corresponding Lagrangean matrix will be set to a scalar missing value.

out1.retcode return code:

0 normal convergence.
1 forced exit.
2 maximum number of iterations exceeded.
3 function calculation failed.
4 gradient calculation failed.
5 Hessian calculation failed.
6 line search failed.
7 error with constraints.
8 function complex.

REMARKS There is one required user-provided procedure, the one computing the objective function to be minimized, and four other optional functions, one each for computing the equality constraints, the inequality constraints, the gradient of the objective function, and the Hessian of the objective function.

All of these functions have one input argument that is an instance of a structure of type struct **PV** and a second argument that is an instance of a structure of type struct **DS**. On input to the call to **sqpSolveMT**, the first argument contains starting values for the parameters and the second argument any required data. The data are passed in a separate argument because the structure in the first argument will be copied as it is passed through procedure calls which would be very costly if it contained large data matrices. Since **sqpSolveMT** makes no

changes to the second argument it will be passed by pointer thus saving time because its contents aren't copied.

Both of the structures of type **PV** are set up using the **PV** "**pack**" procedures, **pvPack**, **pvPackm**, **pvPacks**, and **pvPacksm**. These procedures allow for setting up a parameter vector in a variety of ways.

For example, we might have the following objective function for fitting a nonlinear curve to data:

```
proc Micherlitz(struct PV par1, struct DS data1);
    local p0,e,s2,x,y;
    p0 = pvUnpack(par1,"parameters");
    y = data1.dataMatrix[.,1];
    x = data1.dataMatrix[.,2];
    e = y - p0[1] - p0[2]*exp(-p0[3] * x);
    retp(e'*e);
endp;
```

In this example the dependent and independent variables are passed to the procedure as the first and second columns of a data matrix stored in a single **DS** structure. Alternatively these two columns of data can be entered into a vector of **DS** structures, one for each column of data:

```
proc Micherlitz(struct PV par1, struct DS data1);
    local p0,e,s2,x,y;
    p0 = pvUnpack(par1,"parameters");
    y = data1[1].dataMatrix;
    x = data1[2].dataMatrix;
    e = y - p0[1] - p0[2]*exp(-p0[3]*x);
    retp(e'*e);
endp;
```

The syntax is similar for the optional user-provided procedures. For example, to constrain the squared sum of the first two parameters to be greater than one in

the above problem, provide the following procedure:

```
proc ineqConst(struct PV par1, struct DS data1);
    local p0;
    p0 = pvUnpack(p0,"parameters");
    retp( (p0[2]+p0[1])^2 - 1 );
endp;
```

The following is a complete example for estimating the parameters of the Micherlitz equation in data with bounds constraints on the parameters and where an optional gradient procedure has been provided:


```
#include sqpSolveMT.sdf

struct DS d0;
d0 = dsCreate;

y =  3.183|
     3.059|
     2.871|
     2.622|
     2.541|
     2.184|
     2.110|
     2.075|
     2.018|
     1.903|
     1.770|
     1.762|
     1.550;

x = seqa(1,1,13);
d0.dataMatrix = y~x;
struct sqpSolveMTControl c0;
c0 = sqpSolveMTControlCreate;
```

```
c0.bounds = 0~100; /* constrains parameters */
                   /* to be positive  */

struct PV par1;
par1 = pvCreate;
pvPack(par1,.92|2.62|.114,"parameters");
struct sqpSolveMTout out1;
out1 = sqpSolveMT(&Micherlitz,par1,d0,c0);

print " parameter estimates ";
print pvUnPack(out1.par,"parameters");

proc Micherlitz(struct PV par1, struct DS data1);
    local p0,e,s2,x,y;
    p0 = pvUnpack(par1,"parameters");
    y = data1.dataMatrix[.,1];
    x = data1.dataMatrix[.,2];
    e = y - p0[1] - p0[2]*exp(-p0[3] * x);
    retp(e'*e);
endp;

proc grad(struct PV par1, struct DS data1);
    local p0,e,w,g,r,x,y;
    p0 = pvUnpack(par1,"parameters");
    y = data1.dataMatrix[.,1];
    x = data1.dataMatrix[.,2];
    g = zeros(3,1);
    w = exp(-p0[3] * x);
    e = y - p0[1] - p0[2]*w;
    r = e'*w;
    g[1] = -2*sumc(e);
    g[2] = -2*r;
    g[3] = 2*p0[1]*p0[2]*r;
    retp(g);
endp;
```

SOURCE sqpsolvemt.src

SEE ALSO **sqpSolveMTControlCreate**, **sqpSolveMTlagrangeCreate**, **sqpSolveOutCreate**

sqpSolveMTControlCreate

PURPOSE Creates an instance of a structure of type **sqpSolveMTcontrol** set to default values.

INCLUDE sqpsolvemt.sdf

FORMAT *s* = **sqpSolveMTControlCreate;**

OUTPUT *s* instance of structure of type **sqpSolveMTControl**.

SOURCE sqpsolvemt.src

SEE ALSO **sqpSolve**

sqpSolveMTlagrangeCreate

PURPOSE Creates an instance of a structure of type **sqpSolveMTlagrange** set to default values.

INCLUDE sqpsolvemt.sdf

FORMAT *s* = **sqpSolveMTlagrangeCreate;**

OUTPUT *s* instance of structure of type **sqpSolveMTlagrange**.

SOURCE sqpsolvemt.src

SEE ALSO **sqpSolve**

sqpSolveMToutCreate

PURPOSE Creates an instance of a structure of type **sqpSolveMTout** set to default values.

INCLUDE sqpsolvemt.sdf

FORMAT *s* = **sqpSolveMToutCreate;**

OUTPUT *s* instance of structure of type **sqpSolveMTout**.

SOURCE sqpsolvemt.src

SEE ALSO **sqpSolve**

sqpSolveSet

PURPOSE Resets global variables used by **sqpSolve** to default values.

FORMAT **sqpSolveSet;**

SOURCE sqpsolve.src

sqrt

PURPOSE Computes the square root of every element in *x*.

FORMAT *y* = **sqrt**(*x*);

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array, the square roots of each element of *x*.

REMARKS If *x* is negative, complex results are returned by default. You can turn the generation of complex numbers for negative inputs on or off in the **GAUSS** configuration file, and with the **sysstate** function, case 8. If you turn it off, **sqrt** will generate an error for negative inputs.

If *x* is already complex, the complex number state does not matter; **sqrt** will compute a complex result.

EXAMPLE
```
let x[2,2] = 1 2 3 4;
y = sqrt(x);
```

```
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* = **stdc**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* K×1 vector, the standard deviation of each column of *x*.

REMARKS This function essentially computes:

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

Thus, the divisor is N-1 rather than N, where N is the number of elements being summed. To convert to the alternate definition, multiply by

```
sqrt((N-1)/N)
```

EXAMPLE

```
y = rndn(8100,1);
std = stdc(y);
```

```
std = 1.008377
```

In this example, 8100 standard Normal random variables are generated, and their standard deviation is computed.

SEE ALSO **meanc**

stdsc

PURPOSE Computes the standard deviation of the elements in each column of a matrix.

FORMAT *y* = **stdsc**(*x*);

INPUT *x* N×K matrix.

OUTPUT *y* K×1 vector, the standard deviation of each column of *x*.

REMARKS This function essentially computes:

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

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

EXAMPLE
```
y = rndn(8100,1);
std = stdsc(y);
```

```
std = 1.0151475
```

In this example, 8100 standard Normal random variables are generated, and their standard deviation is computed.

SEE ALSO **stdc**, **astds**, **meanc**

stocv

PURPOSE Converts a string to a character vector.

FORMAT *v* **= stocv(*s*);**

INPUT *s* string, to be converted to character vector.

OUTPUT *v* N×1 character vector, contains the contents of *s*.

REMARKS **stocv** breaks *s* up into a vector of 8-character length matrix elements. Note that the character information in the vector is not guaranteed to be null-terminated.

EXAMPLE
```
s = "Now is the time for all good men";
v = stocv(s);
```

```
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* **= stof(*x*);**

INPUT *x* string or N×K 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 3×1 matrix containing the numbers 1, 2 and 3.

If *x* is a null string, **stof** will return a 0.

This uses the same input conversion routine as **loadm** and **let**. It will convert character elements and missing values. **stof** also converts complex numbers in the same manner as **let**.

SEE ALSO **ftos**, **ftocv**, **chrs**

stop

PURPOSE Stops a program and returns to the command prompt. Does not close files.

FORMAT **stop;**

REMARKS This command has the same effect as **end**, except it does not close files or the auxiliary output.

It is not necessary to put a **stop** or an **end** statement at the end of a program. If neither is found, an implicit **stop** is executed.

SEE ALSO **end**, **new**, **system**

strcombine

PURPOSE Converts an N×M string array to an N×1 string vector by combining each element in a column separated by a user-defined delimiter string.

FORMAT `y = strcombine(sa,delim,qchar);`

INPUT

sa	N×M string array.	
delim	1×1, 1×M, or M×1 delimiter string.	
qchar	scalar, 2×1, or 1×2 string vector containing quote characters as required:	
	scalar:	Use this character as quote character. If this is 0, no quotes are added.
	2×1 or 1×2 string vector:	Contains left and right quote characters.

OUTPUT

y	N×1 string vector result.

SOURCE `strfns.src`

SEE ALSO **satostrC**

strindx

PURPOSE Finds the index of one string within another string.

FORMAT `y = strindx(where,what,start);`

INPUT

where	string or scalar, the data to be searched.
what	string or scalar, the substring to be searched for in *where*.
start	scalar, the starting point of the search in *where* for an occurrence of *what*. The index of the first character in a string is 1.

OUTPUT

y	scalar containing the index of the first occurrence of *what*, within *where*, which is greater than or equal to *start*. If no occurrence is found, it will be 0.

REMARKS An example of the use of this function is the location of a name within a string of names:

```
z = "nameagepaysex";
x = "pay";
y = strindx(z,x,1);
```

y = 8

This function is used with **strsect** for extracting substrings.

SEE ALSO **strrindx**, **strlen**, **strsect**, **strput**

strlen

PURPOSE Returns the length of a string.

FORMAT *y* **= strlen(***x***);**

INPUT *x* string, N×K matrix of character data, or N×K string array.

OUTPUT *y* scalar containing the exact length of the string *x*, or N×K matrix or string array containing the lengths of the elements in *x*.

REMARKS The null character (ASCII 0) is a legal character within strings and so embedded nulls will be counted in the length of strings. The final terminating null byte is not counted, though.

For character matrices, the length is computed by counting the characters (maximum of 8) up to the first null in each element of the matrix. The null character, therefore, is not a valid character in matrices containing character data and is not counted in the lengths of the elements of those matrices.

EXAMPLE

```
x = "How long?";
y = strlen(x);
```

```
y = 9
```

SEE ALSO **strsect**, **strindx**, **strrindx**

strput

PURPOSE Lays a substring over a string.

FORMAT *y* **= strput(***substr***,***str***,***off***);**

INPUT

substr	string, the substring to be laid over the other string.
str	string, the string to receive the substring.
off	scalar, the offset in *str* to place *substr*. The offset of the first byte is 1.

OUTPUT

y	string, the new string.

EXAMPLE

```
str = "max";
sub = "imum";
f = 4;
y = strput(sub,str,f);
print y;
```

produces:

```
maximum
```

SOURCE strput.src

strrindx

PURPOSE Finds the index of one string within another string. Searches from the end of the string to the beginning.

FORMAT *y* **= strrindx(***where*,*what*,*start***);**

INPUT

where	string or scalar, the data to be searched.
what	string or scalar, the substring to be searched for in *where*.
start	scalar, the starting point of the search in *where* for an occurrence of *what*. *where* will be searched from this point backward for *what*.

OUTPUT

y	scalar containing the index of the last occurrence of *what*, within *where*, which is less than or equal to *start*. If no occurrence is found, it will be 0.

REMARKS A negative value for *start* causes the search to begin at the end of the string. An example of the use of **strrindx** is extracting a file name from a complete path specification:

```
path = "/gauss/src/ols.src";
ps = "/";
pos = strrindx(path,ps,-1);
if pos;
    name = strsect(path,pos+1,strlen(path)-pos);
else;
    name = "";
endif;
```

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 *y* **= strsect(***str***,***start***,***len***);**

INPUT

str	string or scalar from which the segment is to be obtained.
start	scalar, the index of the substring in *str*. The index of the first character is 1.
len	scalar, the length of the substring.

OUTPUT

y	string, the extracted substring, or a null string if *start* is greater than the length of *str*.

REMARKS If there are not enough characters in a string for the defined substring to be extracted, then a short string or a null string will be returned.

If *str* is a matrix containing character data, it must be scalar.

EXAMPLE
```
strng = "This is an example string."
y = strsect(strng,12,7);
```

y = “example”

SEE ALSO **strlen**, **strindx**, **strrindx**

strsplit

PURPOSE Splits an N×1 string vector into an N×K string array of the individual tokens.

FORMAT *sa* **= strsplit(***sv***);**

INPUT *sv* N×1 string array.

OUTPUT *sa* N×K string array.

REMARKS Each row of *sv* must contain the same number of tokens. The following characters are considered delimiters between tokens:

space	ASCII 32
tab	ASCII 9
comma	ASCII 44
newline	ASCII 10
carriage return	ASCII 13

Tokens containing delimiters must be enclosed in single or double quotes or parentheses. Tokens enclosed in single or double quotes will NOT retain the quotes upon translation. Tokens enclosed in parentheses WILL retain the parentheses after translation. Parentheses cannot be nested.

EXAMPLE

```
let string sv = {
    "dog 'cat fish' moose",
    "lion, zebra, elk",
    "seal owl whale"
    };

sa = strsplit(sv);
```

```
          'dog'   'cat fish'   'moose'
sa =      'lion'  'zebra'      'elk'
          'seal'  'owl'        'whale'
```

SEE ALSO **strsplitPad**

strsplitPad

PURPOSE Splits a string vector into a string array of the individual tokens. Pads on the right with null strings.

FORMAT *sa* **= strsplitPad(***sv*,*cols***);**

INPUT

sv	N×1 string array.
cols	scalar, number of columns of output string array.

OUTPUT

sa	N×*cols* string array.

REMARKS Rows containing more than *cols* tokens are truncated and rows containing fewer than *cols* tokens are padded on the right with null strings. The following characters are considered delimiters between tokens:

space	ASCII 32
tab	ASCII 9
comma	ASCII 44
newline	ASCII 10
carriage return	ASCII 13

Tokens containing delimiters must be enclosed in single or double quotes or parentheses. Tokens enclosed in single or double quotes will NOT retain the quotes upon translation. Tokens enclosed in parentheses WILL retain the parentheses after translation. Parentheses cannot be nested.

EXAMPLE

```
let string sv = {
    "dog 'cat fish' moose",
    "lion, zebra, elk, bird",
    "seal owl whale"
    };

sa = strsplitPad(sv, 4);
```

```
          'dog'   'cat fish'   'moose'   ''
sa =      'lion'  'zebra'      'elk'     'bird'
          'seal'  'owl'        'whale'   ''
```

SEE ALSO **strsplit**

strtodt

PURPOSE Converts a string array of dates to a matrix in DT scalar format.

FORMAT *x* = **strtodt**(*sa*,*fmt*);

INPUT

sa	N×K string array containing dates.
fmt	string containing date/time format characters.

OUTPUT

x	N×K matrix of dates in DT scalar format.

REMARKS The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number

20050910223505

represents 22:35:05 or 10:35:05 PM on September 10, 2005.

The following formats are supported:

YYYY	Four digit year
YR	Last two digits of year
MO	Number of month, 01-12
DD	Day of month, 01-31
HH	Hour of day, 00-23
MI	Minute of hour, 00-59
SS	Second of minute, 00-59

EXAMPLE

```
x = strtodt("2005-07-12 10:18:32",
    "YYYY-MO-DD HH:MI:SS");
print x;
```

produces:

```
20050712101832.0
```

```
x = strtodt("2005-07-12 10:18:32", "YYYY-MO-DD");
print x;
```

produces:

```
20050712000000.0
```

```
x = strtodt("10:18:32", "HH:MI:SS");
print x;
```

produces:

```
101832.0
```

```
x = strtodt("05-28-05", "MO-DD-YR");
print x;
```

produces:

```
20050528000000.0
```

SEE ALSO **dttostr**, **dttoutc**, **utctodt**

strtof

PURPOSE Converts a string array to a numeric matrix.

FORMAT *x* **= strtof(***sa***);**

INPUT *sa* N×K string array containing numeric data.

OUTPUT *x* N×K matrix.

REMARKS This function supports real matrices only. Use **strtofcplx** for complex data.

SEE ALSO **strtofcplx**, **ftostrC**

strtofcplx

PURPOSE Converts a string array to a complex numeric matrix.

FORMAT *x* **= strtofcplx(***sa***);**

INPUT *sa* N×K string array containing numeric data.

OUTPUT *x* N×K complex matrix.

REMARKS **strtofcplx** supports both real and complex data. It is slower than **strtof** for real matrices. **strtofcplx** requires the presence of the real part. The imaginary part can be absent.

SEE ALSO **strtof**, **ftostrC**

strtriml

PURPOSE Strips all whitespace characters from the left side of each element in a string array.

FORMAT *y* = **strtriml**(*sa*);

INPUT *sa* N×M string array.

OUTPUT *y* N×M string array.

SOURCE strfns.src

SEE ALSO **strtrimr**, **strtrunc**, **strtruncl**, **strtruncpad**, **strtruncr**

strtrimr

PURPOSE Strips all whitespace characters from the right side of each element in a string array.

FORMAT *y* = **strtrimr**(*sa*);

INPUT *sa* N×M string array.

OUTPUT *y* N×M string array.

SOURCE strfns.src

SEE ALSO **strtriml**, **strtrunc**, **strtruncl**, **strtruncpad**, **strtruncr**

strtrunc

PURPOSE Truncates all elements of a string array to not longer than the specified number of characters.

FORMAT *y* = **strtrunc**(*sa*,*maxlen*);

INPUT *sa* N×K string array.
maxlen 1×K or 1×1 matrix, maximum length.

OUTPUT *y* N×K string array result.

SEE ALSO **strtriml**, **strtrimr**, **strtruncl**, **strtruncpad**, **strtruncr**

strtruncl

PURPOSE Truncates the left side of all elements of a string array by a user-specified number of characters.

FORMAT *y* = **strtruncl**(*sa*,*ntrunc*);

INPUT *sa* N×M, N×1, 1×M, or 1×1 string array.
ntrunc N×M, N×1, 1×M, or 1×1 matrix containing the number of characters to strip.

OUTPUT *y* string array result.

SOURCE strfns.src

SEE ALSO **strtriml**, **strtrimr**, **strtrunc**, **strtruncpad**, **strtruncr**

strtruncpad

PURPOSE Truncates all elements of a string array to the specified number of characters, adding spaces on the end as needed to achieve the exact length.

FORMAT *y* = **strtruncpad(***sa*,*maxlen***);**

INPUT *sa* N×K string array.
maxlen 1×K or 1×1 matrix, maximum length.

OUTPUT *y* N×K string array result.

SEE ALSO **strtriml**, **strtrimr**, **strtrunc**, **strtruncl**, **strtruncr**

strtruncr

PURPOSE Truncates the right side of all elements of a string array by a user-specified number of characters.

FORMAT *y* = **strtruncr(***sa*,*ntrunc***);**

INPUT *sa* N×M, N×1, 1×M, or 1×1 string array.
ntrunc N×M, N×1, 1×M, or 1×1 matrix containing the number of characters to strip.

OUTPUT *y* String array result.

SOURCE strfns.src

SEE ALSO **strtriml**, **strtrimr**, **strtrunc**, **strtruncl**, **strtruncpad**

submat

PURPOSE Extracts a submatrix of a matrix, with the appropriate rows and columns given by the elements of vectors.

FORMAT *y* **= submat(***x*,*r*,*c***);**

INPUT *x* N×K matrix.

r L×M matrix of row indices.

c P×Q matrix of column indices.

OUTPUT *y* (L*M)×(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);
```

```
        1   2   3   4
x =     5   6   7   8
        9  10  11  12
```

$$y = \begin{matrix} 2 & 4 \\ 10 & 12 \end{matrix}$$

$$z = \begin{matrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{matrix}$$

SEE ALSO **diag**, **vec**, **reshape**

subscat

PURPOSE Changes the values in a vector depending on the category a particular element falls in.

FORMAT *y* **= subscat(***x*,*v*,*s***);**

INPUT

x N×1 vector.

v P×1 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 P×1 vector, containing values to be substituted.

OUTPUT

y N×1 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:

$$\begin{array}{rcccll} & x & \leq & v[1] & \rightarrow & s[1] \\ v[1] < & x & \leq & v[2] & \rightarrow & s[2] \\ & & & \cdots & & \\ v[p-1] < & x & \leq & v[p] & \rightarrow & s[p] \\ & x & > & v[p] & \rightarrow & \text{the original value of } x \end{array}$$

If missing is not a category specified in *v*, missings in *x* are passed through without change.

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);
```

```
y =
    10
    10
    10
    10
     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* **= substute(***x*,*e*,*v***);**

INPUT

x N×K matrix containing the data to be changed.

e L×M matrix, E×E conformable with *x* containing 1's and 0's.

Elements of *x* will be changed if the corresponding element of *e* is 1.

v P×Q matrix, E×E 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

```
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,"R");
```

$$e = \begin{matrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{matrix}$$

Here is what **x** looks like after substitution:

$$\mathbf{x} = \begin{matrix} R & 55 & 30 \\ N & 57 & 18 \\ Y & 24 & 3 \\ N & 63 & 38 \\ R & 55 & 32 \\ N & 37 & 11 \end{matrix}$$

SOURCE datatran.src

SEE ALSO **code**, **recode**

subvec

PURPOSE Extracts an N×1 vector of elements from an N×K matrix.

FORMAT *y* = **subvec**(*x*,*ci*);

INPUT
x N×K matrix.
ci N×1 vector of column indices.

OUTPUT *y* N×1 vector containing the elements in *x* indicated by *ci*.

REMARKS Each element of *y* is from the corresponding row of *x* and the column set by the corresponding row of *ci*. In other words, *y*[*i*] = *x*[*i*,*ci*[*i*]].

EXAMPLE
```
x = reshape(seqa(1,1,12),4,3);
ci = { 2,3,1,3 };
y = subvec(x,ci);
```

```
x =  1   2   3
     4   5   6
     7   8   9
    10  11  12

ci =  2
      3
      1
      3

y =  2
     6
     7
    12
```

sumc

PURPOSE Computes the sum of each column of a matrix or the sum across the second-fastest moving dimension of an L-dimensional array.

FORMAT *y* **= sumc(***x***);**

INPUT *x* N×K matrix or L-dimensional array where the last two dimensions are N×K.

OUTPUT *y* K×1 vector or L-dimensional array where the last two dimensions are K×1.

EXAMPLE
```
x = reshape(seqa(1,1,12),3,4));
y = sumc(x);
```

```
      1   2   3   4
x =   5   6   7   8
      9  10  11  12
```

```
      15
      18
y =   21
      24
```

```
a = areshape(seqa(1,1,24),2|3|4);
z = sumc(a);
```


a is a 2×3×4 array such that:

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

```
 1   2   3   4
 5   6   7   8
 9  10  11  12
```

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

```
13  14  15  16
17  18  19  20
21  22  23  24
```

z is a 2×4×1 array such that:

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

15
18
21
24

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

51
54
57
60

SEE ALSO **cumsumc**, **meanc**, **stdc**

sumr

PURPOSE Computes the sum of each row of a matrix or the sum of the fastest moving dimension of an L-dimensional array.

FORMAT *y* **= sumr(***x***);**

INPUT *x* N×K matrix or L-dimensional array where the last two dimensions are N×K.

OUTPUT *y* N×1 vector or L-dimensional array where the last two dimensions are N×1.

EXAMPLE
```
x = reshape(seqa(1,1,12),3,4);
y = sumr(x);
```

```
      1   2   3   4
x =   5   6   7   8
      9  10  11  12


      10
y =   26
      42
```

```
a = areshape(seqa(1,1,24),2|3|4);
z = sumr(a);
```


a is a 2×3×4 array such that:

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

```
 1   2   3   4
 5   6   7   8
 9  10  11  12
```

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

```
13  14  15  16
17  18  19  20
21  22  23  24
```

z is a 2×3×1 array such that:

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

```
10
26
42
```

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

58
74
90

SEE ALSO **sumc**

surface

PURPOSE Graphs a 3-D surface.

LIBRARY pgraph

FORMAT **surface**(*x*,*y*,*z*);

INPUT

x 1×K vector, the X axis data.

y N×1 vector, the Y axis data.

z N×K matrix, the matrix of height data to be plotted.

GLOBAL INPUT

_psurf 2×1 vector, controls 3-D surface characteristics.

[1] if 1, show hidden lines. Default 0.

[2] color for base (default 7). The base is an outline of the X-Y plane with a line connecting each corner to the surface. If 0, no base is drawn.

_pticout scalar, if 0 (default), tick marks point inward, if 1, tick marks point outward.

_pzclr Z level color control.

There are 3 ways to set colors for the Z levels of a surface graph.

1. To specify a single color for the entire surface plot, set the color control variable to a scalar value 1–15. For example:

```
_pzclr = 15;
```

2. To specify multiple colors distributed evenly over the entire Z range, set the color control variable to a vector containing the desired colors only. **GAUSS** will automatically calculate the required corresponding Z values for you. The following example will produce a three color surface plot, the Z ranges being lowest=blue, middle=light blue, highest=white:

```
_pzclr = { 1, 10, 15 };
```

3. To specify multiple colors distributed over selected ranges, the Z ranges as well as the colors must be manually input by the user. The following example assumes -0.2 to be the minimum value in the *z* matrix:

```
_pzclr = { -0.2  1,
  /*  z >= -0.2 blue */
           0.0 10,
  /*  z >=  0.0 light blue */
           0.2 15 };
  /*  z >=  0.2 white */
```

Since a Z level is required for each selected color, the user must be responsible to compute the minimum value of the *z* matrix as the first Z range element. This may be most easily accomplished by setting the **_pzclr** matrix as shown above (the first element being an arbitrary value), then resetting the first element to the minimum *z* value as follows:

```
_pzclr = { 0.0   1,
           0.0  10,
           0.2  15 };
_pzclr[1,1] = minc(minc(z));
```

See Colors, Section 24.5, for the list of available colors.

REMARKS **surface** uses only the minimum and maximum of the X axis data in generating the graph and tick marks.

SOURCE psurface.src

SEE ALSO **volume**, **view**

svd

PURPOSE Computes the singular values of a matrix.

FORMAT *s* **= svd(***x***);**

INPUT *x* N×P matrix whose singular values are to be computed.

OUTPUT *s* M×1 vector, where M = min(N,P), containing the singular values of *x* arranged in descending order.

GLOBAL INPUT **_svderr** scalar, if not all of the singular values can be computed, **_svderr** will be nonzero. The singular values in *s*[**_svderr**+1], . . . *s*[M] will be correct.

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

trap 0 set **_svderr** and terminate with message
trap 1 set **_svderr** and continue execution

EXAMPLE
```
x = { 4 3 6 7,
      8 2 9 5 };
y = svd(x);
```

```
y =  16.521787
      3.3212254
```

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 { *u*,*s*,*v* } = **svd1**(*x*);

INPUT

x	N×P matrix whose singular values are to be computed.

OUTPUT

u	N×N matrix, the left singular vectors of *x*.
s	N×P diagonal matrix, containing the singular values of *x* arranged in descending order on the principal diagonal.
v	P×P matrix, the right singular vectors of *x*.

GLOBAL OUTPUT

_svderr	scalar, if all of the singular values are correct, **_svderr** is 0. If not all of the singular values can be computed, **_svderr** is set and the diagonal elements of *s* with indices greater than **_svderr** are correct.

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

trap 0	set **_svderr** and terminate with message
trap 1	set **_svderr** and continue execution

EXAMPLE

```
x = rndu(3,3);
{ u, s, v } = svd1(x);
```

```
          0.97847012   0.20538614   0.59906497
x =       0.85474208   0.79673540   0.22482095
          0.33340653   0.74443792   0.75698778
```

```
          −0.57955818    0.65204491    0.48882486
u =       −0.61005618    0.05056673   −0.79074298
          −0.54031821   −0.75649219    0.36847767

          1.84994646   0.00000000   0.00000000
s =       0.00000000   0.60370542   0.00000000
          0.00000000   0.00000000   0.47539239

          −0.68578561    0.71062560   −0.15719208
v =       −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 N×P matrix whose singular values are to be computed.

OUTPUT

u N×N or N×P matrix, the left singular vectors of *x*. If N > P, then *u* will be N×P, containing only the P left singular vectors of *x*.

s N×P or P×P diagonal matrix, containing the singular values of *x* arranged in descending order on the principal diagonal. If N > P, then *s* will be P×P.

v P×P matrix, the right singular vectors of *x*.

GLOBAL OUTPUT

_svderr scalar, if all of the singular values are correct, **_svderr** is 0. If not all of the singular values can be computed, **_svderr** is set and the diagonal elements of *s* with indices greater than **_svderr** are correct.

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

trap 0 set **_svderr** and terminate with message
trap 1 set **_svderr** and continue execution

SOURCE svd.src

SEE ALSO **svd**, **svd1**, **svdcusv**

svdcusv

PURPOSE Computes the singular value decomposition of *x* so that: $x = u * s * v'$ (compact *u*).

FORMAT { *u*,*s*,*v* } = **svdcusv**(*x*);

INPUT

x N×P matrix or K-dimensional array where the last two dimensions are N×P, whose singular values are to be computed.

OUTPUT

u N×N or N×P matrix or K-dimensional array where the last two dimensions are N×N or N×P, the left singular vectors of *x*. If N > P, *u* is N×P, containing only the P left singular vectors of *x*.

s N×P or P×P diagonal matrix or K-dimensional array where the last two dimensions describe N×P or P×P diagonal arrays, the singular values of *x* arranged in descending order on the principal diagonal. If N > P, *s* is P×P.

v P×P matrix or K-dimensional array where the last two dimensions are P×P, the right singular vectors of *x*.

REMARKS If *x* is an array, the resulting arrays *u*, *s* and *v* will contain their respective results for each of the corresponding 2-dimensional arrays described by the two trailing dimensions of *x*. In other words, for a 10×4×5 array *x*, *u* will be a 10×4×4 array containing the left singular vectors of each of the 10 corresponding 4×5 arrays contained in *x*. *s* will be a 10×4×5 array and *v* will be a 10×5×5 array both containing their respective results for each of the 10 corresponding 4×5 arrays contained in *x*.

If not all of the singular values can be computed, *s*[1,1] is set to a scalar error code. Use **scalerr** to convert this to an integer. The diagonal elements of *s* with indices greater than **scalerr(***s***[1,1])** are correct. If **scalerr(***s***[1,1])** returns a 0, all of the singular values have been computed.

SEE ALSO **svd2**, **svds**, **svdusv**

svds

PURPOSE Computes the singular values of a *x*.

FORMAT *s* **= svds(***x***);**

INPUT *x* N×P matrix or K-dimensional array where the last two dimensions are N×P, whose singular values are to be computed.

OUTPUT *s* min(N,P)×1 vector or K-dimensional array where the last two dimensions are min(N,P)×1, the singular values of *x* arranged in descending order.

REMARKS If *x* is an array, the result will be an array containing the singular values of each of the 2-dimensional arrays described by the two trailing dimensions of *x*. In other words, for a 10×4×5 array *x*, *s* will be a 10×4×1 array containing the singular values of each of the 10 4×5 arrays contained in *x*.

If not all of the singular values can be computed, *s*[1] is set to a scalar error

code. Use **scalerr** to convert this to an integer. The elements of *s* with indices greater than **scalerr(***s***[1])** are correct. If **scalerr(***s***[1])** returns a 0, all of the singular values have been computed.

SEE ALSO **svd**, **svdcusv**, **svdusv**

svdusv

PURPOSE Computes the singular value decomposition of *x* so that: $x = u * s * v'$.

FORMAT { *u*,*s*,*v* } = **svdusv(***x***);**

INPUT

x N×P matrix or K-dimensional array where the last two dimensions are N×P, whose singular values are to be computed.

OUTPUT

u N×N matrix or K-dimensional array where the last two dimensions are N×N, the left singular vectors of *x*.

s N×P diagonal matrix or K-dimensional array where the last two dimensions describe N×P diagonal arrays, the singular values of *x* arranged in descending order on the principal diagonal.

v P×P matrix or K-dimensional array where the last two dimensions are P×P, the right singular vectors of *x*.

REMARKS If *x* is an array, the resulting arrays *u*, *s* and *v* will contain their respective results for each of the corresponding 2-dimensional arrays described by the two trailing dimensions of *x*. In other words, for a 10×4×5 array *x*, *u* will be a 10×4×4 array containing the left singular vectors of each of the 10 corresponding 4×5 arrays contained in *x*. *s* will be a 10×4×5 array and *v* will be a 10×5×5 array both containing their respective results for each of the 10 corresponding 4×5 arrays contained in *x*.

If not all of the singular values can be computed, *s*[1,1] is set to a scalar error code. Use **scalerr** to convert this to an integer. The diagonal elements of *s*

with indices greater than **scalerr(***s***[1,1])** are correct. If **scalerr(***s***[1,1])** returns a 0, all of the singular values have been computed.

SEE ALSO **svd1**, **svdcusv**, **svds**

sysstate

PURPOSE Gets or sets general system parameters.

FORMAT { *rets...* } **= sysstate(***case***,***y***);**

REMARKS The available cases are as follows:

Case 1	**Version Information** Returns the current **GAUSS** version information in an 8-element numeric vector.
Cases 2-7	**GAUSS System Paths** Gets or sets **GAUSS** system path.
Case 8	**Complex Number Toggle** Controls automatic generation of complex numbers in **sqrt**, **ln**, and **log** for negative arguments.
Case 9	**Complex Trailing Character** Gets or sets trailing character for the imaginary part of a complex number.
Case 10	**Printer Width** Gets or sets **lprint** width.
Case 11	**Auxiliary Output Width** Gets or sets the auxiliary output width.
Case 13	**LU Tolerance** Gets or sets singularity tolerance for LU decomposition in current thread.

Case 14 **Cholesky Tolerance**

Gets or sets singularity tolerance for Cholesky decomposition in current thread.

Case 15 **Screen State**

Gets or sets window state as controlled by **screen** command.

Case 18 **Auxiliary Output**

Gets auxiliary output parameters.

Case 19 **Get/Set Format**

Gets or sets format parameters.

Case 21 **Imaginary Tolerance**

Gets or sets imaginary tolerance in current thread.

Case 22 **Source Path**

Gets or sets the path the compiler will search for source files.

Case 24 **Dynamic Library Directory**

Gets or sets the path for the default dynamic library directory.

Case 25 **Temporary File Path**

Gets or sets the path **GAUSS** will use for temporary files.

Case 26 **Interface Mode**

Returns the current interface mode.

Case 28 **Random Number Generator Parameters**

Gets or sets parameters used by the random number generation commands.

Case 30 **Base Year Toggle**

Specifies whether year value returned by **date** is to include base year (1900) or not.

Case 32 **Global LU Tolerance**

Gets or sets global singularity tolerance for LU decomposition.

Case 33 **Global Cholesky Tolerance**

Gets or sets global singularity tolerance for Cholesky decomposition.

Case 34 **Global Imaginary Tolerance**

Gets or sets global imaginary tolerance.

Case 1: Version Information

PURPOSE Returns the current **GAUSS** version information in an 8-element numeric vector.

FORMAT *vi* = **sysstate(1,0);**

OUTPUT *vi* 8×1 numeric vector containing version information:

[1] Major version number.
[2] Minor version number.
[3] Revision.
[4] Machine type.
[5] Operating system.
[6] Runtime module.
[7] Light version.
[8] Always 0.

vi[4] indicates the type of machine on which **GAUSS** is running:

1 Intel x86
2 Sun SPARC
3 IBM RS/6000
4 HP 9000
5 SGI MIPS
6 DEC Alpha
7 Mac 32-bit PowerPC

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 2-7: GAUSS System Paths

PURPOSE Gets or sets **GAUSS** system path.

FORMAT *oldpath* = **sysstate**(*case*,*path*);

INPUT *case* scalar 2-7, path to set.

2 `.exe` file location.
3 **loadexe** path.
4 **save** path.
5 **load**, **loadm** path.
6 **loadf**, **loadp** path.
7 **loads** path.

path scalar 0 to get path, or string containing the new path.

OUTPUT *oldpath* string, original path.

REMARKS If *path* is of type matrix, the path will be returned but not modified.

Case 8: Complex Number Toggle

PURPOSE Controls automatic generation of complex numbers in **sqrt**, **ln** and **log** for negative arguments.

FORMAT *oldstate* = **sysstate(8,***state***);**

INPUT *state* scalar, 1, 0, or -1

OUTPUT *oldstate* scalar, the original state.

REMARKS If *state* = 1, **log**, **ln**, and **sqrt** will return complex numbers for negative arguments. If *state* = 0, the program will terminate with an error message when negative numbers are passed to **log**, **ln**, and **sqrt**. If *state* = -1, the current state is returned and left unchanged. The default state is 1.

Case 9: Complex Trailing Character

PURPOSE Gets or sets trailing character for the imaginary part of a complex number.

FORMAT *oldtrail* **= sysstate(9,***trail***);**

INPUT *trail* scalar 0 to get character, or string containing the new trailing character.

OUTPUT *oldtrail* string, the original trailing character.

REMARKS The default character is "**i**".

Case 10: Printer Width

PURPOSE Gets or sets **lprint** width.

FORMAT *oldwidth* **= sysstate(10,***width***);**

INPUT *width* scalar, new printer width.

OUTPUT *oldwidth* scalar, the current original width.

REMARKS If *width* is 0, the printer width will not be changed.

This may also be set with the **lpwidth** command.

SEE ALSO **lpwidth**

Case 11: Auxiliary Output Width

PURPOSE Gets or sets the auxiliary output width.

FORMAT *oldwidth* = **sysstate(11,***width***);**

INPUT *width* scalar, new output width.

OUTPUT *oldwidth* scalar, the original output width.

REMARKS If *width* is 0 then the output width will not be changed.

This may also be set with the **outwidth** command.

SEE ALSO **outwidth**

Case 13: LU Tolerance

PURPOSE Gets or sets singularity tolerance for LU decomposition in current thread.

FORMAT *oldtol* = **sysstate(13,***tol***);**

INPUT *tol* scalar, new tolerance.

OUTPUT *oldtol* scalar, the original tolerance.

REMARKS The tolerance must be ≥ 0. If *tol* is negative, the tolerance is returned and left unchanged.

This tolerance is thread-safe. It must be set in the same thread in which it is to be referenced. To set the global singularity tolerance for LU decomposition, use case 32.

SEE ALSO **croutp**, **inv**

Case 14: Cholesky Tolerance

PURPOSE Gets or sets singularity tolerance for Cholesky decomposition in current thread.

FORMAT *oldtol* = **sysstate(14,***tol***);**

INPUT *tol* scalar, new tolerance.

OUTPUT *oldtol* scalar, the original tolerance.

REMARKS The tolerance must be ≥ 0. If *tol* is negative, the tolerance is returned and left unchanged.

This tolerance is thread-safe. It must be set in the same thread in which it is to be referenced. To set the global singularity tolerance for Cholesky decomposition, use case 33.

SEE ALSO **chol**, **invpd**, **solpd**

Case 15: Screen State

PURPOSE Gets or sets window state as controlled by **screen** command.

FORMAT *oldstate* = **sysstate(15,***state***);**

INPUT *state* scalar, new window state.

OUTPUT *oldstate* scalar, the original window state.

REMARKS If *state* = 1, window output is turned on. If *state* = 0, window output is turned off. If *state* = -1, the state is returned unchanged.

SEE ALSO **screen**

Case 18: Auxiliary Output

PURPOSE Gets auxiliary output parameters.

FORMAT { *state*,*name* } = **sysstate(18,0);**

OUTPUT *state* scalar, auxiliary output state, 1 - on, 0 - off.
name string, auxiliary output filename.

SEE ALSO **output**

Case 19: Get/Set Format

PURPOSE Gets or sets format parameters.

FORMAT *oldfmt* = **sysstate(19,***fmt***);**

INPUT *fmt* scalar or 11×1 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* 11×1 vector containing the current format parameters. The characters in quotes are components of the format string that gets passed through to the C library **sprintf** function:

[1] format conversion type:

- **0** string format ("**s**").
- **1** compact format ("**g**").
- **2** auto format ("**#g**").
- **3** scientific format ("**e**").
- **4** decimal format ("**f**").
- **5** compact format, upper case ("**G**").
- **6** auto format, upper case ("**#G**").
- **7** scientific format, upper case ("**E**").

[2] justification:

- **0** right justification.
- **1** left justification ("**-**").

[3] sign:

- **0** sign used only for negative numbers.
- **1** sign always used ("**+**").

[4] leading zero:

- **0** no leading zero.
- **1** leading zero ("**0**").

[5] trailing character:

- **0** no trailing character.
- **1** trailing space (" ").
- **2** trailing comma ("**,**").
- **3** trailing tab ("**t**").

[6] row delimiter:

- **0** no row delimiter.
- **1** one newline between rows ("**n**").
- **2** two newlines between rows ("**n****n**").
- **3** print "**Row 1**, **Row 2**, ..." before each row ("**nRow %u****n**", where "**%u**" is the row number).

[7] carriage line feed position:

- **0** newline row delimiters positioned before rows.
- **1** newline row delimiters positioned after rows.

[8] automatic line feed for row vectors.

0 newline row delimiters occur between rows of a matrix only if that matrix has more than one row.

1 newline row delimiters occur between rows of a matrix, regardless of number of rows.

[9] field width.

[10] precision.

[11] formatted flag.

0 formatting disabled.

1 formatting enabled.

REMARKS If *fmt* is scalar 0, then the format parameters will be left unchanged.

See the **format** and **print** commands for more information on the formatting parameters.

SEE ALSO **format**, **print**

Case 21: Imaginary Tolerance

PURPOSE Gets or sets imaginary tolerance in current thread.

FORMAT *oldtol* = **sysstate(21,***tol***);**

INPUT *tol* scalar, the new tolerance.

OUTPUT *oldtol* scalar, the original tolerance.

REMARKS The imaginary tolerance is used to test whether the imaginary part of a complex matrix can be treated as zero or not. Functions that are not defined for complex matrices check the imaginary part to see if it can be ignored. The default tolerance is 2.23e-16, or machine epsilon.

If *tol*<0, the current tolerance is returned.

This tolerance is thread-safe. It must be set in the same thread in which it is to be referenced. To set the global imaginary tolerance, use case 34.

SEE ALSO **hasimag**

Case 22: Source Path

PURPOSE Gets or sets the path the compiler will search for source files.

FORMAT *oldpath* = **sysstate(22,***path***);**

INPUT *path* scalar 0 to get path, or string containing the new path.

OUTPUT *oldpath* string, original path.

REMARKS If *path* is a matrix, the current source path is returned.

This resets the **src_path** configuration variable. **src_path** is initially defined in the **GAUSS** configuration file, `gauss.cfg`.

path can list a sequence of directories, separated by semicolons.

Resetting **src_path** affects the path used for subsequent **run** and **compile** statements.

Case 24: Dynamic Library Directory

PURPOSE Gets or sets the path for the default dynamic library directory.

FORMAT *oldpath* = **sysstate(24,***path***);**

INPUT *path* scalar 0 to get path, or string containing the new path.

OUTPUT *oldpath* string, original path.

REMARKS If *path* is a matrix, the current path is returned.

path should list a single directory, not a sequence of directories.

Changing the dynamic library path does not affect the state of any DLL's currently linked to **GAUSS**. Rather, it determines the directory that will be searched the next time **dlibrary** is called.

UNIX

Changing the path has no effect on **GAUSS**'s default DLL, `libgauss.so`. `libgauss.so` must always be located in the GAUSSHOME directory.

Windows

Changing the path has no effect on **GAUSS**'s default DLL, `gauss.dll`. `gauss.dll` must always be located in the GAUSSHOME directory.

SEE ALSO **dlibrary**, **dllcall**

Case 25: Temporary File Path

PURPOSE Gets or sets the path **GAUSS** will use for temporary files.

FORMAT *oldpath* = **sysstate(25,***path***);**

INPUT *path* scalar 0 to get path, or string containing the new path.

OUTPUT *oldpath* string, original path.

REMARKS If *path* is of type matrix, the path will be returned but not modified.

Case 26: Interface Mode

PURPOSE Returns the current interface mode.

FORMAT *mode* = **sysstate(26,0);**

OUTPUT *mode* scalar, interface mode flag

- **0** non-X mode
- **1** terminal (-v) mode
- **2** X Windows mode

REMARKS A mode of 0 indicates that you're running a non-X version of **GAUSS**; i.e., a version that has no X Windows capabilities. A mode of 1 indicates that you're running an X Windows version of **GAUSS**, but in terminal mode; i.e., you started **GAUSS** with the -v flag. A mode of 2 indicates that you're running **GAUSS** in X Windows mode.

Case 28: Random Number Generator Parameters

PURPOSE Gets or sets the random number generator (RNG) parameters.

FORMAT *oldprms* = **sysstate(28,***prms***);**

INPUT *prms* scalar 0 to get parameters, or 3×1 matrix of new parameters.

[1] seed,	$0<\text{seed}<2^{32}$
[2] multiplier,	$0<\text{mult}<2^{32}$
[3] constant,	$0<=\text{const}<2^{32}$

REMARKS If *prms* is a scalar 0, the current parameters will be returned without being changed.

The modulus of the RNG cannot be changed; it is fixed at 2^{32}.

SEE ALSO **rndcon**, **rndmult**, **rndseed**, **rndn**, **rndu**

Case 30: Base Year Toggle

PURPOSE Specifies whether year value returned by **date** is to include base year (1900) or not.

FORMAT *oldstate* = **sysstate(30,***state***);**

INPUT *state* scalar, 1, 0, or missing value.

OUTPUT *oldstate* scalar, the original state.

REMARKS Internally, **date** acquires the number of years since 1900. **sysstate** case 30 specifies whether **date** should add the base year to that value or not. If *state* = 1, **date** adds 1900, returning a fully-qualified 4-digit year.

If *state* = 0, **date** returns the number of years since 1900. If *state* is a missing value, the current state is returned. The default state is 1.

Case 32: Global LU Tolerance

PURPOSE Gets or sets global singularity tolerance for LU decomposition.

FORMAT *oldtol* = **sysstate(32,***tol***);**

INPUT *tol* scalar, new tolerance.

OUTPUT *oldtol* scalar, the original tolerance.

REMARKS The tolerance must be ≥ 0. If *tol* is negative, the tolerance is returned and left unchanged.

This is a global tolerance and therefore not thread-safe. To set the singularity tolerance for LU decomposition in the current thread, use case 13.

SEE ALSO **croutp**, **inv**

Case 33: Global Cholesky Tolerance

PURPOSE Gets or sets global singularity tolerance for Cholesky decomposition.

FORMAT *oldtol* = **sysstate(33,***tol***);**

INPUT *tol* scalar, new tolerance.

OUTPUT *oldtol* scalar, the original tolerance.

REMARKS The tolerance must be ≥ 0. If *tol* is negative, the tolerance is returned and left unchanged.

This is a global tolerance and therefore not thread-safe. To set the singularity tolerance for Cholesky decomposition in the current thread, use case 14.

SEE ALSO **chol**, **invpd**, **solpd**

Case 34: Global Imaginary Tolerance

PURPOSE Gets or sets the global imaginary tolerance.

FORMAT *oldtol* = **sysstate(34,***tol***);**

INPUT *tol* scalar, the new tolerance.

OUTPUT *oldtol* scalar, the original tolerance.

REMARKS The imaginary tolerance is used to test whether the imaginary part of a complex matrix can be treated as zero or not. Functions that are not defined for complex matrices check the imaginary part to see if it can be ignored. The default tolerance is 2.23e-16, or machine epsilon.

If *tol*<0, the current tolerance is returned.

This is a global tolerance and therefore not thread-safe. To set the imaginary tolerance in the current thread, use case 21.

SEE ALSO **hasimag**

system

PURPOSE Quits **GAUSS** and returns to the operating system.

FORMAT **system;**

system *c*;

INPUT *c* scalar, an optional exit code that can be recovered by the program that invoked **GAUSS**. The default is 0. Valid arguments are 0-255.

REMARKS The **system** command always returns an exit code to the operating system or invoking program. If you don't supply one, it returns 0. This is usually interpreted as indicating success.

SEE ALSO **exec**

tab

PURPOSE Tabs the cursor to a specified text column.

FORMAT **tab(***col***);**

print expr1 expr2 tab(*col1***) expr3 tab(***col2***) expr4 ...;**

INPUT *col* scalar, the column position to tab to.

REMARKS *col* specifies an absolute column position. If *col* is not an integer, it will be truncated.

tab can be called alone or embedded in a **print** statement. You cannot embed it within a parenthesized expression in a **print** statement, though. For example:

```
print (tab(20) c + d * e);
```

will not give the results you expect. If you have to use parenthesized expressions, write it like this instead:

```
print tab(20) (c + d * e);
```

tan

PURPOSE Returns the tangent of its argument.

FORMAT *y* **= tan(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array.

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

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

EXAMPLE
```
let x = 0 .5 1 1.5;
y = tan(x);
```

```
y =  0.00000000
     0.54630249
     1.55740772
    14.10141995
```

SEE ALSO **atan**, **pi**

tanh

PURPOSE Computes the hyperbolic tangent.

FORMAT *y* **= tanh(***x***);**

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array 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);
```

```
x =  −1.570796
     −0.785398
      0.000000
      0.785398
      1.570796
      3.141593
```

```
y =  −0.917152
     −0.655794
      0.000000
      0.655794
      0.917152
      0.996272
```

SOURCE trig.src

tempname

PURPOSE Creates a temporary file with a unique name.

FORMAT *tname* = **tempname**(*path*,*pre*,*suf*);

INPUT

path	string, path where the file will reside.
pre	string, a prefix to begin the file name with.
suf	string, a suffix to end the file name with.

OUTPUT

tname	string, unique temporary file name of the form *path*/*pre*XXXXnnnnn*suf*, where XXXX are 4 letters, and nnnnn is the process id of the calling process.

REMARKS Any or all of the inputs may be a null string or 0. If *path* is not specified, the current working directory is used.

If unable to create a unique file name of the form requested, **tempname** returns a null string.

WARNING: **GAUSS** does not remove temporary files created by **tempname**. It is left to the user to remove them when they are no longer needed.

time

PURPOSE Returns the current system time.

FORMAT *y* **= time;**

OUTPUT *y* 4×1 numeric vector, the current time in the order: hours, minutes, seconds, and hundredths of a second.

EXAMPLE
```
print time;

 7.000000
31.000000
46.000000
33.000000
```

SEE ALSO **date**, **datestr**, **datestring**, **datestrymd**, **hsec**, **timestr**

timedt

PURPOSE Returns system date and time in DT scalar format.

FORMAT *dt* **= timedt;**

OUTPUT *dt* scalar, system date and time in DT scalar format.

REMARKS The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number

20050306071511

represents 07:15:11 or 7:15:11 AM on March 6, 2005.

SOURCE time.src

SEE ALSO **todaydt**, **timeutc**, **dtdate**

timestr

PURPOSE Formats a time in a vector to a string.

FORMAT *ts* = **timestr**(*t*);

INPUT *t* 4×1 vector from the **time** function, or a zero. If the input is 0, the **time** function will be called to return the current system time.

OUTPUT *ts* 8 character string containing current time in the format: **hr:mn:sc**

EXAMPLE
```
t = { 7, 31, 46, 33 };
ts = timestr(t);
print ts;
```

produces:

```
 7:31:46
```

SOURCE time.src

SEE ALSO **date**, **datestr**, **datestring**, **datestrymd**, **ethsec**, **etstr**, **time**

timeutc

PURPOSE Returns the number of seconds since January 1, 1970 Greenwich Mean Time.

FORMAT *tc* **= timeutc;**

OUTPUT *tc* scalar, number of seconds since January 1, 1970 Greenwich Mean Time.

EXAMPLE
```
tc = timeutc;
utv = utctodtv(tc);
```

```
tc = 1125511090
```

```
utv =  2005   8   31   10   58   10   3   242
```

SEE ALSO **dtvnormal**, **utctodtv**

title

PURPOSE Sets the title for the graph.

LIBRARY pgraph

FORMAT **title(***str***);**

INPUT *str* string, the title to display above the graph.

REMARKS Up to three lines of title may be produced by embedding a line feed character ("\L") in the title string.

EXAMPLE
```
title("First title line\LSecond title line\L"\
        "Third title line");
```

Fonts may be specified in the title string. For instructions on using fonts. see SELECTING FONTS, Section 24.4.1.

SOURCE pgraph.src

SEE ALSO **xlabel**, **ylabel**, **fonts**

tkf2eps

PURPOSE Converts a .tkf file to an Encapsulated PostScript file.

LIBRARY pgraph

FORMAT *ret* = **tkf2eps**(*tekfile*,*epsfile*);

INPUT
tekfile string, name of .tkf file.
epsfile string, name of Encapsulated PostScript file.

OUTPUT
ret scalar, 0 if successful

REMARKS The conversion is done using the global parameters in peps.dec. You can modify these globally by editing the .dec file, or locally by setting them in your program before calling **tkf2eps**.

See the header of the output Encapsulated PostScript file and a PostScript manual if you want to modify these parameters.

tkf2ps

PURPOSE Converts a `.tkf` file to a PostScript file.

LIBRARY `pgraph`

FORMAT *ret* = **tkf2ps**(*tekfile*,*psfile*);

INPUT
tekfile string, name of `.tkf` file.
epsfile string, name of Encapsulated PostScript file.

OUTPUT
ret scalar, 0 if successful.

REMARKS The conversion is done using the global parameters in `peps.dec`. You can modify these globally by editing the `.dec` file, or locally by setting them in your program before calling **tkf2ps**.

See the header of the output Encapsulated PostScript file and a PostScript manual if you want to modify these parameters.

tocart

PURPOSE Converts from polar to cartesian coordinates.

FORMAT *xy* = **tocart**(*r*,*theta*);

INPUT
r N×K real matrix, radius.
theta L×M real matrix, E×E conformable with *r*, angle in radians.

OUTPUT
xy max(N,L) by max(K,M) complex matrix containing the *X* coordinate in the real part and the *Y* coordinate in the imaginary part.

SOURCE coord.src

todaydt

PURPOSE Returns system date in DT scalar format. The time returned is always midnight (00:00:00), the beginning of the returned day.

FORMAT *dt* = **todaydt;**

OUTPUT *dt* scalar, system date in DT scalar format.

REMARKS The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number

20050306130525

represents 13:05:25 or 1:05:25 PM on March 6, 2005.

SOURCE time.src

SEE ALSO **timedt**, **timeutc**, **dtdate**

toeplitz

PURPOSE Creates a Toeplitz matrix from a column vector.

FORMAT *t* = **toeplitz**(*x*)**;**

INPUT *x* K×1 vector.

OUTPUT *t* K×K 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: 15.00
```

SOURCE token.src

SEE ALSO **parse**

topolar

PURPOSE Converts from cartesian to polar coordinates.

FORMAT { *r*,*theta* } = **topolar**(*xy*);

INPUT *xy* N×K complex matrix containing the *X* coordinate in the real part and the *Y* coordinate in the imaginary part.

OUTPUT *r* N×K real matrix, radius.

theta N×K real matrix, angle in radians.

SOURCE `coord.src`

trace

PURPOSE Allows the user to trace program execution for debugging purposes.

FORMAT **`trace`** *new*`;`

`trace` *new*`,` *mask*`;`

INPUT *new* scalar, new value for trace flag.

mask scalar, optional mask to allow leaving some bits of the trace flag unchanged.

REMARKS The **`trace`** command has no effect unless you are running your program under **GAUSS**'s source level debugger. Setting the **`trace`** flag will not generate any debugging output during normal execution of a program.

The argument is converted to a binary integer with the following meanings:

bit	decimal	meaning
ones	1	trace calls/returns
twos	2	trace line numbers
fours	4	unused
eights	8	output to window
sixteens	16	output to print
thirty-twos	32	output to auxiliary output
sixty-fours	64	output to error log

You must set one or more of the output bits to get any output from **trace**. If you set **trace** to 2, you'll be doing a line number trace of your program, but the output will not be displayed anywhere.

The **trace** output as a program executes will be as follows:

(+GRAD)	calling function or procedure **GRAD**
(-GRAD)	returning from **GRAD**
[47]	executing line 47

Note that the line number trace will only produce output if the program was compiled with line number records.

To set a single bit use two arguments:

trace 16,16;	turn on output to printer
trace 0,16;	turn off output to printer

EXAMPLE

```
trace 1+8;    /* trace fn/proc calls/returns to standard
              ** output
              */
trace 2+8;    /* trace line numbers to standard output */
trace 1+2+8;  /* trace line numbers and fn/proc
              ** calls/returns to standard output
              */
trace 1+16;   /* trace fn/proc calls/returns to printer */
trace 2+16;   /* trace line numbers to printer */
trace 1+2+16; /* trace line numbers and fn/proc
              ** calls/returns to printer
              */
```

SEE ALSO **#lineson**

trap

PURPOSE Sets the trap flag to enable or disable trapping of numerical errors.

FORMAT **trap** *new*;

trap *new*, *mask*;

INPUT *new* scalar, new trap value.

mask scalar, optional mask to allow leaving some bits of the trap flag unchanged.

REMARKS The trap flag is examined by some functions to control error handling. There are 16 bits in the trap flag, but most **GAUSS** functions will examine only the lowest order bit:

trap 1;	turn trapping on
trap 0;	turn trapping off

If we extend the use of the trap flag, we will use the lower order bits of the trap flag. It would be wise for you to use the highest 8 bits of the trap flag if you create some sort of user-defined trap mechanism for use in your programs. (See the function **trapchk** for detailed instructions on testing the state of the trap flag; see **error** for generating user-defined error codes.)

To set only one bit and leave the others unchanged, use two arguments:

trap 1,1;	set the ones bit
trap 0,1;	clear the ones bit

EXAMPLE
```
proc(0) = printinv(x);
    local oldval,y;
    oldval = trapchk(1);
    trap 1,1;
    y = inv(x);
    trap oldval,1;
    if scalerr(y);
        errorlog "WARNING: x is singular";
    else;
```

```
        print "y" y;
    endif;
endp;
```

In this example the result of **inv** is trapped in case **x** is singular. The trap state is reset to the original value after the call to **inv**.

Calling **printinv** as follows:

```
x = eye(3);
printinv(x);
```

produces:

```
y =
       1.0000000       0.0000000       0.0000000
       0.0000000       1.0000000       0.0000000
       0.0000000       0.0000000       1.0000000
```

while

```
x = ones(3,3);
printinv(x);
```

produces:

```
WARNING: x is singular
```

SEE ALSO **scalerr**, **trapchk**, **error**

trapchk

PURPOSE Tests the value of the trap flag.

FORMAT *y* = **trapchk**(*m*);

INPUT *m* scalar mask value.

OUTPUT *y* scalar which is the result of the bitwise logical AND of the trap flag and the mask value.

REMARKS To check the various bits in the trap flag, add the decimal values for the bits you wish to check according to the chart below and pass the sum in as the argument to the **trapchk** function:

bit	decimal value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768

If you want to test if either bit 0 or bit 8 is set, then pass an argument of 1+256 or 257 to **trapchk**. The following table demonstrates values that will be returned for:

```
y=trapchk(257);
```

	0	1	value of bit 0 in trap flag
0	0	1	
1	256	257	
value of bit 8 in trap flag			

GAUSS functions that test the trap flag currently test only bits 0 and 1.

SEE ALSO **scalerr**, **trap**, **error**

trigamma

PURPOSE Computes trigamma function.

FORMAT *y* = **trigamma**(*x*);

INPUT *x* M×N matrix or N-dimensional array.

OUTPUT *y* M×N matrix or N-dimensional array, trigamma.

REMARKS The trigamma function is the second derivative of the log of the gamma function with respect to its argument.

trimr

PURPOSE Trims rows from the top and/or bottom of a matrix.

FORMAT *y* **= trimr(***x*,*t*,*b***);**

INPUT

x	N×K 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	R×K 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

          0.93077511   0.06961078   0.04207563
y =       0.53640701   0.06640062   0.07222560
```

SEE ALSO **submat**, **rotater**, **shiftr**

trunc

PURPOSE Converts numbers to integers by truncating the fractional portion.

FORMAT *y* = **trunc**(*x*);

INPUT *x* N×K matrix or N-dimensional array.

OUTPUT *y* N×K matrix or N-dimensional array containing the truncated elements of *x*.

EXAMPLE
```
x = 100*rndn(2,2);
y = trunc(x);
```

```
x =  77.68  -14.10
      4.73  -158.88

y =  77.00  -14.00
      4.00  -158.00
```

SEE ALSO **ceil**, **floor**, **round**

type

PURPOSE Returns the symbol table type of its argument.

FORMAT *t* = **type**(*x*);

INPUT *x* local or global symbol, can be an expression.

OUTPUT *t* scalar, argument type.

6 matrix
13 string
15 string array
17 structure
21 array
23 structure pointer
23 sparse matrix

REMARKS **type** returns the type of a single symbol. The related function **typecv** will take a character vector of symbol names and return a vector of either their types or the missing value code for any that are undefined. **type** works for the symbol types listed above; **typecv** works for user-defined procedures, keywords and functions as well. **type** works for global or local symbols; **typecv** works only for global symbols.

EXAMPLE
```
k = { "CHARS" };
print k;
if type(k) =\,= 6;
    k = "" $+ k;  /* force matrix to string */
endif;
print k;
```

produces:

```
            +DEN
CHARS
```

SEE ALSO **typecv**, **typef**

PURPOSE Returns the symbol table type of objects whose names are given as a string or as elements of a character vector or string array.

FORMAT *y* **= typecv(***x***);**

INPUT *x* string, or N×1 character vector or string array which contains the names of variables whose type is to be determined.

OUTPUT *y* scalar or N×1 vector containing the types of the respective symbols in *x*.

REMARKS The values returned by **typecv** for the various variable types are as follows:

5	keyword (**keyword**)
6	matrix (numeric, character, or mixed)
8	procedure (**proc**)
9	function (**fn**)
13	string
15	string array
17	structure
21	array
23	structure pointer

typecv will return the **GAUSS** missing value code if the symbol is not found, so it may be used to determine if a symbol is defined or not.

EXAMPLE
```
xvar = sqrt(5);
yvar = "Montana";
fn area(r) = pi*r*r;
let names = xvar yvar area;
y = typecv(names);
```

```
          XVAR
names =   YVAR
          AREA
```

```
      6
y =  13
      9
```

SEE ALSO **type**, **typef**, **varput**, **varget**

typef

PURPOSE Returns the type of data (the number of bytes per element) in a **GAUSS** data set.

FORMAT *y* **= typef(***fp***);**

INPUT *fp* scalar, file handle of an open file.

OUTPUT *y* scalar, type of data in **GAUSS** data set.

REMARKS If *fp* is a valid **GAUSS** file handle, then *y* will be set to the type of the data in the file as follows:

2	2-byte signed integer
4	4-byte IEEE floating point
8	8-byte IEEE floating point

EXAMPLE
```
infile = "dat1";
outfile = "dat2";
open fin = ^infile;
names = getname(infile);
create fout = ^outfile with ^names,0,typef(fin);
```

In this example, a file `dat2.dat` is created which has the same variables and variable type as the input file, `dat1.dat`. **typef** is used to return the type of the input file data for the **create** statement.

SEE ALSO **colsf**, **rowsf**

union

PURPOSE Returns the union of two vectors with duplicates removed.

FORMAT *y* **= union(***v1*,*v2*,*flag***);**

INPUT

v1	N×1 vector.
v2	M×1 vector.
flag	scalar, 1 if numeric data, 0 if character.

OUTPUT

y	L×1 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
```

unionsa

PURPOSE Returns the union of two string vectors with duplicates removed.

FORMAT *y* = **unionsa(***sv1*,*sv2***);**

INPUT
sv1 N×1 or 1×N string vector.
sv2 M×1 or 1×M string vector.

OUTPUT
y L×1 vector containing all unique values that are in *sv1* and *sv2*, sorted in ascending order.

EXAMPLE
```
string sv1 = { "mary", "jane", "linda", "john" };
string sv2 = { "mary", "sally" };
y = unionsa(sv1,sv2);
```

```
      jane
      john
y =   linda
      mary
      sally
```

SOURCE unionsa.src

SEE ALSO **union**

uniqindx

PURPOSE Computes the sorted index of *x*, leaving out duplicate elements.

FORMAT *index* = **uniqindx(***x*,*flag***);**

INPUT
x N×1 or 1×N vector.
flag scalar, 1 if numeric data, 0 if character.

OUTPUT *index* M×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
```

SEE ALSO **unique**, **uniqindxsa**

uniqindxsa

PURPOSE Computes the sorted index of a string vector, omitting duplicate elements.

FORMAT *ind* **= uniqindxsa(***sv***);**

INPUT *sv* N×1 or 1×N string vector.

OUTPUT *ind* M×1 vector, indices corresponding to the elements of *sv* sorted in ascending order with duplicates removed.

REMARKS Among sets of duplicates it is unpredictable which elements will be indexed.

EXAMPLE
```
string sv = {"mary","linda","linda","jane",
             "jane","cindy","betty"};
ind = uniqindxsa(sv);
y = sv[ind];
```

```
      7
      6
ind = 5
      2
      1
```

```
      betty
      cindy
y =   jane
      linda
      mary
```

SOURCE uniquesa.src

SEE ALSO **unique**, **uniquesa**, **uniqindx**

unique

PURPOSE Sorts and removes duplicate elements from a vector.

FORMAT *y* **= unique(***x***,***flag***);**

INPUT *x* N×1 or 1×N vector.

flag scalar, 1 if numeric data, 0 if character.

OUTPUT *y* M×1 vector, sorted *x* with the duplicates removed.

EXAMPLE
```
let x = 5 4 4 3 3 2 1;
y = unique(x,1);
```

```
      1
      2
y =   3
      4
      5
```

SEE ALSO **uniquesa**, **uniqindx**

uniquesa

PURPOSE Removes duplicate elements from a string vector.

FORMAT *y* **= uniquesa(***sv***);**

INPUT *sv* N×1 or 1×N string vector.

OUTPUT *y* sorted M×1 string vector containing all unique elements found in *sv*.

EXAMPLE
```
string sv1 = { "mary", "jane", "mary", "linda", "john", "jane" };
y = uniquesa(sv);
```

```
y =
jane
john
linda
mary
```

SOURCE uniquesa.src

SEE ALSO **unique**, **uniqindxsa**, **uniqindx**

upmat, upmat1

PURPOSE Returns the upper portion of a matrix. **upmat** returns the main diagonal and every element above. **upmat1** is the same except it replaces the main diagonal with ones.

FORMAT *u* **= upmat(***x***);**

u **= upmat1(***x***);**

INPUT *x* N×K matrix.

OUTPUT *u* N×K matrix containing the upper elements of *x*. The lower elements are replaced with zeros. **upmat** returns the main diagonal intact. **upmat1** replaces the main diagonal with ones.

EXAMPLE

```
x = { 1  2 -1,
      2  3 -2,
      1 -2  1 };

u = upmat(x);
u1 = upmat1(x);
```

The resulting matrices are

$$u = \begin{matrix} 1 & 2 & -1 \\ 0 & 3 & -2 \\ 0 & 0 & 1 \end{matrix}$$

$$u1 = \begin{matrix} 1 & 2 & -1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{matrix}$$

SOURCE diag.src

SEE ALSO **lowmat**, **lowmat1**, **diag**, **diagrv**, **crout**

upper

PURPOSE Converts a string, matrix of character data, or string array to uppercase.

FORMAT *y* **= upper(***x***);**

INPUT *x* string, or N×K matrix, or string array containing the character data to be converted to uppercase.

OUTPUT *y* string, or N×K matrix, or string array containing the uppercase equivalent of the data in *x*.

REMARKS If *x* is a numeric matrix, *y* will contain garbage. No error message will be generated since **GAUSS** does not distinguish between numeric and character data in matrices.

EXAMPLE
```
x = "uppercase";
```

```
y = upper(x);
```

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

The **use** command can be used at the top of files that are to be compiled with the **compile** command. This can greatly shorten compile time for a set of closely related programs. For example:

```
library pgraph;
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 programs could be compiled very quickly with the following statement at the top of each:

```
use pgraph;
```

or the same statement could be executed once, for instance from the command prompt, to instantly load all the procedures for PQG.

When the compiled file is loaded with **use**, all previous symbols and procedures are deleted before the program is loaded. It is therefore unnecessary to execute a **new** before **use**'ing a compiled file.

use can appear only ONCE at the TOP of a program.

SEE ALSO **compile**, **run**, **saveall**

utctodt

PURPOSE Converts UTC scalar format to DT scalar format.

FORMAT *dt* = **utctodt**(*utc*);

INPUT *utc* N×1 vector, UTC scalar format.

OUTPUT *dt* N×1 vector, DT scalar format.

REMARKS A UTC scalar gives the number of seconds since or before January 1, 1970 Greenwich Mean Time. In DT scalar format, 08:35:52 on June 11, 2005 is 20050611083552.

EXAMPLE
```
tc = 1126290409;
print "tc = " tc;
dt = utctodt(tc);
print "dt = " dt;
```

produces:

```
tc =  1126290409
dt =  20050909112649
```

SOURCE time.src

SEE ALSO **dtvnormal**, **timeutc**, **utctodtv**, **dttodtv**, **dtvtodt**, **dttoutc**, **dtvtodt**, **strtodt**, **dttostr**

utctodtv

PURPOSE Converts UTC scalar format to DTV vector format.

FORMAT *dtv* = **utctodtv**(*utc*);

INPUT *utc* N×1 vector, UTC scalar format.

OUTPUT *dtv* N×8 matrix, DTV vector format.

REMARKS A UTC scalar gives the number of seconds since or before January 1, 1970 Greenwich Mean Time.

Each row of *dtv*, in DTV vector format, contains:

[N,1] Year, four digit integer.
[N,2] Month in Year, 1-12.
[N,3] Day of month, 1-31.
[N,4] Hours since midnight, 0-23.
[N,5] Minutes, 0-59.
[N,6] Seconds, 0-59.
[N,7] Day of week, 0-6, 0=Sunday.
[N,8] Days since Jan 1 of current year, 0-365.

EXAMPLE
```
tc = timeutc;
print "tc = " tc;
dtv = utctodtv(tc);
print "dtv = " dtv;
```

produces:

```
tc =  1126290724
dtv =  2005  9    9    11   32   4    5    251
```

SEE ALSO **dtvnormal**, **timeutc**, **utctodt**, **dttodtv**, **dttoutc**, **dtvtodt**, **dtvtoutc**, **strtodt**, **dttostr**

utrisol

PURPOSE Computes the solution of $Ux = b$ where U is an upper triangular matrix.

FORMAT x = **utrisol**(b,U);

INPUT
b P×K matrix.
U P×P upper triangular matrix.

OUTPUT
x P×K matrix, solution of $Ux = b$.

REMARKS **utrisol** applies a back solve to $Ux = b$ to solve for x. If b has more than one column, each column is solved for separately, i.e., **utrisol** applies a back solve to $Ux[., i] = b[., i]$.

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 N×1 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 keyboard input. When **key** returns a nonzero value, meaning a key has been pressed, the ASCII value it returns is tested to see if it is an uppercase or lowercase 'Y'. If it is, the program will jump to the label **doit**, otherwise the program will end.

SEE ALSO **chrs**, **ftos**, **stof**

varget

PURPOSE Accesses a global variable whose name is given as a string argument.

FORMAT *y* **= varget(***s***);**

INPUT *s* string containing the name of the global symbol you wish to access.

OUTPUT *y* contents of the variable whose name is in *s*.

REMARKS This function searches the global symbol table for the symbol whose name is in *s* and returns the contents of the variable if it exists. If the symbol does not exist, the function will terminate with an **Undefined symbol** error message. If you want to check to see if a variable exists before using this function, use **typecv**.

EXAMPLE
```
dog = rndn(2,2);
y = varget("dog");
```

```
dog =  -0.83429985   0.34782433
        0.91032546   1.75446391
```

```
y =  −0.83429985   0.34782433
      0.91032546   1.75446391
```

SEE ALSO **typecv**, **varput**

vargetl

PURPOSE Accesses a local variable whose name is given as a string argument.

FORMAT *y* **= vargetl(***s***);**

INPUT *s* string containing the name of the local symbol you wish to access.

OUTPUT *y* contents of the variable whose name is in *s*.

REMARKS This function searches the local symbol list for the symbol whose name is in *s* and returns the contents of the variable if it exists. If the symbol does not exist, the function will terminate with an **Undefined symbol** error message.

EXAMPLE

```
proc 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;

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 **varputl**

varmall

PURPOSE Computes log-likelihood of a Vector ARMA model.

FORMAT *ll* = **varmall**(*w*,*phi*,*theta*,*vc*);

INPUT

w	N×K matrix, time series.
phi	(K*P)×K matrix, AR coefficient matrices.
theta	(K*Q)×K matrix, MA coefficient matrices.
vc	K×K matrix, covariance matrix.

OUTPUT

ll	scalar, log-likelihood. If the calculation fails *ll* is set to missing value with error code:

Error Code	Reason for Failure
1	M < 1
2	N < 1
3	P < 0
4	Q < 0
5	P = 0 and Q = 0
7	floating point work space too small
8	integer work space too small
9	vc is not positive definite
10	AR parameters too close to stationarity boundary
11	model not stationary
12	model not invertible
13	I+M'H'HM not positive definite

REMARKS **varmall** is adapted from code developed by Jose Alberto Mauricio of the Universidad Complutense de Madrid. It was published as Algorithm AS311 in Applied Statistics. Also described in "Exact Maximum Likelihood Estimation of Stationary Vector ARMA Models," JASA, 90:282-264.

varmares

PURPOSE Computes residuals of a Vector ARMA model.

FORMAT *res* = **varmares**(*w*,*phi*,*theta*);

INPUT

w	N×K matrix, time series.
phi	(K*P)×K matrix, AR coefficient matrices.
theta	(K*Q)×K matrix, MA coefficient matrices.

OUTPUT

res	N×K matrix, residuals. If the calculation fails *res* is set to missing value with error code:

Error Code	Reason for Failure
1	M < 1
2	N < 1
3	P < 0
4	Q < 0
5	P = 0 and Q = 0
7	floating point work space too small
8	integer work space too small
10	AR parameters too close to stationarity boundary
11	model not stationary
12	model not invertible
13	I+M'H'HM not positive definite

REMARKS **varmares** is adapted from code developed by Jose Alberto Mauricio of the Universidad Complutense de Madrid. It was published as Algorithm AS311 in Applied Statistics. Also described in "Exact Maximum Likelihood Estimation of Stationary Vector ARMA Models," JASA, 90:282-264.

varput

PURPOSE Allows a matrix, array, string, or string array to be assigned to a global symbol whose name is given as a string argument.

FORMAT *y* = **varput**(*x*,*n*);

INPUT

x matrix, array, string, or string array which is to be assigned to the target variable.

n string containing the name of the global symbol which will be the target variable.

OUTPUT

y scalar, 1 if the operation is successful and 0 if the operation fails.

REMARKS *x* and *n* may be global or local. The variable, whose name is in *n*, that *x* is assigned to is always a global.

If the function fails, it will be because the global symbol table is full.

This function is useful for returning values generated in local variables within a procedure to the global symbol table.

EXAMPLE

```
source = rndn(2,2);
targname = "target";
if 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**

varputl

PURPOSE Allows a matrix, array, string, or string array to be assigned to a local symbol whose name is given as a string argument.

FORMAT *y* **= varputl(***x*,*n***);**

INPUT

x	matrix, array, string, or string array which is to be assigned to the target variable.
n	string containing the name of the local symbol which will be the target variable.

OUTPUT *y* scalar, 1 if the operation is successful and 0 if the operation fails.

REMARKS *x* and *n* may be global or local. The variable, whose name is in *n*, that *x* is assigned to is always a local.

EXAMPLE

```
proc 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**

vartypef

PURPOSE Returns a vector of ones and zeros that indicate whether variables in a data set are character or numeric.

FORMAT *y* = **vartypef**(*f*);

INPUT *f* file handle of an open file.

OUTPUT *y* N×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.

vcm, vcx

PURPOSE Computes a variance-covariance matrix.

FORMAT *vc* = **vcm**(*m*);

vc = **vcx**(*x*);

INPUT *m* K×K moment ($x'x$) matrix. A constant term MUST have been the first variable when the moment matrix was computed.

	x	N×K matrix of data.
OUTPUT	*vc*	K×K variance-covariance matrix.

SOURCE corr.src

SEE ALSO **momentd**

vcms, vcxs

PURPOSE Computes a sample variance-covariance matrix.

FORMAT *vc* = **vcms**(*m*);

vc = **vcxs**(*x*);

INPUT	*m*	K×K moment ($x'x$) matrix. A constant term MUST have been the first variable when the moment matrix was computed.
	x	N×K matrix of data.
OUTPUT	*vc*	K×K variance-covariance matrix.

REMARKS Computes sample covariance matrix, that is, it divides the sample size, **N**, rather than **N** - 1. For population covariance matrix which uses **N** - 1 rather than **N** see **vcm** or **vcx**.

SOURCE corrs.src

SEE ALSO **momentd**, **corrms**, **corrvcs**, **corrxs**

vec, vecr

PURPOSE Creates a column vector by appending the columns/rows of a matrix to each other.

FORMAT *yc* = **vec**(*x*);

yr = **vecr**(*x*);

INPUT *x* N×K matrix.

OUTPUT *yc* (N*K)×1 vector, the columns of *x* appended to each other.

yr (N*K)×1 vector, the rows of *x* appended to each other and the result transposed.

REMARKS **vecr** is much faster.

EXAMPLE
```
x = { 1 2,
      3 4 };
yc = vec(x);
yr = vecr(x);
```

```
x =  1.000000  2.000000
     3.000000  4.000000
```

```
yc =  1.000000
      3.000000
      2.000000
      4.000000
```

```
yr =  1.000000
      2.000000
      3.000000
      4.000000
```

vech

PURPOSE Vectorizes a symmetric matrix by retaining only the lower triangular portion of the matrix.

FORMAT *v* **= vech(***x***);**

INPUT *x* N×N symmetric matrix.

OUTPUT *v* (N*(N+1)/2)×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 trangular portion of the matrix into a column vector in row-wise order.

EXAMPLE
```
x = seqa(10,10,3) + seqa(1,1,3)';
v = vech(x);
sx = xpnd(v);
```

```
      11  12  13
x =   21  22  23
      31  32  33
```

```
v =  11
     21
     22
     31
     32
     33

sx =  11  21  31
      21  22  32
      31  32  33
```

SEE ALSO **xpnd**

vector (dataloop)

PURPOSE Specifies the creation of a new variable within a data loop.

FORMAT **vector** [[**#**]] *numvar* **=** *numeric_expression*;

vector $ *charvar* **=** *character_expression*;

REMARKS A *numeric_expression* is any valid expression returning a numeric value. A *character_expression* is any valid expression returning a character value. If neither '**$**' nor '**#**' is specified, '**#**' is assumed.

vector is used in place of **make** when the expression returns a scalar rather than a vector. **vector** forces the result of such an expression to a vector of the correct length. **vector** could actually be used anywhere that **make** is used, but would generate slower code for expressions that already return vectors.

Any variables referenced must already exist, either as elements of the source data set, as **extern**'s, or as the result of a previous **make**, **vector**, or **code** statement.

EXAMPLE
```
vector const = 1;
```

SEE ALSO **make (dataloop)**

vget

PURPOSE Extracts a matrix or string from a data buffer constructed with **vput**.

FORMAT { *x*,*dbufnew* } = **vget**(*dbuf*,*name*);

INPUT

dbuf	N×1 vector, a data buffer containing various strings and matrices.
name	string, the name of the string or matrix to extract from *dbuf*.

OUTPUT

x	L×M matrix or string, the item extracted from *dbuf*.
dbufnew	K×1 vector, the remainder of *dbuf* after *x* has been extracted.

SOURCE pack.src

SEE ALSO **vlist**, **vput**, **vread**

view

PURPOSE Sets the position of the observer in workbox units for 3-D plots.

LIBRARY pgraph

FORMAT **view**(*x*,*y*,*z*);

INPUT

x	scalar, the X position in workbox units.

y scalar, the Y position in workbox units.

z scalar, the Z position in workbox units.

REMARKS The size of the workbox is set with **volume**. The viewer MUST be outside of the workbox. The closer the position of the observer, the more perspective distortion there will be. If $x = y = z$, the projection will be isometric.

If **view** is not called, a default position will be calculated.

Use **viewxyz** to locate the observer in plot coordinates.

SOURCE pgraph.src

SEE ALSO **volume**, **viewxyz**

viewxyz

PURPOSE To set the position of the observer in plot coordinates for 3-D plots.

LIBRARY pgraph

FORMAT **viewxyz(***x*,*y*,*z***);**

INPUT *x* scalar, the X position in plot coordinates.

y scalar, the Y position in plot coordinates.

z scalar, the Z position in plot coordinates.

REMARKS The viewer MUST be outside of the workbox. The closer the observer, the more perspective distortion there will be.

If **viewxyz** is not called, a default position will be calculated.

Use **view** to locate the observer in workbox units.

SOURCE pgraph.src

SEE ALSO **volume**, **view**

vlist

PURPOSE Lists the contents of a data buffer constructed with **vput**.

FORMAT **vlist(***dbuf***);**

INPUT *dbuf* N×1 vector, a data buffer containing various strings and matrices.

REMARKS **vlist** lists the names of all the strings and matrices stored in *dbuf*.

SOURCE vpack.src

SEE ALSO **vget**, **vput**, **vread**

vnamecv

PURPOSE Returns the names of the elements of a data buffer constructed with **vput**.

FORMAT *cv* **= vnamecv(***dbuf***);**

INPUT *dbuf* N×1 vector, a data buffer containing various strings and matrices.

OUTPUT *cv* K×1 character vector containing the names of the elements of *dbuf*.

SEE ALSO **vget**, **vput**, **vread**, **vtypecv**

volume

PURPOSE Sets the length, width, and height ratios of the 3-D workbox.

LIBRARY pgraph

FORMAT **volume**(*x*,*y*,*z*);

INPUT
x scalar, the X length of the 3-D workbox.
y scalar, the Y length of the 3-D workbox.
z scalar, the Z length of the 3-D workbox.

REMARKS The ratio between these values is what is important. If **volume** is not called, a default workbox will be calculated.

SOURCE pgraph.src

SEE ALSO **view**

vput

PURPOSE Inserts a matrix or string into a data buffer.

FORMAT *dbufnew* = **vput**(*dbuf*,*x*,*xname*);

INPUT
dbuf N×1 vector, a data buffer containing various strings and matrices. If *dbuf* is a scalar 0, a new data buffer will be created.
x L×M matrix or string, item to be inserted into *dbuf*.
xname string, the name of *x*, will be inserted with *x* into *dbuf*.

OUTPUT *dbufnew* K×1 vector, the data buffer after *x* and *xname* have been inserted.

REMARKS If *dbuf* already contains *x*, the new value of *x* will replace the old one.

SOURCE vpack.src

SEE ALSO **vget**, **vlist**, **vread**

vread

PURPOSE Reads a string or matrix from a data buffer constructed with **vput**.

FORMAT *x* = **vread**(*dbuf*,*xname*);

INPUT *dbuf* N×1 vector, a data buffer containing various strings and matrices.

xname string, the name of the matrix or string to read from *dbuf*.

OUTPUT *x* L×M matrix or string, the item read from *dbuf*.

REMARKS **vread**, unlike **vget**, does not change the contents of *dbuf*. Reading *x* from *dbuf* does not remove it from *dbuf*.

SOURCE vpack.src

SEE ALSO **vget**, **vlist**, **vput**

vtypecv

PURPOSE Returns the types of the elements of a data buffer constructed with **vput**.

FORMAT *cv* = **vtypecv**(*dbuf*);

INPUT *dbuf* N×1 vector, a data buffer containing various strings and matrices.

OUTPUT *cv* K×1 character vector containing the types of the elements of *dbuf*.

SEE ALSO **vget**, **vput**, **vread**, **vnamecv**

wait, waitc

PURPOSE Waits until any key is pressed.

FORMAT **wait**;

waitc;

REMARKS If you are working in terminal mode, these commands do not "see" any keystrokes until ENTER is pressed. **waitc** clears any pending keystrokes before waiting until another key is pressed.

SOURCE wait.src, waitc.src

SEE ALSO **pause**

walkindex

PURPOSE Walks the index of an array forward or backward through a specified dimension.

FORMAT *ni* = **walkindex**(*i*,*o*,*dim*);

INPUT

i M×1 vector of indices into an array, where M<=N.

o N×1 vector of orders of an N-dimensional array.

dim scalar [1-to-M], index into the vector of indices *i*, corresponding to the dimension to walk through, positive to walk the index forward, or negative to walk backward.

OUTPUT

ni M×1 vector of indices, the new index.

REMARKS

walkindex will return a scalar error code if the index cannot walk further in the specified dimension and direction.

EXAMPLE

```
orders = (3,4,5,6,7);
a = arrayinit(orders,1);
ind = { 2,3,3 };
ind = walkindex(ind,orders,-2);
```

```
      2
ind = 2
      3
```

This example decrements the second value of the index vector **ind**.

```
ind = walkindex(ind,orders,3);
```

```
      2
ind = 2
      4
```

Using the **orders** from the example above and the **ind** that was returned, this example increments the third value of the index vector **ind**.

SEE ALSO

nextindex, **previousindex**, **loopnextindex**

window

PURPOSE Partitions the window into tiled regions (graphic panels) of equal size.

LIBRARY pgraph

FORMAT **window(***row*,*col*,*typ*);

INPUT
row scalar, number of rows of graphic panels.
col scalar, number of columns of graphic panels.
typ scalar, graphic panel attribute type. If 1, the graphic panels will be transparent, if 0, the graphic panels will be nontransparent (blanked).

REMARKS The graphic panels will be numbered from 1 to (*row*)×(*col*) starting from the left topmost graphic panel and moving right.

See **makewind** for creating graphic panels of a specific size and position. (For more information, see GRAPHIC PANELS, Section 24.3.

SOURCE pwindow.src

SEE ALSO **endwind**, **begwind**, **setwind**, **nextwind**, **getwind**, **makewind**

writer

PURPOSE Writes a matrix to a **GAUSS** data set.

FORMAT *y* **= writer(***fh*,*x*);

INPUT
fh handle of the file that data is to be written to.

x N×K matrix.

OUTPUT

y scalar specifying the number of rows of data actually written to the data set.

REMARKS

The file must have been opened with **create**, **open for append**, or **open for update**.

The data in *x* will be written to the data set whose handle is *fh* starting at the current pointer position in the file. The pointer position in the file will be updated, so the next call to **writer** will put the next block of data after the first block. (See **open** and **create** for the initial pointer positions in the file for reading and writing.)

x must have the same number of columns as the data set. **colsf** returns the number of columns in a data set.

writer returns the number of rows actually written to the data set. If *y* does not equal **rows(***x***)**, the disk is probably full.

If the data set is not double precision, the data will be rounded as it is written out.

If the data contain character elements, the file must be double precision or the character information will be lost.

If the file being written to is the 2-byte integer data type, then missing values will be written out as -32768. These will not automatically be converted to missings on input. They can be converted with the **miss** function:

```
x = miss(x,-32768);
```

Trying to write complex data to a data set that was originally created to store real data will cause a program to abort with an error message. (See **create** for details on creating a complex data set.)

EXAMPLE

```
create fp = data with x,10,8;
```

```
if fp =\,= -1;
    errorlog "Can't create output file";
    end;
endif;
c = 0;
do until c >= 10000;
    y = rndn(100,10);
    k = writer(fp,y);
    if k /= rows(y);
        errorlog "Disk Full";
        fp = close(fp);
        end;
    endif;
    c = c+k;
endo;
fp = close(fp);
```

In this example, a 10000×10 data set of Normal random numbers is written to a data set called `data.dat`. The variable names are **X01**-**X10**.

SEE ALSO **open**, **close**, **create**, **readr**, **saved**, **seekr**

xlabel

PURPOSE Sets a label for the X axis.

LIBRARY `pgraph`

FORMAT **xlabel(***str***);**

INPUT *str* string, the label for the X axis.

SOURCE `pgraph.src`

SEE ALSO **title**, **ylabel**, **zlabel**

xlsGetSheetCount

PURPOSE Gets the number of sheets in an Excel® spreadsheet.

FORMAT *nsheets* = **xlsGetSheetCount**(*file*);

INPUT *file* string, name of `.xls` file.

OUTPUT *nsheets* scalar, sheet count or an error code.

PORTABILITY **Windows** only

REMARKS If **xlsGetSheetCount** fails, it will return a scalar error code, which can be decoded with **scalerr**.

SEE ALSO **xlsGetSheetSize**, **xlsGetSheetTypes**, **xlsMakeRange**

xlsGetSheetSize

PURPOSE Gets the size (rows and columns) of a specified sheet in an Excel® spreadsheet.

FORMAT { *rows*,*cols* } = **xlsGetSheetSize**(*file*,*sheet*);

INPUT *file* string, name of `.xls` file.
sheet scalar, sheet index (1-based).

OUTPUT *rows* scalar, number of rows.

cols scalar, number of columns.

PORTABILITY **Windows** only

REMARKS If **xlsGetSheetSize** fails, it will return a scalar error code, which can be decoded with **scalerr**.

SEE ALSO **xlsGetSheetCount**, **xlsGetSheetTypes**, **xlsMakeRange**

xlsGetSheetTypes

PURPOSE Gets the cell format types of a row in an Excel® spreadsheet.

FORMAT *nsheets* = **xlsGetSheetTypes**(*file*,*sheet*,*row*);

INPUT *file* string, name of `.xls` file.

sheet scalar, sheet index (1-based).

row scalar, the row of cells to be scanned.

OUTPUT *types* 1×K vector of predefined data types representing the format of each cell in the specified row.

The possible types are:

0 Text

1 Numeric

2 Date

PORTABILITY **Windows** only

REMARKS K is the number of columns found in the spreadsheet.

If **xlsGetSheetTypes** fails, it will return a scalar error code, which can be decoded with **scalerr**.

SEE ALSO **xlsGetSheetCount**, **xlsGetSheetSize**, **xlsMakeRange**

xlsMakeRange

PURPOSE Builds an Excel® range string from a row/column pair.

FORMAT *range* = **xlsMakeRange**(*row*,*col*);

INPUT *row* scalar or 2×1 vector.

col scalar or 2×1 vector.

OUTPUT *range* string, an Excel®-formatted range specifier.

PORTABILITY **Windows** only

REMARKS If *row* is a 2×1 vector, it is interpreted as follows

*row***[1]** starting row

*row***[2]** ending row

If *col* is a 2×1 vector, it is interpreted as follows

*col***[1]** starting column

*col***[2]** ending column

SEE ALSO **xlsGetSheetCount**, **xlsGetSheetSize**, **xlsGetSheetTypes**

xlsreadm

PURPOSE Reads from an Excel® spreadsheet into a **GAUSS** matrix.

FORMAT *mat* **= xlsreadm(***file***,***range***,***sheet***,***vls***);**

INPUT

file string, name of `.xls` file.

range string, range to read, e.g. **a2:b20**, or the starting point of the read, e.g. **a2**.

sheet scalar, sheet number.

vls null string or 9×1 matrix, specifies the conversion of Excel® empty cells and special types into **GAUSS** (see Remarks). A null string results in all empty cells and special types being converted to **GAUSS** missing values.

OUTPUT *mat* matrix or a Microsoft error code.

PORTABILITY **Windows** only

REMARKS If range is a null string, then by default the read will begin at cell **a1**.

The *vls* argument lets users control the import of Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell
1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!
7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of #DIV/0! to 9999.99, and all other empty cells and special types to **GAUSS** missing values:

```
vls = reshape(error(0),9,1);

vls[4] = 9999.99;
```

SEE ALSO **xlsReadSA**, **xlsWrite**, **xlsWriteM**, **xlsWriteSA**, **xlsGetSheetCount**, **xlsGetSheetSize**, **xlsGetSheetTypes**, **xlsMakeRange**

xlsreadsa

PURPOSE Reads from an Excel® spreadsheet into a **GAUSS** string array or string.

FORMAT *s* **= xlsreadsa(***file***,***range***,***sheet***,***vls***);**

INPUT

file string, name of `.xls` file.

range string, range to read, e.g. **a2:b20** or the starting point of the read, e.g. **a2**.

sheet scalar, sheet number.

vls null string or 9×1 string array, specifies the conversion of Excel® empty cells and special types into **GAUSS** (see Remarks). A null string results in all empty cells and special types being converted to null strings.

OUTPUT

s string array or string or a Microsoft error code.

PORTABILITY

Windows only

REMARKS

If *range* is a null string, then by default the read will begin at cell **a1**.

The *vls* argument lets users control the import of Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell
1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!
7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of #DIV/0! to “Division by Zero”, and all other empty cells and special types to null strings:

```
vls = reshape("",9,1);
vls[4] = "Division by Zero";
```

SEE ALSO

xlsReadM, **xlsWrite**, **xlsWriteM**, **xlsWriteSA**, **xlsGetSheetCount**, **xlsGetSheetSize**, **xlsGetSheetTypes**, **xlsMakeRange**

xlsWrite

PURPOSE Writes a **GAUSS** matrix, string, or string array to an Excel® spreadsheet.

FORMAT *ret* = **xlsWrite(***data*,*file*,*range*,*sheet*,*vls*);

INPUT

data	matrix, string, or string array.
file	string, name of `.xls` file.
range	string, the starting point of the write, e.g. **a2**.
sheet	scalar, sheet number.
vls	null string or 9×1 matrix or string array, specifies the conversion of **GAUSS** values or characters into Excel® empty cells and special types (see Remarks). A null string results in all **GAUSS** missing values and null strings being converted to empty cells.

OUTPUT

ret	scalar, 0 if success or a Microsoft error code.

PORTABILITY **Windows** only

REMARKS The *vls* argument lets users control the export to Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell
1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!
7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of 9999.99 to #DIV/0! in Excel® and convert all **GAUSS** missing values to empty cells in Excel®:

```
vls = reshape(error(0),9,1);

vls[4] = 9999.99;
```

SEE ALSO **xlsReadSA**, **xlsReadm**, **xlsWriteM**, **xlsWriteSA**, **xlsGetSheetCount**, **xlsGetSheetSize**, **xlsGetSheetTypes**, **xlsMakeRange**

xlsWritem

PURPOSE Writes a **GAUSS** matrix to an Excel® spreadsheet.

FORMAT *ret* **= xlsWritem(***data***,***file***,***range***,***sheet***,***vls***);**

INPUT

data	matrix.
file	string, name of `.xls` file.
range	string, the starting point of the write, e.g. **a2**.
sheet	scalar, sheet number.
vls	null string or 9×1 matrix, specifies the conversion of **GAUSS** values into Excel® empty cells and special types (see Remarks). A null string results in all **GAUSS** missing values being converted to empty cells.

OUTPUT

ret	scalar, 0 if success or a Microsoft error code.

PORTABILITY **Windows** only

REMARKS The *vls* argument lets users control the export to Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell
1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!
7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of 9999.99 to #DIV/0! in Excel® and convert all **GAUSS** missing values to empty cells in Excel®:

```
vls = reshape(error(0),9,1);

vls[4] = 9999.99;
```

SEE ALSO **xlsReadSA**, **xlsReadm**, **xlsWrite**, **xlsWriteSA**, **xlsGetSheetCount**, **xlsGetSheetSize**, **xlsGetSheetTypes**, **xlsMakeRange**

xlswritesa

PURPOSE Writes a **GAUSS** string or string array to an Excel® spreadsheet.

FORMAT *ret* **= xlsWritesa(***data***,***file***,***range***,***sheet***,***vls***);**

INPUT

data	string or string array.
file	string, name of `.xls` file.
range	string, the starting point of the write, e.g. **a2**.
sheet	scalar, sheet number.

vls null string or 9×1 string array, specifies the conversion of **GAUSS** characters into Excel® empty cells and special types (see Remarks). A null string results in all null strings being converted to empty cells.

OUTPUT

ret scalar, 0 if success or a Microsoft error code.

PORTABILITY

Windows only

REMARKS

The *vls* argument lets users control the export to Excel® empty cells and special types, according to the following table:

Row Number	Excel® Cell
1	empty cell
2	#N/A
3	#VALUE!
4	#DIV/0!
5	#NAME?
6	#REF!
7	#NUM!
8	#NULL!
9	#ERR

Use the following to convert all occurrences of "Division by Zero" to #DIV/0!, and all null strings to empty cells:

```
vls = reshape("",9,1);
vls[4] = "Division by Zero";
```

SEE ALSO

xlsReadM, **xlsWrite**, **xlsWriteM**, **xlsReadSA**, **xlsGetSheetCount**, **xlsGetSheetSize**, **xlsGetSheetTypes**, **xlsMakeRange**

PURPOSE Expands a column vector into a symmetric matrix.

FORMAT *x* **= xpnd(***v***);**

INPUT *v* K×1 vector, to be expanded into a symmetric matrix.

OUTPUT *x* M×M matrix, the results of taking *v* and filling in a symmetric matrix with its elements.

M = ((-1 + sqrt(1+8*K))/2)

REMARKS If *v* does not contain the right number of elements, (that is, if **sqrt(1 + 8*K)** is not integral), then an error message is generated.

This function is particularly useful for hard-coding symmetric matrices, because only about half of the matrix needs to be entered.

EXAMPLE
```
x = { 1,
      2, 3,
      4, 5, 6,
      7, 8, 9, 10 };
y = xpnd(x);
```

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

$$y = \begin{matrix} 1 & 2 & 4 & 7 \\ 2 & 3 & 5 & 8 \\ 4 & 5 & 6 & 9 \\ 7 & 8 & 9 & 10 \end{matrix}$$

SEE ALSO **vech**

xtics

PURPOSE Sets and fixes scaling, axes numbering and tick marks for the X axis.

LIBRARY pgraph

FORMAT **xtics(***min***,***max***,***step***,***minordiv***);**

INPUT

min	scalar, the minimum value.
max	scalar, the maximum value.
step	scalar, the value between major tick marks.
minordiv	scalar, the number of minor subdivisions.

REMARKS This routine fixes the scaling for all subsequent graphs until **graphset** is called.

This gives you direct control over the axes endpoints and tick marks. If **xtics** is called after a call to **scale**, it will override **scale**.

X and Y axes numbering may be reversed for **xy**, **logx**, **logy**, and **loglog** graphs. This may be accomplished by using a negative step value in the **xtics** and **ytics** functions.

SOURCE pscale.src

SEE ALSO **scale**, **ytics**, **ztics**

xy

PURPOSE Graphs X vs. Y using Cartesian coordinates.

LIBRARY pgraph

FORMAT **xy**(*x*,*y*);

INPUT
x N×1 or N×M matrix. Each column contains the X values for a particular line.
y N×1 or N×M matrix. Each column contains the Y values for a particular line.

REMARKS Missing values are ignored when plotting symbols. If missing values are encountered while plotting a curve, the curve will end and a new curve will begin plotting at the next non-missing value.

SOURCE pxy.src

SEE ALSO **xyz**, **logx**, **logy**, **loglog**

xyz

PURPOSE Graphs X vs. Y vs. Z using Cartesian coordinates.

LIBRARY pgraph

FORMAT **xyz**(*x*,*y*,*z*);

INPUT
x N×1 or N×K matrix. Each column contains the X values for a particular line.

y N×1 or N×K matrix. Each column contains the Y values for a particular line.

z N×1 or N×K 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

ylabel

PURPOSE Sets a label for the Y axis.

LIBRARY pgraph

FORMAT **ylabel(***str***);**

INPUT *str* string, the label for the Y axis.

SOURCE pgraph.src

SEE ALSO **title**, **xlabel**, **zlabel**

ytics

PURPOSE Sets and fixes scaling, axes numbering and tick marks for the Y axis.

LIBRARY pgraph

FORMAT **ytics(***min*,*max*,*step*,*minordiv*);

INPUT

min	scalar, the minimum value.
max	scalar, the maximum value.
step	scalar, the value between major tick marks.
minordiv	scalar, the number of minor subdivisions.

REMARKS This routine fixes the scaling for all subsequent graphs until **graphset** is called.

This gives you direct control over the axes endpoints and tick marks. If **ytics** is called after a call to **scale**, it will override **scale**.

X and Y axes numbering may be reversed for **xy**, **logx**, **logy** and **loglog** graphs. This may be accomplished by using a negative step value in the **xtics** and **ytics** functions.

SOURCE pscale.src

SEE ALSO **scale**, **xtics**, **ztics**

zeros

PURPOSE Creates a matrix of zeros.

FORMAT *y* = **zeros(***r*,*c*);

INPUT

r	scalar, the number of rows.
c	scalar, the number of columns.

OUTPUT

y	$r \times c$ matrix of zeros.

REMARKS This is faster than **ones**.

Noninteger arguments will be truncated to an integer.

EXAMPLE

```
y = zeros(3,2);
```

```
      0.000000   0.000000
y =   0.000000   0.000000
      0.000000   0.000000
```

SEE ALSO **ones**, **eye**

zlabel

PURPOSE Sets a label for the Z axis.

LIBRARY pgraph

FORMAT **zlabel(***str***);**

INPUT *str* string, the label for the Z axis.

SOURCE pgraph.src

SEE ALSO **title**, **xlabel**, **ylabel**

ztics

PURPOSE Sets and fixes scaling, axes numbering and tick marks for the Z axis.

LIBRARY pgraph

FORMAT **ztics**(*min*,*max*,*step*,*minordiv*);

INPUT

min	scalar, the minimum value.
max	scalar, the maximum value.
step	scalar, the value between major tick marks.
minordiv	scalar, the number of minor subdivisions. If this function is used with **contour**, contour labels will be placed every *minordiv* levels. If 0, there will be no labels.

REMARKS This routine fixes the scaling for all subsequent graphs until **graphset** is called.

This gives you direct control over the axes endpoints and tick marks. If **ztics** is called after a call to **scale3d**, it will override **scale3d**.

SOURCE pscale.src

SEE ALSO **scale3d**, **xtics**, **ytics**, **contour**

Obsolete Commands D

The following commands will no longer be supported and therefore should not be used when creating new programs.

color

coreleft

csrtype

denseSubmat

dfree

disable

editm

eigcg

eigcg2

eigch

eigch2

eigrg

eigrg2

eigrs

eigrs2

enable

export

exportf

files

font

FontLoad

FontUnload

FontUnloadAll

graph

import

importf

isSparse

line

lpos

lprint

lprint on/off

lpwidth

lshow

medit

nametype

ndpchk

ndpclex

ndpcntrl

plot

plotsym

prcsn

print on/off

rndns

rndus

scroll

setvmode

sparseCols

sparseEye

sparseFD

sparseFP

sparseHConcat

sparseNZE

sparseOnes

sparseRows

sparseScale

sparseSet

sparseSolve

sparseSubmat

sparseTD

sparseTranspose

sparseTrTD

sparseTScalar

sparseVConcat

spline1d

spline2d

vartype

WinClear

WinClearArea

WinClearTTYlog

WinClose

WinCloseAll

WinGetActive

WinGetAttributes

WinGetColorCells

WinGetCursor

WinMove

WinOpenPQG

WinOpenText

WinOpenTTY

WinPan

WinPrint

WinPrintPQG

WinRefresh

WinRefreshArea

WinResize

WinSetActive

WinSetBackground

WinSetColorCells

WinSetColormap

WinSetCursor

WinSetForeground

WinSetRefresh **WinZoomPQG**

WinSetTextWrap

Colors E

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

Index

' , 11-8
.' , 11-8
~ , 11-9
| , 11-8

! , 11-6
% , 11-5
* , 11-5
*~ , 11-7
.* , 11-6
.*. , 11-6
+ , 11-4
– , 11-4
/ , 11-5
./ , 11-6
^ , 11-6, 11-19
.^ , 11-6

, (comma) , 11-16
. (dot) , 11-16
: (colon) , 11-17
; (semicolon) , 10-2

, 22-3, 31-119, 31-120, 31-516, 31-682, 31-907
$, 31-119, 31-120, 31-516, 31-682, 31-907
~ , 11-19
$| , 11-19
$+ , 11-18

& , 11-17, 12-10

= , 10-2, 10-12, 10-40
= , 11-16

/ = , 11-11
./= , 11-12
== , 10-40, 11-11
.== , 11-12
> , 11-11
. > , 11-13
>= , 11-11
. >= , 11-12
< , 11-10
. < , 11-12
<= , 11-10
. <= , 11-12

__altnam, 31-240
__output, 31-240, 31-505, 31-800
__title, 31-240
__Tol, 31-240
_eqs_IterInfo, 31-240
_eqs_JacobianProc, 31-240
_eqs_MaxIters, 31-240
_eqs_StepTol, 31-240
_eqs_TypicalF, 31-240
_eqs_TypicalX, 31-240
_loess_Degree, 31-505

_loess_NumEval, 31-505
_loess_Span, 31-505
_loess_WgtType, 31-505
_sqp_A, 31-797
_sqp_B, 31-797
_sqp_Bounds, 31-798
_sqp_C, 31-798
_sqp_D, 31-798
_sqp_DirTol, 31-799
_sqp_EqProc, 31-797
_sqp_FeasibleTest, 31-799
_sqp_GradProc, 31-798
_sqp_HessProc, 31-799
_sqp_IneqProc, 31-798
_sqp_MaxIters, 31-799
_sqp_ParNames, 31-799
_sqp_PrintIters, 31-799
_sqp_RandRadius, 31-800

A

abs, 31-1
absolute value, 31-1
acf, 31-2
aconcat, 15-4, 31-3
additive sequence, 31-756
aeye, 15-6, 31-5
algebra, linear, 30-4
amax, 15-25, 31-6
amean, 15-25, 31-8
AmericanBinomCall, 31-10
AmericanBinomCall_Greeks, 31-11
AmericanBinomCall_ImpVol, 31-13
AmericanBinomPut, 31-14
AmericanBinomPut_Greeks, 31-15
AmericanBinomPut_ImpVol, 31-17
AmericanBSCall, 31-18
AmericanBSCall_Greeks, 31-19
AmericanBSCall_ImpVol, 31-20
AmericanBSPut, 31-21
AmericanBSPut_Greeks, 31-22
AmericanBSPut_ImpVol, 31-23
amin, 15-25, 31-24
ampersand, 11-17
amult, 15-23, 31-26
and, 11-13, 11-14
.and, 11-15
annualTradingDays, 31-28
append, ATOG command, 26-3
arccos, 31-29
arcsin, 31-30
areshape, 15-2, 31-31
arguments, 10-40, 12-3, 12-7
array indexing, 14-3
arrayalloc, 15-7, 31-32
arrayindex, 31-33
arrayinit, 15-6, 31-34
arrays, 14-1, 15-1, 30-27
arrays of structures, 16-4
arraytomat, 15-28, 31-35
arrows, 24-14, 24-16
ASCII files, 26-1
ASCII files, packed, 26-8
ASCII files, reading, 20-3
ASCII files, writing, 20-4
asciiload, 31-36
asclabel, 31-37
assigning to arrays, 15-8
assignment operator, 10-2, 10-40, 11-16
astd, 31-38
astds, 31-40
asum, 31-42
atan, 31-44
atan2, 31-45
atog, 20-3

ATOG, 26-1
atranspose, 15-21, 31-46
autoloader, 10-4, 10-5, 18-1, 31-480
auxiliary output, 20-4, 31-574
auxiliary output, width, 31-577
axes, 24-17, 24-19
axes numbering, 24-26
axes, reversed, 31-929, 31-932
axmargin, 31-48

B

backslash, 10-22
balance, 31-50
band, 31-51
bandchol, 31-52
bandcholsol, 31-53
bandltsol, 31-55
bandrv, 31-56
bandsolpd, 31-58
bar shading, 24-17
bar width, 24-18
bar, 31-58
base10, 31-60
batch mode, 3-1
begwind, 31-61
besselj, 31-61
bessely, 31-62
beta function, 31-67
binary file, loading, 31-360
binary files, 20-15
bivariate Normal, 31-69
blank lines, 10-38
bookmarks, 5-2
Boolean operators, 11-13
box, 24-18
box, 31-63
boxcox, 31-64
branching, 30-41
break, 31-65
Breakpoints, 5-8
browse, 3-3

C

call, 31-66
calling a procedure, 12-6
caret, 11-6, 11-19
Cartesian coordinates, 31-930
case, 10-38, 31-511, 31-890
cdfbeta, 31-67
cdfbvn, 31-69
cdfbvn2, 31-71
cdfbvn2e, 31-73
cdfchic, 31-74
cdfchii, 31-76
cdfchinc, 31-77
cdffc, 31-78
cdffnc, 31-80
cdfgam, 31-81
cdfm.src, 31-84, 31-86, 31-87, 31-89, 31-90, 31-92
cdfmvn, 31-83
cdfmvn2e, 31-86
cdfmvnce, 31-83
cdfmvne, 31-85
cdfmvt2e, 31-91
cdfmvtce, 31-87
cdfmvte, 31-89
cdfn, 31-93
cdfn2, 31-95
cdfnc, 31-93
cdfni, 31-96
cdftc, 31-97
cdftci, 31-98
cdftnc, 31-99

cdftvn, 31-100
cdir, 31-102
ceil, 31-103
ChangeDir, 31-104
characteristic polynomial, 31-587
chdir, 31-104
chi-square, 31-74
chi-square, noncentral, 31-77
chiBarSquare, 31-105
chol, 31-106
choldn, 31-108
Cholesky decomposition, 0-1, 11-5, 31-107, 31-772
cholsol, 31-109
cholup, 31-110
chrs, 31-111
circles, 24-22
clear, 31-112
clearg, 31-112
close, 31-113
closeall, 31-115
cls, 31-116
code (dataloop), 31-119
code, 31-117
coefficient of determination, 31-554, 31-560
coefficients, 31-554, 31-559
coefficients, standardized, 31-554, 31-559
colon, 10-39
color, 24-19, 24-25
Colors, 0-1
colors, 0-1
cols, 31-120
colsf, 31-121
columns in a matrix, 31-120
combinate, 31-122
combinated, 31-123
comlog, 31-124
comma, 11-16
command, 10-2
Command Input - Output Window, 5-4
command line, 3-1
comments, 10-38
comparison functions, 31-278, 31-280
comparison operator, 10-40
compilation phase, 22-3
compile, 19-1
compile options, 5-11
compile time, 10-1
compile, 31-125
compiled language, 10-1
compiler, 19-1
compiler directives, 30-39
compiling, 30-44
compiling files, 19-2
compiling programs, 19-2
complex constants, 10-14, 31-176, 31-474, 31-815
complex modulus, 31-1
complex, 26-4, 31-127
con, 31-128
concatenation, matrix, 11-8, 11-9
concatenation, string, 11-18
cond, 31-130
condition number, 31-130
conditional branching, 10-35
config, 3-3
conformability, 11-1
conj, 31-131
cons, 31-132
ConScore, 31-132
constants, complex, 10-14, 31-176, 31-474, 31-815
continue, 31-136
contour levels, 24-22

contour, 31-137
control flow, 10-32
control structures, 16-22
conv, 31-138
conversion, character to ASCII value, 31-895
conversion, float to ASCII, 31-316, 31-317
conversion, string to floating point, 31-814
convertsatostr, 31-139
convertstrtosa, 31-139
convolution, 31-138
coordinates, 24-6
correlation matrix, 31-140, 31-141, 31-554, 31-560
corrm, 31-140
corrms, 31-141
corrvc, 31-140
corrx, 31-140
corrxs, 31-141
cos, 31-141
cosh, 31-142
cosine, inverse, 31-29
counts, 31-143
countwts, 31-145
create, 31-146
cropping, 24-19
cross-product, 31-152
crossprd, 31-152
Crout decomposition, 31-153, 31-154
Crout LU decomposition, 0-1
crout, 31-153
croutp, 31-154
csrcol, 31-156
csrlin, 31-156
cumprodc, 31-157
cumsumc, 31-158
cumulative distribution function, 31-67
cumulative products, 31-157
cumulative sums, 31-158
cursor, 31-156, 31-504
curve, 31-159
cvtos, 31-160

D

data coding, 30-36
data handling, 30-30
data loop, 22-1
data sets, 20-7, 30-33
data transformations, 22-1, 31-117, 31-183
data, writing, 31-915
datacreate, 31-161
datacreatecomplex, 31-163
datalist, 31-165
dataload, 31-166
dataloop translator, 3-3
dataloop, 31-166
dataopen, 31-167
datasave, 31-169
date, 24-20, 31-170
date, 25-2, 31-170
datestr, 31-170
datestring, 31-171
datestrymd, 31-172
dayinyr, 31-172
dayofweek, 31-173
debug, 31-174
Debugger, 5-7
debugging, 3-4, 19-3, 30-47, 31-482
declare, 31-174
delete (dataloop), 31-181
delete, 31-180
DeleteFile, 31-182
deletion, 31-211, 31-213, 31-581
delif, 31-183
delimited, 26-1

delimited files, 20-3
delimited, hard, 26-6
delimited, soft, 26-5
denseToSp, 31-184
denseToSpRE, 31-185
denToZero, 31-186
derivatives, 31-380
derivatives, second partial, 31-397
descriptive statistics, 31-210, 31-212
design matrix, 31-187
design, 31-187
det, 31-188
determinant, 31-188
detl, 31-189
dfft, 31-190
dffti, 31-191
diag, 31-191
diagonal, 31-191
diagrv, 31-192
differentiation, 30-3
digamma, 31-193
dimension index, 14-2
dimension number, 14-2
directory, 31-102
division, 11-5
dlibrary, 21-1, 31-194
dllcall, 21-1, 31-195
do loop, 10-32
do until, 31-197
do while, 31-197
dos, 31-200
doswin, 31-202
DOSWinCloseall, 31-202
DOSWinOpen, 31-203
dot relational operator, 11-11, 11-21
dotmtfeq, 31-205
dotmtfeqmt, 31-206
dotfge, 31-205
dotfgemt, 31-206
dotfgt, 31-205
dotfgtmt, 31-206
dotfle, 31-205
dotflemt, 31-206
dotflt, 31-205
dotfltmt, 31-206
dotfne, 31-205
dotfnemt, 31-206
downloading, 2-2
draw, 31-208
drop (dataloop), 31-209
DS structure, 16-15, 17-7
dsCreate, 31-210
dstat, 31-210
dstatmt, 31-212
dstatmtControlCreate, 31-214
dtdate, 31-215
dtday, 31-216
dttime, 31-216
dttodtv, 31-217
dttostr, 31-218
dttoutc, 31-220
dtv vector, 25-3
dtvnormal, 25-3, 31-221
dtvtodt, 31-222
dtvtoutc, 31-223
dummy variables, 31-224
dummy, 31-224
dummybr, 31-225
dummydn, 31-227
Durbin-Watson statistic, 31-553, 31-557
dynamic libraries, 21-3

E

E×E conformable, 11-1

ed, 31-228
edit windows, 5-1
edit, 31-229
editing matrices, 6-1
editor, 31-229
editor properties, 5-2
editor, alternate, 31-229
Editor, Matrix, 6-1
eig, 31-230
eigenvalues, 30-9, 31-230
eigenvalues and eigenvectors, 31-233
eigh, 31-231
eighv, 31-232
eigv, 31-233
elapsedTradingDays, 31-235
element-by-element conformability, 11-1, 14-5
element-by-element operators, 11-1
else, 31-402
elseif, 31-402
empty matrix, 10-15, 31-121, 31-475, 31-496, 31-734, 31-747
end of file, 31-239
end, 31-235
endp, 12-2, 12-5, 31-236
endwind, 31-237
envget, 31-238
environment, search, 31-238
eof, 31-239
eq, 11-11
.eq, 11-12
eqSolve, 31-240
eqSolvemt, 31-244
eqSolvemtControlCreate, 31-248
eqSolvemtOutCreate, 31-248
eqSolveSet, 31-249
eqv, 11-14, 11-15
.eqv, 11-15
erf, 31-249
erfc, 31-249
error bar, 24-20
error code, 31-251, 31-746
error function, 31-249
error handling, 30-47
error messages, 27-1, 31-252, 31-482
Error Output Window, 5-7
error trapping, 31-875
error, 31-250
errorlog, 31-252
errorlogat, 31-252
escape character, 10-22
etdays, 31-253
ethsec, 31-254
etstr, 25-5, 31-255
EuropeanBinomCall, 31-255
EuropeanBinomCall_Greeks, 31-257
EuropeanBinomCall_ImpVol, 31-258
EuropeanBinomPut, 31-259
EuropeanBinomPut_Greeks, 31-260
EuropeanBinomPut_ImpVol, 31-262
EuropeanBSCall, 31-263
EuropeanBSCall_Greeks, 31-264
EuropeanBSCall_ImpVol, 31-265
EuropeanBSPut, 31-266
EuropeanBSPut_Greeks, 31-267
EuropeanBSPut_ImpVol, 31-269
exctsmpl, 31-270
exec, 31-271
execbg, 31-272
executable code, 10-4
executable statement, 10-3
execution phase, 22-4
execution time, 10-1
exp, 31-273

exponential function, 31-273
exponentiation, 11-6
expression, 10-1
expression, evaluation order, 10-30
expression, scalar, 10-32
extern (dataloop), 31-273
external, 31-274
extraneous spaces, 10-39
eye, 31-276

F

F distribution, 31-78, 31-80
factorial, 11-6
FALSE, 10-32
fcheckerr, 31-277
fclearerr, 31-277
feq, 31-278
feqmt, 31-280
fflush, 31-281
fft, 31-282
fft, 31-282
ffti, 31-283
fftm, 31-284
fftmi, 31-286
fftn, 31-289
fge, 31-278
fgemt, 31-280
fgets, 31-290
fgetsa, 31-291
fgetsat, 31-292
fgetst, 31-293
fgt, 31-278
fgtmt, 31-280
file formats, 20-14
file handle, 31-149, 31-568
fileinfo, 31-293
files, 20-3
files, binary, 20-15
files, matrix, 20-13
files, string, 20-16
filesa, 31-295
finance functions, 30-20
fle, 31-278
flemt, 31-280
floor, 31-296
flow control, 10-32
flt, 31-278
fltmt, 31-280
fmod, 31-297
fn, 31-297
fne, 31-278
fnemt, 31-280
fonts, 0-1, 31-298
fonts, 31-298
fopen, 31-299
for, 31-300
Foreign Language Interface, 21-1
format, 31-302
formatcv, 31-309
formatnv, 31-310
forward reference, 18-2
Fourier transform, 31-282
Fourier transform, discrete, 31-190, 31-191
fourier transforms, 30-10
fputs, 31-311
fputst, 31-312
fseek, 31-313
fstrerror, 31-314
ftell, 31-315
ftocv, 31-316
ftos, 31-317
ftostrC, 31-320
function, 10-37, 31-448, 31-609
functions, 30-43

fuzzy conditional functions, 30-12

G

gamma function, 31-322
gamma, 31-322
gamma, incomplete, 31-81
gamma, log, 31-490
gammaii, 31-322
GAUSS Data Archives, 20-11, 20-24, 30-32
GAUSS Source Browser, 8-1
Gauss-Legendre quadrature, 31-429
gausset, 29-6, 31-323
gdaAppend, 31-323
gdaCreate, 31-325
gdaDStat, 31-326
gdaDStatMat, 31-328
gdaGetIndex, 31-331
gdaGetName, 31-332
gdaGetNames, 31-332
gdaGetOrders, 31-333
gdaGetType, 31-334
gdaGetTypes, 31-335
gdaGetVarInfo, 31-336
gdaIsCplx, 31-337
gdaLoad, 31-338
gdaPack, 31-341
gdaRead, 31-342
gdaReadByIndex, 31-343
gdaReadSome, 31-343
gdaReadSparse, 31-345
gdaReadStruct, 31-346
gdaReportVarInfo, 31-347
gdaSave, 31-348
gdaUpdate, 31-350
gdaUpdateAndPack, 31-352
gdaVars, 31-353
gdaWrite, 31-354
gdaWrite32, 31-355
gdaWriteSome, 31-356
ge, 11-11
.ge, 11-12
generalized inverse, 31-440, 31-585, 31-586
getarray, 31-358
getArray, 15-12
getdims, 31-359
getDims, 15-27
getf, 31-360
getmatrix, 31-361
getMatrix, 15-13
getmatrix4D, 31-362
getMatrix4D, 15-13
getname, 31-363
getnamef, 31-364
getNextTradingDay, 31-365
getNextWeekDay, 31-366
getnr, 31-366
getnrmt, 31-367
getOrders, 15-27
getorders, 31-368
getpath, 31-368
getPreviousTradingDay, 31-369
getPreviousWeekDay, 31-370
getRow, 31-370
getScalar3D, 15-14
getscalar3D, 31-371
getScalar4D, 15-14
getscalar4D, 31-372
getTrRow, 31-373
getwind, 31-373
global control variables, 29-5
global variable, 12-3
Goertzel algorithm, 31-190
gosub, 31-374
goto, 31-377

gradient, 31-380
gradMT, 31-378
gradMTm, 31-379
gradp, 31-380
graphic panels, 24-7
graphic panels, nontransparent, 24-8
graphic panels, overlapping, 24-7
graphic panels, tiled, 24-7
graphic panels, transparent, 24-8
graphics, publication quality, 24-1
graphprt, 31-382
graphset, 31-384
grid, 24-21
grid subdivisions, 24-21
gt, 11-11
.gt, 11-13

H

hard delimited, 26-6
hasimag, 31-385
hat operator, 11-6, 11-19
header, 31-386
headermt, 31-387
help, 9-1
help facility, 31-482
hermitian matrix, 31-232
hess, 31-388
Hessian, 31-397
hessMT, 31-389
hessMTg, 31-390
hessMTgw, 31-391
hessMTm, 31-393
hessMTmw, 31-394
hessMTw, 31-395
hessp, 31-397
hidden lines, 24-27
hist, 31-398
histf, 31-399
histogram, 31-398, 31-399
histp, 31-400
horizontal direct product, 11-7
hsec, 31-401
hyperbolic cosine, 31-142
hyperbolic sine, 31-770
hyperbolic tangent, 31-863

I

if, 31-402
imag, 31-403
imaginary matrix, 31-403
inch coordinates, 24-6
#include, 31-404
incomplete beta function, 31-67
incomplete gamma function, 31-81
indcv, 31-405
indefinite, 10-28
index variables, 31-567
indexcat, 31-406
indexing matrices, 10-41, 11-17
indexing procedures, 11-17
indexing, array, 14-3
indexing, structure, 16-5
indices, 31-408
indices2, 31-409
indicesf, 31-410
indicesfn, 31-411
indnv, 31-412
indsav, 31-413
infinity, 10-28
initialize, 12-4
initializing arrays, 15-1
inner product, 11-5
input, ATOG command, 26-4
input, console, 31-128

input, keyboard, 31-128
installation, 2-1
installation, UNIX/Linux, 2-1
installation, Windows, 2-2
instruction pointer, 10-3
integration, 30-3, 30-4, 31-418, 31-421, 31-423, 31-426, 31-429, 31-437
interactive commands, 3-2
interpreter, 10-1
intersection, 31-436
intgrat2, 31-414
intgrat3, 31-416
inthp1, 31-418
inthp2, 31-420
inthp3, 31-423
inthp4, 31-426
inthpControlCreate, 31-429
intquad1, 31-429
intquad2, 31-431
intquad3, 31-432
intrinsic function, 10-8
intrleav, 31-434
intrleavsa, 31-435
intrsect, 31-436
intrsectsa, 31-437
intsimp, 31-437
inv, 31-438
invar, ATOG command, 26-5
inverse cosine, 31-29
inverse sine, 31-30
inverse, generalized, 31-440, 31-585, 31-586
inverse, matrix, 31-438
inverse, sweep, 31-440
invpd, 31-438
invswp, 31-440
iscplx, 31-441
iscplxf, 31-442
isden, 31-442
isinfnanmiss, 31-443
ismiss, 31-443

J

Jacobian, 31-380

K

keep (dataloop), 31-444
key, 31-445
keyav, 31-447
keyboard input, 31-132
keyboard, reading, 31-445
keys, command, 5-17
keys, edit, 5-16
keys, function, 5-18
keys, menu, 5-19
keys, movement, 5-15
keys, text selection, 5-17
keystroke macros, 5-2
keyw, 31-447
keyword, 12-1, 12-7
keyword procedure, 31-448
keyword, 31-448
keywords, 30-43
Kronecker, 11-6

L

label, 10-36, 10-39, 12-1, 31-374, 31-377
lag (dataloop), 31-449
lag1, 31-450
lagn, 31-450
lambda, 31-77
lapeighb, 31-451
lapeighi, 31-452
lapeigvb, 31-453

lapeigvi, 31-455
lapgeig, 31-456
lapgeigh, 31-457
lapgeighv, 31-458
lapgeigv, 31-459
lapgschur, 31-468
lapgsvdcst, 31-460
lapgsvds, 31-463
lapgsvdst, 31-465
lapsvdcusv, 31-469
lapsvds, 31-471
lapsvdusv, 31-472
le, 11-10
.le, 11-12
least squares, 11-5
least squares regression, 31-551, 31-556
left-hand side, 18-2
legend, 24-22
let, 31-473
lib, 31-478
libraries, 18-1, 30-44
libraries, active, 31-480
Library Tool, 7-1
library, 31-479
line numbers, 31-482
line thickness, 24-16, 24-20, 24-25
line type, 24-25
linear algebra, 30-4
linear equation, 31-772
linear equation solution, 11-5
lines, 24-21, 24-22, 24-24
#linesoff, 31-482
#lineson, 31-482
linsolve, 31-483
listwise (dataloop), 31-484
listwise deletion, 31-211, 31-213, 31-581
literal, 10-23, 11-19
ln, 31-484
lncdfbvn, 31-485
lncdfbvn2, 31-486
lncdfmvn, 31-488
lncdfn, 31-488
lncdfn.src, 31-83, 31-96
lncdfn2, 31-489
lncdfnc, 31-490
lnfact, 31-490
lnpdfmvn, 31-491
lnpdfmvt, 31-492
lnpdfn, 31-493
lnpdft, 31-493
load, 31-494
loadarray, 31-499
loadd, 31-501
loadf, 31-494
loadk, 31-494
loadm, 31-494
loadp, 31-494
loads, 31-494
loadstruct, 31-502
loadwind, 31-502
local variable declaration, 12-3
local variables, 10-8, 12-3, 31-503
local, 12-2, 31-503
locate, 31-504
loess, 31-504
loessmt, 31-505
loessmtControlCreate, 31-506
log coordinates, 31-508
log factorial, 31-490
log gamma, 31-490
log, 31-507
log, base 10, 31-507
log, natural, 31-484
logging commands, 31-124

logical operators, 11-13
loglog, 31-508
logx, 31-508
logy, 31-509
looping, 10-32, 30-41, 31-197
looping with arrays, 15-17
loopnextindex, 15-19, 31-510
lower triangular matrix, 31-512
lower, 31-511
lowmat, 31-512
lowmat1, 31-512
lt, 11-10
.lt, 11-12
ltrisol, 31-513
LU decomposition, 11-5, 31-514
lu, 31-514
lusol, 31-515

M

machEpsilon, 31-515
machine epsilon, 31-93, 31-855, 31-860
machine requirements, 2-2
macros, 5-2
magnification, 24-29
make (dataloop), 31-516
makevars, 31-516
makewind, 31-518
margin, 31-519
matalloc, 31-520
matinit, 31-521
matrices, indexing, 10-41
matrix conformability, 11-1
Matrix Editor, 6-1
matrix files, 20-13
matrix manipulation, 30-22
matrix, creation, 31-473
matrix, empty, 10-15, 31-121, 31-475, 31-496, 31-734, 31-747
matrix, ones, 31-565
matrix, zeros, 31-932
mattoarray, 15-28, 31-521
maxbytes, 31-526
maxc, 31-522
maximizing performance, 28-1
maximum element, 31-522
maximum element index, 31-523
maxindc, 31-523
maxv, 31-524
mbesseli, 31-527
mean, 31-530
meanc, 31-530
median, 31-530
memory, 31-181
memory, clear all, 31-545
menus, 4-1
mergeby, 31-531
mergevar, 31-532
merging, 30-38
minc, 31-533
minimum element, 31-533
minimum element index, 31-534
minindc, 31-534
minv, 31-538
miss, 31-535
missex, 31-537
missing character, 31-544
missing values, 11-5, 31-211, 31-212, 31-443, 31-535, 31-537, 31-544, 31-581, 31-749
missrv, 31-535
modulo division, 11-5
moment matrix, 31-554, 31-559
momentd, 31-541

Moore-Penrose pseudo-inverse, 31-585, 31-586
movingave, 31-542
movingaveExpwgt, 31-543
movingaveWgt, 31-544
msym, 31-544
msym, ATOG command, 26-10
multiplication, 11-5
multiplicative sequence, 31-756

N

N-dimensional arrays, 14-1, 15-1, 30-27
NaN, 10-28
NaN, testing for, 10-29, 11-9
ne, 11-11
.ne, 11-12
new, 31-545
nextindex, 31-546
nextn, 31-547
nextnevn, 31-547
nextwind, 31-548
nocheck, 26-10
Normal distribution, 31-83, 31-85, 31-86, 31-87, 31-89, 31-91, 31-93, 31-95, 31-485, 31-488, 31-489, 31-490
Normal distribution, bivariate, 31-69
not, 11-13, 11-14
.not, 11-15
null space, 31-549
null, 31-549
null1, 31-550
numCombinations, 31-551

O

obsolete commands, 0-1
ols, 31-551
olsmt, 31-556
olsmtControlCreate, 31-562
olsqr, 31-562
olsqr2, 31-563
olsqrmt, 31-564
ones, 31-565
open, 31-566
operators, 10-1, 11-4
operators, element-by-element, 11-1
optimization, 30-16
optn, 31-571
optnevn, 31-571
or, 11-14, 11-15
.or, 11-15
orth, 31-573
orthogonal complement, 31-549
orthonormal, 31-549, 31-573
outer product, 11-6
output, 20-4
output functions, 30-52
output, 31-574
output, ATOG command, 26-10
outtyp (dataloop), 31-577
outtyp, ATOG command, 26-11
outvar, ATOG command, 26-11
outwidth, 31-577

P

pacf, 31-578
packed ASCII, 26-1, 26-8
packedToSp, 31-579
packr, 31-581
_pageshf, 24-14
_pagesiz, 24-14
pairwise deletion, 11-5, 31-211, 31-213
panel data, 15-32
_parrow, 24-14

_parrow3, 24-16
parse, 31-582
pause, 31-583
_paxes, 24-17
_paxht, 24-17
_pbartyp, 24-17
_pbarwid, 24-18
_pbox, 24-18
_pboxlim, 24-19
_pcolor, 24-19
_pcrop, 24-19
_pcross, 24-19
_pdate, 24-20
pdfn, 31-583
_perrbar, 24-20
_pframe, 24-20
_pgrid, 24-21
pi, 31-584
pinv, 31-585
pinvmt, 31-586
pixel coordinates, 24-6
_plctrl, 24-21
_plegctl, 24-22
_plegstr, 24-22
_plev, 24-22
_pline, 24-22
_pline3d, 24-24
plot coordinates, 24-6
_plotshf, 24-24
_plotsiz, 24-25
_pltype, 24-25
_plwidth, 24-25
_pmcolor, 24-25
_pmsgctl, 24-26
_pmsgstr, 24-26
_pnotify, 24-26
_pnum, 24-26
_pnumht, 24-27
pointer, 11-17, 12-10, 12-11, 31-503
pointer, instruction, 10-3
pointers, structure, 16-10
polar, 31-587
polychar, 31-587
polyeval, 31-588
polyint, 31-589
polymake, 31-590
polymat, 31-591
polymroot, 31-591
polymult, 31-593
polynomial, 31-590
polynomial interpolation, 31-589
polynomial operations, 30-9
polynomial regression, 31-591
polynomial, characteristic, 31-587
polynomial, evaluation, 31-588
polynomial, roots, 31-594
polyroot, 31-594
pop, 31-594
pqgwin, 31-595
precedence, 10-30
precision control, 30-20
predicted values, 31-563
preferences, 5-10
preservecase, 26-12
previousindex, 31-596
princomp, 31-597
print, 31-598
printdos, 31-604
printfm, 31-605
printfmt, 31-608
probability density function, Normal, 31-583
proc, 12-2, 31-609
procedure, 12-1, 31-503, 31-609
procedure, definitions, 10-3, 12-2

procedures, 30-43
procedures, indexing, 12-10
procedures, multiple returns, 12-11
procedures, passing to other procedures, 12-9
prodc, 31-610
products, 31-611
Profiler, 23-1
program, 10-4
program control, 30-40
program space, 31-766
program, run, 31-736
properties, editor, 5-2
_protate, 24-27
_pscreen, 24-27
pseudo-inverse, 31-585, 31-586
_psilent, 24-27
_pstype, 24-27
_psurf, 24-27
_psym, 24-28
_psym3d, 24-28
_psymsiz, 24-28
_ptek, 24-28
_pticout, 24-28
_ptitlht, 24-28
Publication Quality Graphics, 24-1, 30-54
putArray, 15-15
putarray, 31-611
putf, 31-612
putvals, 31-614
PV structure, 16-16, 17-1
pvCreate, 31-615
_pversno, 24-29
pvGetIndex, 31-615
pvGetParNames, 31-616
pvGetParVector, 31-617
pvLength, 31-618
pvList, 31-619
pvPack, 31-619
pvPacki, 31-620
pvPackm, 31-621
pvPackmi, 31-623
pvPacks, 31-624
pvPacksi, 31-625
pvPacksm, 31-627
pvPacksmi, 31-629
pvPutParVector, 31-631
pvTest, 31-632
pvUnpack, 31-633
_pxpmax, 24-29
_pxsci, 24-29
_pypmax, 24-29
_pysci, 24-29
_pzclr, 24-29
_pzoom, 24-29
_pzpmax, 24-29
_pzsci, 24-29

Q

QNewton, 31-633
QNewtonmt, 31-636
QNewtonmtControlCreate, 31-641
QNewtonmtOutCreate, 31-641
QNewtonSet, 31-642
QProg, 31-642
QProgmt, 31-644
QProgmtInCreate, 31-646
qqr, 31-646
qqre, 31-648
qqrep, 31-651
QR decomposition, 31-562, 31-564
qr, 31-653
qre, 31-654
qrep, 31-657
qrsol, 31-659

qrtsol, 31-660
qtyr, 31-660
qtyre, 31-663
qtyrep, 31-666
quadrature, 31-429
quantile, 31-668
quantiled, 31-669
qyr, 31-671
qyre, 31-672
qyrep, 31-674

R

radii, 24-22
random numbers, 30-10
rank of a matrix, 31-676
rank, 31-676
rankindx, 31-677
readr, 31-678
real, 31-679
recode (dataloop), 31-682
recode, 31-680
recserar, 31-683
recsercp, 31-685
recserrc, 31-686
recursion, 12-5
reduced row echelon form, 31-735
regression, 31-551, 31-556
relational operator, dot, 11-11, 11-21
relational operators, 11-9
relative error, 31-93, 31-98
rerun, 31-687
reserved words, 0-1
reshape, 31-688
residuals, 31-553, 31-557, 31-563
retp, 12-2, 12-5, 31-689
return, 31-690
rev, 31-690
rfft, 31-691
rfffti, 31-692
rfftip, 31-693
rfftn, 31-694
rfftnp, 31-695
rfftp, 31-697
right-hand side, 18-2
rndbeta, 31-698
rndcon, 31-699
rndgam, 31-701
rndi, 31-702
rndKMbeta, 31-703
rndKMgam, 31-704
rndKMi, 31-706
rndKMn, 31-707
rndKMnb, 31-709
rndKMp, 31-710
rndKMu, 31-711
rndKMvm, 31-713
rndLCbeta, 31-714
rndLCgam, 31-715
rndLCi, 31-717
rndLCn, 31-719
rndLCnb, 31-721
rndLCp, 31-723
rndLCu, 31-724
rndLCvm, 31-726
rndmult, 31-699
rndn, 31-728
rndnb, 31-729
rndp, 31-730
rndseed, 31-699
rndu, 31-730
rndvm, 31-732
rotater, 31-732
round down, 31-296
round up, 31-103

round, 31-733
rows, 31-734
rowsf, 31-735
rref, 31-735
rules of syntax, 10-38
run options, 5-11
run, 31-736
Run-Time Library structures, 17-1
running commands, 5-4
running programs, 5-5

S

satostrC, 31-738
save, 31-739
saveall, 31-741
saved, 31-742
savestruct, 31-743
savewind, 31-744
saving the workspace, 19-2
scalar error code, 31-251, 31-746
scalar expression, 10-32
scale, 31-744
scale3d, 31-745
scalerr, 31-746
scalinfnanmiss, 31-748
scaling, 31-744, 31-745
scalmiss, 31-749
schtoc, 31-750
schur, 31-751
scientific functions, 30-1
screen, 31-752
searchsourcepath, 31-753
secondary section, 10-5
seekr, 31-754
select (dataloop), 31-755
selif, 31-755
semicolon, 10-2
seqa, 31-756
seqm, 31-756
sequence function, 31-756
sequence functions, 30-19
series functions, 30-19
set difference function, 31-759
setArray, 15-16
setarray, 31-758
setdif, 31-759
setdifsa, 31-760
setvars, 31-761
setvwrmode, 31-762
setwind, 31-762
shell, 31-763
shiftr, 31-764
show, 31-765
Simpson's method, 31-437
sin, 31-768
sine, inverse, 31-30
singleindex, 31-769
singular value decomposition, 31-841, 31-842, 31-843, 31-845
singular values, 31-840, 31-844
singularity tolerance, 0-1
sinh, 31-770
sleep, 31-771
soft delimited, 26-5
solpd, 31-772
sort data file, 31-775
sort index, 31-777
sort, heap sort, 31-776
sort, multiple columns, 31-778
sort, quicksort, 31-774
sortc, 31-774
sortcc, 31-774
sortd, 31-775
sorthc, 31-776

sorthcc, 31-776
sortind, 31-777
sortindc, 31-777
sorting, 30-38
sortmc, 31-778
sortr, sortrc, 31-779
Source Browser, 8-1
spaces, 11-17
spaces, extraneous, 10-39, 11-17
sparse matrices, 30-26
spCreate, 31-780
spDenseSubmat, 31-781
spDiagRvMat, 31-782
spEye, 31-784
spGetNZE, 31-785
spline, 31-786
spNumNZE, 31-787
spOnes, 31-788
SpreadsheetReadM, 31-789
SpreadsheetReadSA, 31-790
spreadsheets, 30-30
SpreadsheetWrite, 31-790
spScale, 31-791
spSubmat, 31-792
spToDense, 31-793
spTrTDense, 31-794
spTScalar, 31-795
spZeros, 31-796
sqpSolve, 31-797
sqpSolvemt, 16-23
sqpSolveMT, 31-801
sqpSolvemtControl structure, 16-25
sqpSolveMTControlCreate, 31-809
sqpSolveMTlagrangeCreate, 31-809
sqpSolveMToutCreate, 31-810
sqpSolveSet, 31-810
sqrt, 31-811
square root, 31-811
src_path, 18-1
standard deviation, 31-38, 31-40, 31-212, 31-213, 31-812, 31-813
standard deviation of residual, 31-554, 31-560
standard errors, 31-554, 31-560
statement, 10-2, 10-38
statement, executable, 10-3
statement, nonexecutable, 10-3
statistical distributions, 30-17
statistical functions, 30-13
statistics, descriptive, 31-210, 31-212
status bar, 4-15
stdc, 31-812
stdsc, 31-813
Stirling's formula, 31-491
stocv, 31-814
stof, 31-814
stop, 31-815
strcombine, 31-815
strindx, 31-816
string array concatenation, 11-19
string arrays, 10-24, 10-25
string concatenation, 11-18
string files, 20-16
string handling, 30-47
string index, 31-816, 31-819
string length, 31-817
string, long, 10-38
string, substring, 31-820
strings, graphics, 24-26
strlen, 31-817
strput, 31-818
strrindx, 31-819
strsect, 31-820
strsplit, 31-821

strsplitPad, 31-822
strtodt, 31-823
strtof, 31-825
strtofcplx, 31-825
strtriml, 31-826
strtrimr, 31-826
strtrunc, 31-827
strtruncl, 31-827
strtruncpad, 31-828
strtruncr, 31-828
structure definition, 16-1
structure indexing, 16-5
structure instance, 16-2
structure pointers, 16-10
structure, **DS**, 16-15, 17-7
structure, **PV**, 16-16, 17-1
structures, 16-1, 30-29
structures, arrays of, 16-4
structures, control, 16-22
submat, 31-829
submatrix, 31-829
subroutine, 10-37, 31-374
subroutines, 30-42
subsample, 31-270
subscat, 31-830
substitution, 11-19
substring, 31-820
substute, 31-831
subvec, 31-833
sum, 31-834
sumc, 31-834
sumr, 31-836
surface, 31-838
svd, 31-840
svd1, 31-841
svd2, 31-842
svdcusv, 31-843
svds, 31-844
svdusv, 31-845
sweep inverse, 31-440
symbol names, 10-39
symbol table, 31-765
symbol table type, 31-880
symbols, allocate maximum number, 31-545
syntax, 10-38
sysstate, 31-846
system, 31-861

T

t distribution, Student's, 31-97
tab, 31-861
table, 11-6
tan, 31-862
tanh, 31-863
tempname, 31-864
tensor, 11-6
text files, 30-31
TGAUSS, 3-1
thickness, line, 24-16, 24-20, 24-25
tick marks, 24-28
tilde, 11-9
time and date functions, 30-50
time, 25-2, 31-865
time, elapsed, 31-253
timed iterations, 25-6
timedt, 31-865
timestr, 31-866
timeutc, 31-867
timing functions, 31-401
title, 31-867
tkf2eps, 31-868
tkf2ps, 31-869
tocart, 31-869
todaydt, 31-870

Toeplitz matrix, 31-870
toeplitz, 31-870
token, 31-871
toolbars, 4-10
topolar, 31-872
trace program execution, 31-873
trace, 31-873
translation phase, 22-3
transpose, 11-8
transpose, bookkeeping, 11-8
trap flag, 31-875, 31-877
trap state, 31-747
trap, 31-874
trapchk, 31-877
triangular matrix, lower, 31-512
triangular matrix, upper, 31-889
trigamma, 31-878
trimr, 31-879
trivariate Normal, 31-100
troubleshooting, libraries, 18-12
TRUE, 10-32, 11-10
trunc, 31-880
truncating, 31-880
type, 31-880
typecv, 31-881
typef, 31-883

U

unconditional branching, 10-36
underdetermined, 31-554, 31-559
union, 31-884
unionsa, 31-884
uniqindx, 31-885
uniqindxsa, 31-886
unique, 31-887
uniquesa, 31-888
until, 31-197
upmat, 31-889
upmat1, 31-889
upper triangular matrix, 31-889
upper, 31-890
use, 31-891
user-defined function, 31-448, 31-609
utctodt, 31-892
utctodtv, 31-893
utrisol, 31-894

V

vals, 31-895
varget, 31-896
vargetl, 31-897
variable names, 31-363, 31-364
variance, 31-211, 31-213
variance-covariance matrix, 31-554, 31-559, 31-903, 31-904
varindxi, 31-567
varmall, 31-898
varmares, 31-899
varput, 31-900
varputl, 31-901
vartypef, 31-903
vcm, 31-903
vcms, 31-904
vcx, 31-903
vcxs, 31-904
vec, vecr, 31-905
vech, 31-906
vector (dataloop), 31-907
vectors, 10-41
vget, 31-908
view, 31-908
viewing graphics, 3-2
viewing variables, 6-3
viewxyz, 31-909

vlist, 31-910
vnamecv, 31-910
volume, 31-911
vput, 31-911
vread, 31-912
vtypecv, 31-912

W

wait, 31-913
waitc, 31-913
walkindex, 31-913
watch variables, 6-3
watch window, 5-10
weighted count, 31-145
while, 31-197
window, 20-4
window, 31-915
window, clear, 31-117
workbox, 31-909, 31-911
workspace, 31-181, 31-766
writer, 31-915

X

xlabel, 31-917
xlsGetSheetCount, 31-918
xlsGetSheetSize, 31-918
xlsGetSheetTypes, 31-919
xlsMakeRange, 31-920
xlsreadm, 31-921
xlsreadsa, 31-922
xlsWrite, 31-924
xlsWritem, 31-925
xlswritesa, 31-926
xor, 11-14, 11-15
.xor, 11-15
xpnd, 31-927
xtics, 31-929
xy, 31-930
xyz, 31-930

Y

ylabel, 31-931
ytics, 31-931

Z

zeros, 31-932
zlabel, 31-933
zooming graphs, 24-29
ztics, 31-933