GAUSS™ 3.2 for UNIX

User's Guide

Aptech Systems, Inc.

Part Number: GM-UNIX-PDF
Revision Date: August 9, 2000

Contents

Chapter 1

Installation and Operation

This chapter covers the basics of installing and running **GAUSS** on a UNIX workstation. Please read the entire chapter before beginning. It is short and will save time in the long run.

Please read the `README*` files that came with your shipment! They contain important information that became available after the manual was printed.

1.1 Installing GAUSS

GAUSS is shipped on 3.5″ floppy diskettes, 1/4″ tape or DAT tape. This section gives installation instructions for each. Each shipment also includes a hardcopy **Installation** memo.

SEE THE INSTALLATION MEMO that came with your shipment! Instructions on the memo, where different, override the instructions in this supplement.

If you are unfamiliar with UNIX, see your system administrator or system documentation for information on the system commands referred to below. The device names given are probably correct for your system.

1.1.1 Extracting the Files

Use `mkdir` to create a directory for **GAUSS**.

```
mkdir gauss
```

Use `cd` to make the directory you created the current working directory.

```
cd gauss
```

Use `tar` to extract the files.

`tar xvf` *device_name*

The following device names are suggestions. See your system administrator. If you are using Solaris 2.x, see Section 1.1.2.

Operating System	3.5 inch diskette	1/4 inch tape	DAT tape
Solaris 1.x SPARC	`/dev/rfd0`	`/dev/rst8`	
Solaris 2.x SPARC	`/dev/rfd0a` (vol. mgt. off)	`/dev/rst12`	`/dev/rmt/1l`
Solaris 2.x SPARC	`/vol/dev/aliases/floppy0`	`/dev/rst12`	`/dev/rmt/1l`
Solaris 2.x x86	`/dev/rfd0c` (vol. mgt. off)		`/dev/rmt/1l`
Solaris 2.x x86	`/vol/dev/aliases/floppy0`		`/dev/rmt/1l`
HP-UX	`/dev/rfloppy/c20Ad1s0`		`/dev/rmt/0m`
IBM AIX	`/dev/rfd0`	`/dev/rmt.0`	
SGI IRIX	`/dev/rdsk/fds0d2.3.5hi`		

If **GAUSS** came on diskettes, repeat the `tar` command for each diskette.

1.1.2 Solaris 2.x Volume Management

If Solaris 2.x volume management is running, insert the floppy disk and type:

```
volcheck
```

to signal the system to mount the floppy. Then execute the `tar` command. When the `tar` command finishes type:

```
eject
```

and remove the floppy if it is not automatically ejected.

The floppy device names for Solaris 2.x change when the volume manager is turned off and on. To turn off volume management, become the superuser and type:

```
/etc/init.d/volmgt off
```

To turn on volume management, become the superuser and type:

```
/etc/init.d/volmgt on
```

1.1.3 Uncompressing

If the media was diskette, use the `gunpack` script to uncompress the files:

```
gunpack
```

1.1.4 Configuring GAUSS

GAUSSHOME environment variable

GAUSS needs an environment variable called GAUSSHOME. It should contain the complete name of your **GAUSS** home directory. For example:

C shell

```
setenv GAUSSHOME /home/usr1/gauss
```

Korn, Bourne shell

```
GAUSSHOME=/home/usr1/gauss
export GAUSSHOME
```

GAUSS looks in the $GAUSSHOME directory for its configuration file, `gauss.cfg`. `gauss.cfg` is a commented ASCII file that controls several startup defaults. It can be edited before getting into **GAUSS**, allowing you to change any of the defaults. Anyone who will be running **GAUSS** needs to have at least *read* access to `gauss.cfg`.

The standard **GAUSS** system directory layout is as follows:

`$GAUSSHOME`	Base directory, executable files, configuration files
`$GAUSSHOME/src`	Source files
`$GAUSSHOME/lib`	Library files
`$GAUSSHOME/examples`	Example files
`$GAUSSHOME/wksp`	Workspace files, log files

As it is shipped, `gauss.cfg` assumes this directory structure and uses GAUSSHOME to establish the base directory, so you don't have to edit it unless you change the directory structure.

GAUSS creates temporary files during operation. Anyone who uses **GAUSS** must have *read/write/execute* access to the directories where those files are located. As it is shipped, this is the `$GAUSSHOME/wksp` directory.

You can specify a path to an alternate configuration file with the GAUSS_CFG environment variable. For example:

C shell

```
setenv GAUSS_CFG /home/usr1/gauss/project1
```

Korn, Bourne shell

```
GAUSS_CFG=/home/usr1/gauss/project1
export GAUSS_CFG
```

The name of the file remains the same, i.e., `gauss.cfg`.

You can use the **$()** syntax in `gauss.cfg` to reference environment variables. For example:

```
src_path = $(GAUSSHOME)/src
```

GAUSS uses an internal variable named **gaussdir** in `gauss.cfg`, so do not try to use an environment variable named **gaussdir** in the file.

PATH environment variable

You should place a reference to the $GAUSSHOME directory in your system PATH environment variable, so the operating system can find the **GAUSS** executables.

1.2 Starting GAUSS

UNIX is case sensitive, so the commands below are case sensitive.

To start **GAUSS** in X Window mode, type:

```
gauss &
```

To start **GAUSS** in terminal or vanilla mode, type:

```
gauss -v
```

You must be in a UNIX terminal window such as an **xterm** or **Command Tool**. The graphics and windowing functions are not available in terminal mode.

The **GAUSS** prompt looks like this:

```
(gauss)
```

1.3 Exiting GAUSS

1.3.1 X Window Mode

To end a **GAUSS** session, click on the **Quit** button.

1.3.2 Terminal Mode

If you are in terminal mode, type:

`quit` *enter*

at the **GAUSS** command prompt. See Section 1.6 for ways to stop a **GAUSS** program that is executing.

1.4 Log Files

GAUSS creates an error log file and a command log file for each session. These will need to be deleted periodically. They are generated with unique names containing the process ID of the **GAUSS** session that created them. They are located in the directories specified in the configuration file. The default is `$GAUSSHOME/wksp`.

1.5 Running GAUSS programs

To run a program from the **GAUSS** prompt, type

`run` *filename*

For example:

```
run max1.e
```

Programs to run can be listed on the command line when you start **GAUSS**. At the UNIX command prompt, type

`gauss` *filename* [[*filename...*]]

GAUSS will run the programs specified, and return to the **GAUSS** command line.

1.6 Quitting GAUSS Programs

1.6.1 X Window Mode

Click on **Stop** or **Kill**.

Use **Kill** only when necessary—it will always work, but it literally kills the compiler and executer with no questions asked. It can leave your workspace in an indeterminate state.

1.6.2 Terminal Mode

Esc *enter*

or

Esc! *enter*

or

`stop` *enter*

or

Ctrl-C

or

`kill` *enter*

You will be returned to the **GAUSS** command line.

If you have programs that expect to pick up **Esc** with the **key** function, you can turn the trapping of **Esc** off in the configuration file. Either way, "**Esc!**" is active to stop a program.

Use **kill** only when necessary, see above reference under X Window mode.

1.7 Editing Files

For the time being, you will have to use your own text editor to write **GAUSS** programs. Several come with UNIX if you don't have one of your own. You can specify a hook to an editor in the configuration file (see `gauss.cfg` for instructions). The default editor is `vi` which is started by typing

```
edit myfile
```

from the **GAUSS** prompt.

1.8 Command Line Flags

There are several flags you can specify on the command line when starting **GAUSS**. The following flags are supported:

-b	run **GAUSS** in batch mode, look for keyboard input.
-B	run **GAUSS** in batch mode, don't look for keyboard input. Use this when you want to run **GAUSS** in the background.
-m	turn off make.
-M	turn on make.
-n	do an implicit **new** before each program. Ignored if not in batch mode.
-s	run **stdin** as a **GAUSS** program. During the execution of the program, no keyboard input will be recognized. This flag cannot be combined with others behind the same '–', and it must come after any other flags.
-T	turn the translator on.
-t	turn the translator off.
-v	terminal or vanilla mode.

You can run **GAUSS** in batch mode. If your programs require user input, use the **-b** flag. At the UNIX command prompt, type

`gauss -b` *filename* [[*filename...*]]

GAUSS will start up in batch mode, run the programs specified, then drop to the UNIX command prompt. You can break the run with **Ctrl-C**, **Esc**, etc. (see below); **GAUSS** will drop to command mode.

If your programs don't require user input, use the **-B** flag. At the UNIX command prompt, type

`gauss -B` *filename* [[*filename...*]]

or

`gauss -B` *filename* [[*filename...*]] `&`

GAUSS will start up in batch mode, run the programs specified, then drop to the system. You can break the run with **Ctrl-C**; **GAUSS** will drop to the **GAUSS** prompt. All other keyboard input is ignored, including **Esc**. When you want to run **GAUSS** in the background, use **-B**.

If you want to run **stdin** as a **GAUSS** program, use the **-s** flag. At the UNIX command prompt, type

`gauss` [[*filename...*]] `-s` [[*filename...*]]

Other programs can be run at the same time as **stdin**. **GAUSS** will start up in batch mode, run the programs and **stdin** in the order listed, then drop to the system. The **-s** flag forces **GAUSS** into the same mode as the **-B** flag; all keyboard input is ignored except for **Ctrl-C**.

The **-T** and **-t** flags turn the translator on and off respectively, overriding the translator setting in `gauss.cfg`.

The **-n** flag is used to execute a **new** command between programs to prevent any symbol conflicts if the programs use variables with the same names.

When the -M flag is on if you edit a file that contains symbols in use the file will be recompiled.

1.9 Command Filtering

By default, when **GAUSS** is in terminal mode, most of the following commands are available interactively at any time, even during compilation or execution of a **GAUSS** program:

cancel	cancel any pending commands
stop	stop the current program politely
Esc, *Esc*!	stop the current program politely
kill	terminate current program immediately
quit	terminate **GAUSS** and return to the OS
status	list status, current command, pending commands
translator on|**off**	turn data loop translator on or off
config	set compile-time and run-time options
help	on-line help
dos	exec a command shell
[[*editor alias*]]	exec an editor (see `gauss.cfg`)
!!	repeat previous command
debug	run program under debugger

This is done by filtering them out of the keyboard input stream and handling them immediately, rather than putting them in the compiler command queue. This can give you unexpected results, e.g., if you type "status" as the input to a **cons** statement. This filtering can be turned off during program execution. You can specify the default command filtering mode for all **GAUSS** sessions by setting the **cmd_filter** parameter in the configuration file, or use the **config** command (run options) to set the default filtering mode for a single **GAUSS** session. You can also use the **#filteron** and **#filteroff** compiler directives to set the command filtering mode for a single program run. **#filteron** and **#filteroff** must be placed at the top of your program.

stop and **dos** can be used in **GAUSS** programs; the rest of the commands can only be used interactively.

1.10 Running Programs Written for the PC

In general, you should be able to run **GAUSS** programs that were originally written for the PC without any problems. The main limitations relate to the advanced hardware control commands.

The coprocessor control commands—**enable**, **disable**, etc.—are not available. All coprocessor exceptions are currently masked.

The flags for the **lib** command are specified with a dash ("–") instead of a slash ("/"). For example:

```
lib gauss -n;
```

This is because the slash is a separator in UNIX file specifications.

The **new** command cannot change the main program space size or the number of global symbols available. These settings can be specified in `gauss.cfg`.

If you're working in a **GAUSS** X window, keyboard input works the same as on PC's. If you're working in terminal mode, however, it doesn't. Because of the way the system buffers input, **GAUSS** doesn't "see" any keyboard input until you hit **Enter**. So, if you are using the **key** command for single key input, you will have to enter

keystroke enter

If you are using **cons** or the **key** command for multiple key input, you can type in your entire input, then hit **Enter**, and it will all be available to **GAUSS**.

If you want to input a carriage return, press **Enter** when there is no pending input, and **GAUSS** will recognize it as input.

Programs written for the PC are generally written with an 80-character wide screen in mind. So, if you're running under an X window manager, you should size your windows to at least 80 columns to guarantee that your program output doesn't wrap.

1.11 atog

atog for UNIX has no interactive mode, and can only be run from within **GAUSS** by using the **shell** command, i.e., **shell atog**. Execute **atog** without command line arguments for a help screen. For general instructions on using **atog**, see the *Utility: atog* chapter in Volume I.

1.12 File Handling

Files are created with the following permissions:

Owner	**read/write**
Group	**read**
World	**read**

This can be controlled from `gauss.cfg`.

1.12.1 Data Sets, Matrix Files and String Files

There are new formats available for data sets, matrix files and string files. See the *File I/O* chapter in Volume I of the manual.

Data sets now consist of one file with a `.dat` extension, not two files with `.dht` and `.dat` extensions.

The header has changed for matrix and string files—otherwise, they're the same as before.

We have a new utility, **transdat**, for converting these files from one format to another. If you have data sets, matrix files or string files you want to transfer from your PC to the workstation, you can use **transdat** to convert them. **transdat** must be run from the UNIX prompt. Execute **transdat** with no arguments for a help screen. See G for details.

1.13 Cache Size

There is a line for cache size in the configuration file. Set it to the size of the CPU data cache for your machine. The default is 32K.

This affects the choice of algorithms used for certain math functions. This can dramatically affect performance for large matrices. Results will be the same.

1.14 Graphics Configuration File

The name and format of the graphics configuration file have been changed. The file is now called `.gaussrc` instead of `pqgrun.cfg`; formatwise, it is a standard X Windows resource file. **GAUSS** searches for `.gaussrc` in the following directories, in the order listed:

1. The directory specified by GAUSS_CFG, if it is defined
2. The user's home directory
3. The $GAUSSHOME directory
4. The current working directory

The first one found is used. If none is found, **GAUSS** creates a default `.gaussrc` (using internal defaults) in the user's home directory.

The following resources can be included in `.gaussrc`:

Resource

```
gauss.pqg.print.margin.left
gauss.pqg.print.margin.right
gauss.pqg.print.margin.top
gauss.pqg.print.margin.bottom
```

Definition Left/right/top/bottom print margin.

Options Any nonnegative decimal value. Interpreted as inches.

Default 1.00"

Resource

```
gauss.pqg.print.resolution
```

Definition Print resolution.

Options MinRes, LoRes, HighRes, MaxRes

Default MaxRes

Resource

```
gauss.pqg.print.filename
```

Definition Output filename when printing to file or converting graph.

Options String, any valid filename. File must be writeable.

Default none

Resource

```
gauss.pqg.print.placement
```

Definition Placement of graph within margins. Affects fixed aspect graphs only.

Options TopLeft, Top, TopRight, Left, Center, Right, BotLeft, Bottom, BotRight

Default Center

■ Resource

`gauss.pqg.print.orientation`

Definition Orientation of graph on page.

Options Landscape, Portrait

Default Landscape

■ Resource

`gauss.pqg.print.dest`

Definition Destination of graphics printout.

Options Printer, File

Default Printer

■ Resource

`gauss.pqg.print.aspect`

Definition Aspect ratio of printed graph. Fixed aspect graphs are printed within the limits defined by the margins, whereas variable aspect graphs are stretched to fit the margins.

Options Fixed, Variable

Default Fixed

■ Resource

`gauss.pqg.print.driver`

Definition Driver used to convert graph to format suitable for your printer.

Options PS, EPS, HPLJET3, HPGL, PIC

The file `prndev.tbl` in the $GAUSSHOME directory lists additional printer/file types that are available but not tested. You can try any of the types listed there except the type 4 entries (i.e., those with a 4 in the second field), which are NOT supported.

Default PS

Resource

`gauss.pqg.print.command`

Definition Command that will be executed to print a graph.

Options String, any valid print command.

Default lp -c $FILE

Resource

`gauss.pqg.print.color`

Definition Print color mode: black/white, color, or gray scale.

Options BW, RGB, Gray

Default BW

Resource

```
hpgl.pen.black
hpgl.pen.dark_grey
hpgl.pen.red
hpgl.pen.light_red
hpgl.pen.brown
hpgl.pen.light_brown
hpgl.pen.green
hpgl.pen.light_green
hpgl.pen.cyan
hpgl.pen.light_cyan
hpgl.pen.blue
hpgl.pen.light_blue
hpgl.pen.magenta
hpgl.pen.light_magenta
hpgl.pen.grey
hpgl.pen.white
```

Definition Pen color for HPGL-format output.

Options Integer, any valid pen number.

Default

`hpgl.pen.black`	1
`hpgl.pen.dark_grey`	1
`hpgl.pen.red`	5
`hpgl.pen.light_red`	5
`hpgl.pen.brown`	7
`hpgl.pen.light_brown`	7
`hpgl.pen.green`	3
`hpgl.pen.`	3
`hpgl.pen.cyan`	4
`hpgl.pen.light_cyan`	4
`hpgl.pen.blue`	2
`hpgl.pen.light_blue`	2
`hpgl.pen.magenta`	6
`hpgl.pen.light_magenta`	6
`hpgl.pen.grey`	8
`hpgl.pen.white`	8

You may also see the following entries in `.gaussrc`:

`gauss.pqg.print.size`: 100%
`gauss.pqg.print.scaling`: ByPage

These are reserved for future development.

The **WinPrintPQG** command now works. Also, its functionality has been expanded to include that of **WinConvertPQG**, and **WinConvertPQG** has been dropped. See the on-line help for details on how to use **WinPrintPQG**–it has a different syntax now.

You can now zoom multiple times in a graph.

Default Fonts

You can specify default normal and bold fonts for **GAUSS** to use in Text and TTY windows by including `gauss.font.normal` and `gauss.font.bold` resources in the `.gaussrc` file. Set them to any valid nonproportional font specification string.

Chapter 2

Tutorial

This tutorial will introduce you to the basics of **GAUSS**.

2.1 Loading and Exiting GAUSS

This lesson covers starting, stopping and suspending **GAUSS**.

2.1.1 Starting GAUSS

From the OS (operating system) prompt there are two ways to start **GAUSS**. The first:

```
gauss &
```

will start up **GAUSS** in an X window. The second:

```
gauss -v
```

will start **GAUSS** in text mode in your current terminal.

2.1.2 Exiting GAUSS

To exit **GAUSS** when in X window mode, click on the **Quit** button. To exit **GAUSS** in terminal mode, at the `(gauss)` prompt type:

```
quit
```

2.2 Matrices

This lesson covers the basic commands that create matrices.

2.2.1 The let Statement

Type this in and press **Enter**:

```
let x = { 1 2 3, 4 5 6, 7 8 9 };
```

You just created a 3x3 matrix called **x**. To print it out type:

```
print x;
```

Try it again without the **let**.

```
x = { 1 2 3, 4 5 6, 7 8 9 };
```

An assignment statement with the data enclosed in curly braces is an implicit **let** statement. **let** statements are run-time initialization statements. Only a list of constants is allowed to the right of the equal sign in a **let** statement.

Now let's create a column vector and print it out.

```
x = { 1,2,3,4,5,6,7,8,9 };
print x;
```

Now a row vector.

```
x = { 1 2 3 4 };
print x;
```

The constants in a **let** statement can be real or complex. Try the following:

```
x = { 1 2+3 4 5-6i 7 8i };
print x;
```

Complex constants can be entered in one of two ways. If a number has both a real and an imaginary part, just join the two with the sign (+ or -). If a number has no real part, append an "i" to it, and **GAUSS** will recognize it as imaginary. Notice that the "i" can be appended even if the number has a real part.

2.2.2 Concatenation

The concatenation operators are used to create matrices from other matrices or from expressions. Enter:

```
a = { 1.1  3.2 };
b = { 2.3  -4.5 };
```

Now try these statements:

```
hc = a~b;
print hc;

hcp = a+1~b+2;
print hcp;

vc = a|b;
print vc;
```

The $\sim$ is the horizontal concatenation operator, and the `|` is the vertical concatenation operator.

2.2.3 Special Matrix Functions

To create a matrix of zeros or ones, or random numbers, try the following statements.

Zeros

```
x = zeros(4,4);
print x;
```

Ones

```
y  = ones(2,4);
print y;
```

Uniform Random Numbers

```
z = rndu(6,3);
print z;
```

Normal Random Numbers

```
z = rndn(6,3);
print z;
```

Random Integers

```
rndint = ceil( rndu(6,4) * 100 );
print rndint;
```

Constant Matrix

To create a constant matrix, all you need to do is add the desired constant to a matrix of zeros. For example, to create a 6x3 matrix of 7's, this statement will work:

```
sevens = 7 + zeros(6,3);
print sevens;
```

Identity Matrix

```
id = eye(3);
print id;
```

Additive Sequence

```
t1 = seqa(0,0.1,10);
print t1;

t2 = seqa(pi/4,pi/4,4);
print t2;
```

Multiplicative Sequence

```
t1 = seqm(2,2,10);
print t1;
```

Reshaping Matrices

```
x = seqa(1,1,3);
y = reshape(x,5,3);
print x;
print y;
```

2.2.4 Keyboard Input

The **con** function requests keyboard input of a matrix. Type this in.

```
x = con(3,3);
```

The **?** is your prompt. The **con** function is requesting 9 numbers. Type in 9 numbers, each followed by a space.

Are you back at the `(gauss)` prompt? If so, print your new matrix **x**.

The **con** function can accept complex numbers. It uses the same rules for entering complex numbers as the **let** statement. Try entering some.

```
x = con(2,2);
print x;
```

Notice that the **con** function, like the **let** statement, only understands constants.

2.3 Strings

We will cover two ways to create a string.

2.3.1 String Constant

```
s = "This is a string.";
```

The double quotes enclose a string constant. Printing a string is like printing a matrix. Try it.

2.3.2 Keyboard Input

The **cons** function requests keyboard input of a string. Type this in.

```
s = cons;
```

There is no visible prompt. Type something in and press **Enter**.

You should be back to the **GAUSS** prompt. Print **s**.

Now let's do it with a prompt so we can tell when we are being asked to type something in.

```
print /flush "Enter the string: ";; s = cons;
```

Enter another string, then print **s**. The double semicolons at the end of the **print** statement suppressed a line feed afterward so the **cons** statement would start requesting input on the same line. The **/flush** flag forces immediate display of the prompt.

2.3.3 String Concatenation

Let's tack two strings together. Type this in.

```
s = s $+ "catfish";
print s;
```

The **$+** is the string addition or concatenation operator.

2.4 Matrix Description and Printing

This is a short section on using some of the functions that *describe* matrices. You will learn how to find the number of rows and columns in a matrix and find the largest and smallest elements in a particular column or in the entire matrix. Also, you will be introduced to some more features of the **print** statement.

To find the number of rows and columns in a matrix, use the **rows** and **cols** functions. Type in the following statements:

```
x = rndu(10,4);
r = rows(x);
c = cols(x);
print "Matrix x is: "   r   " by "   c;
```

The **print** statement could have been written:

```
print "Matrix x is :"   rows(x)   " by "   cols(x);
```

To find the maximum and minimum elements of each column of a matrix, use the **maxc** and **minc** functions. Try this:

```
print x;
print maxc(x);
print minc(x);
```

To find the largest or smallest element in an entire matrix, call **maxc** and **minc** twice:

```
print "Maximum is: "   maxc(maxc(x));
print "Minimum is: "   minc(minc(x));
```


2.4.1 The print Statement

In entering the statement

```
print "Matrix x is: "   r   " by "   c;
```

you probably noticed that the statement combined the printing of string constants (the phrases between double quotes) and variables (**r** and **c**). **GAUSS** uses a space to separate the individual expressions in a **print** statement. Therefore, extraneous spaces are illegal in **print** statements. Try:

```
print 3 + 5;
```

The error you will get is:

```
(0) : error G0064 : Operand missing
```

GAUSS translated that as three expressions. The middle expression was the addition operator with no operands. There are two ways to write the above statement correctly:

1. No extraneous spaces.

   ```
   print 3+5;
   ```

2. Parentheses.

   ```
   print (3 + 5);
   ```

2.4.2 The format Statement

If you are still using the default print format, then the statements:

```
print "Rows: "   r;
print "Cols: "   c;
```

print the number of rows and columns with 8 digits of precision.

Create a variable containing an integer value.

```
x = 5;
```

To change the print format, enter the following **format** statement:

```
format /rdn 1,0;
```

The **/rdn** flag indicates that numbers are right-justified in decimal format followed by nothing. The **1** sets the field width, which will be overridden if the number requires more space, and the **0** sets the precision or number of places to the right of the decimal point. This **format** statement is suitable for printing integers.

```
print "(" x ")";
```

Let's leave out the **n** and see what happens, type:

```
format /rd 1,0;
```

The **/rd** flag indicates that numbers are right-justified in decimal format followed by a space, which is the default.

```
print "(" x ")";
```

As you can see, the number is now followed by a single space character. Try another format and print **x** again.

```
format /len 12,4;
```

Any numbers printed after this **format** statement will be left-justified using scientific notation in a 12-character field followed by nothing with 4 decimal places of precision.

Type in the following:

```
let x = 1+2 3-4;
format /rds 4,2;
print x;
```

For complex numbers, the field and precision settings refer to the field width and precision for *each* part of the number. Also, when the number is printed, the real and imaginary parts are separated by a sign (" + " or " - "), and an "i" is appended to the imaginary part. You will need to figure this in when you want to print complex numbers in a nice format.

Let's change **x** a bit.

```
let x = { 1+2 3-4, 5 6i };
print x;
```

When the imaginary part of a complex number is 0, it is not printed at all. We do, however, pad the place where the 0 would appear with spaces, to keep the rest of the printout aligned.

Now reset the format to one more like the default format.

```
format /ro 12,4;
```

2.5 Indexing Matrices and Extracting Submatrices

This section describes how to index individual elements of a matrix and extract submatrices of a matrix. First clear your workspace of all previously defined globals. To do this, enter

```
new;
```

Now create a matrix using **seqa** and **reshape**:

```
x = reshape( seqa(1,1,32), 8, 4 );
print x;
```

2.5.1 Indexing

Now, let's extract the first and third columns of **x**:

```
col1 = x[.,1];
print col1;
```

```
col3 = x[.,3];
print col3;
```

Now, extract a row, whose index might be variable:

```
i = 2;
row = x[i,.];
print row;
```

Now, index a set of rows. I will leave out the **print** statements from here on, except when necessary. You can print whatever you want to take a look at.

```
idx = { 1 3 6 8 };
ridx = x[idx,.];
```

Finally, index a set of rows and columns:

```
rowidx = { 1 3 5 8 };
colidx = { 3 4 };
newmat = x[rowidx,colidx];
```

The matrix **newmat** consists of the intersection of rows 1, 3, 5, and 8 and columns 3 and 4.

```
newmat = x[1 3:6,2:4]
```

The matrix **newmat** now consists of the intersection of rows 1, 3-6 and columns 2-4.

The **submat** function can also be used to extract submatrices.

```
r = { 1 3 5 };
c = { 3 4 };
newmat = submat(x,r,c);
```

This will index rows 1, 3 and 5 and columns 3 and 4.

2.5.2 Using delif and selif

You may want to create a submatrix based on some selection or deletion criteria. This section will show you one way to do this, using the **selif** and **delif** functions.

First, create a matrix of normally distributed random numbers:

```
y = rndn(100,4);
```

Now we will create a new matrix, which contains only those rows of **y** where column 2 is greater than 0.5. To do this, we will create a vector of 1's and 0's that has a 1 in each row we want to select.

```
ex = y[.,2] .> 0.5;
print ex;
```

The vector **ex** should have a 1 wherever the corresponding entry in the second column of **y** is greater than 0.5. The expression `y1[.,2] .> 0.5` uses a *dot operator* which returns a matrix of 1's and 0's with a 1 wherever the expression is true and a 0 wherever it is false. The result of the expression above is a vector with the same number of rows as **y**. We'll use such a vector of 1's and 0's in the **selif** and **delif** statements.

Now, we want to create a submatrix of the matrix **y** containing all the rows in which the second column is greater than 0.5. Here is how we would use the **selif** function to create this submatrix:

```
suby = selif(y,ex);
```

Print **suby** to see that column 2 contains only numbers greater than 0.5.

Here is another example of a logical expression that can be used with the **selif** function.

```
ex2 = abs( y[.,1] ) .< 0.33 .and y[.,2] .> 0.66;
```

The **delif** function works in a complementary fashion. Instead of *selecting* elements to go in the submatrix, you *delete* those elements that you don't want. For example:

```
ex = y[.,1] .< 0.33;
v = delif(y,ex);
```

Of course, this would be equivalent to this set of statements which use **selif**:

```
ex  = y[.,1] .>= 0.33;
v = selif(y,ex);
```

The logical expression may almost always be nested inside of the call to the functions that require them. The above two statements could be combined into one statement:

```
v = selif(y,y[.,1] .>= 0.33);
```

2.6 Running a Program

Up until nowwe have been executing single line commands. Now let's write a program. Create a file called `lesson1` and type this into the new file.

```
A = rndn(3,3);
b = rndn(3,1);
x = b/A;
print A;
print b;
print x;
```

Save your file without exiting your editor and at the (gauss) prompt type:

```
run lesson1;
```

from the **GAUSS** prompt.

You just solved a system of linear equations. This is an example of the / (slash) operator. It is quite powerful.

The / operator is supposed to solve the linear system

$$Ax = b$$

Let's check it from the command line. If the equation above is true, **A*x-b** should be zeros.

```
print A*x;
print b;
print A*x-b;
```

It's pretty close, isn't it? There will usually be a little rounding error when doing calculations on real numbers.

Let's try the / operator on a different kind of problem. Edit your program so that the whole thing looks like this:

```
x = { 1 2 3,
      4 5 6,
      7 8 9 };
y = 2;
z = x / y;
print x;
print;
print y;
print z;
```

The empty **print** statement just prints a carriage return/line feed.

Execute your new program.

The / operator is now doing standard division. You should have gotten:

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

2

 0.5   1  1.5
   2 2.5    3
 3.5   4  4.5
```


2.7 Element-by-Element Operators

Now get back into your program and change the second and third lines so the program looks like this:

```
x = { 1 2 3,
      4 5 6,
      7 8 9 };
y = { 1 2 3 };    /* row vector */
z = x ./ y;       /* dot slash operator */
print x;
print;
print y;
print z;
```

The `/* row vector */` statement is a comment. Comments don't do anything but you can use them to make your programs easier to understand when you come back to them six months later.

Execute your new program. You should have gotten:

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

 1  2  3

 1   1  1
 4 2.5  2
 7   4  3
```

This is element-by-element division. The dot slash always does element-by-element division. This is an example of what we refer to as element-by-element conformability. The row vector **y** was swept down the matrix **x** and the corresponding divisions were done.

Get back into the file.

Make it look like this:

```
x = { 1 2 3,
      4 5 6,
      7 8 9 };
y = { 1, 2, 3 };  /* column vector */
z = x ./ y;       /* use dot slash */
print x;
print y;
print z;
```

Execute it. You will get:

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

1
2
3

    1      2  3
    2    2.5  3
2.333  2.667  3
```

Here the column vector was swept across the matrix.

Now get out of **GAUSS** by typing **quit** or clicking on the **Quit** button. Get back into **GAUSS** and run your program at the same time by typing either:

```
gauss lesson1
```

or

```
gauss -v lesson1
```

Let's make one final change to your file. Add two lines to the program so it looks like:

```
x = { 1 2 3,
      4 5 6,
      7 8 9 };
x = complex(x,1);   /* create complex matrix */
y = { 1, 2, 3 };            /* column vector */
y = complex(y,1);   /* create complex matrix */
z = x ./ y;                 /* use dot slash */
format /rz 4,2;     /* suitable format for x, y and z */
print x;
print y;
print z;
```


Run the program. The output should look like this:

$$\begin{matrix} 1 + 1i & 2 + 1i & 3 + 1i \\ 4 + 1i & 5 + 1i & 6 + 1i \\ 7 + 1i & 8 + 1i & 9 + 1i \end{matrix}$$

$$\begin{matrix} 1 + 1i \\ 2 + 1i \\ 3 + 1i \end{matrix}$$

$$\begin{matrix} 1 & 1.5 - 0.5i & 2 - 1i \\ 1.8 - 0.4i & 2.2 - 0.6i & 2.6 - 0.8i \\ 2.2 - 0.4i & 2.5 - 0.5i & 2.8 - 0.6i \end{matrix}$$

This is an example of element-by-element complex division. Try using the **.*** element-by-element multiplication operator to check the answer you just got. You can do this from the command line.

```
a = z .* y;
print a;
```

a equals **x** (except for a little round-off error). Can you figure out why **x ./ z** doesn't equal **y**?

2.8 Creating a Data Set

We will now create a small data set.

2.8.1 Reading an ASCII File

First we will create a small ASCII data file. In your editor create a file named `rawdata`. Type in the following numbers.

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

Save `rawdata`. We can now load `rawdata` into a matrix.

```
load x[6,4] = rawdata;
print x;
```

This is the quick and dirty way to use **load** for small ASCII files. See **load** in the *COMMAND REFERENCE* section for a way to use it that has more error checking. See the *Utility: atog* chapter in Volume I of the manual for converting larger files.

To create a small **GAUSS** data set from the matrix **x**, we will use **saved**.

```
call saved(x,"mydata",0);
```

This statement created a data set called `mydata.dat`. Check to see if it was created.

If the third argument to **saved** is 0, the names for the columns of the data set will all start with **X**. People using **GAUSS** for statistical analysis will use these for the names of their variables. Each row of the data set is then a case or observation.

We can now get some statistics on this data set.

```
call dstat("mydata",0); call ols("mydata",0,0);
```

If you want to see what the arguments to the **ols** or **dstat** procedures mean, click on the **Help** button or type **help** at the **GAUSS** prompt. You will be asked for the topic you want help on. Answer with **ols** or **dstat**. Press the capitalized letter of the desired option to page up and down, etc., in the file. Type **q** *enter* to exit the help system.

This on-line help is available for your own functions. Let's make one.

2.9 Writing a Procedure

Create a file named `sumd.src` in your text editor.

We will create a procedure that sums the columns of a data set and returns a vector of the sums. We will go a line or two at a time and I will explain each statement.

```
proc sumd(name);
```

This is the beginning of a procedure definition. This is a user-defined function. It will return 1 item. That is the default. It will take one argument which will be the name of the data set.

```
    local fp,sum;
```

These variables, **fp** and **sum**, are local variables. They only exist while this procedure is being called.

```
    open fp = ^name;
```

The **open** command opens a file. The argument, **name**, will be a string variable. The other variable, **fp**, is going to be used as a file handle. Most file-related functions in **GAUSS** take a file handle to identify which file you want to work on. The ^ (caret) operator in the context of the **open** statement means that what follows is a string variable and we need the contents of it.

```
    if fp == -1;
        errorlog "Can't open the file";
        end;
    endif;
```

The comparison operator is == instead of just one equal sign. We are testing to see if the file handle is a -1 because the **open** command will set it to -1 if it can't open the file. **errorlog** prints an error message to the screen and the error log file. If the file cannot be opened, the program will end.

```
    sum = 0;
```

We need to zero out the variable to contain the sums.

```
    do until eof(fp);
```

We are going to start a loop. We will keep doing this loop until we get to the end of the file.

```
        sum = sum + sumc(readr(fp,50));
```

This line calls **readr** which will read 50 rows from the data set we opened under the handle **fp**. The result of this read is passed to **sumc** which will return the sums of each column. These sums are added to **sum** on each iteration of the loop. If there aren't 50 rows left, then **readr** will read however many there are.

```
    endo;
```

This is the end of the loop.

```
    fp = close(fp);
```

Now we close the file, again using the file handle.

```
    retp(sum);
```

The **retp** statement is used to return from the procedure. The vector of column sums is returned.

```
endp;
```

The **endp** statement is the close of the procedure definition. Here is our complete procedure.

```
proc sumd(name);
    local fp,sum;
    open fp = ^name;
    if fp == -1;
        errorlog "Can't open the file";
        end;
    endif;
    sum = 0;
    do until eof(fp);
        sum = sum + sumc(readr(fp,50));
    endo;
    fp = close(fp);
    retp(sum);
endp;
```

Save the procedure without exiting your editor as before.

We need to do one more thing to make this procedure available to **GAUSS**. Type this in at the **GAUSS** command prompt:

```
lib user sumd.src
```

This adds a listing for all procedures and global variables in `sumd.src` (there's only one) to the default library `user.lcg`. Now **GAUSS** knows where to find the procedure **sumd** when you call it.

sumd is also available to the help system. Click on **Help** (in terminal mode, type **help** at the **GAUSS** prompt) and answer **sumd** at the help prompt.

This is how to get on-line help for your own functions. If you put a comment at the top which explains what it is used for, your **sumd** procedure will be complete. I'll leave that for you to do later if you think **sumd** would be a useful function to have around. If not, you can delete the entry for it from the **user** library. For now, let's try to use it on our data set. Type **q** *enter* to exit the help system.

```
sum = sumd("mydata");
print sum;
```

Since this returns a column vector, if you want the sum of the entire data set you could use:

```
sum = sumc(sumd("mydata"));
print sum;
```

These procedures work just like any other function. This makes the **GAUSS** language easy to extend to accommodate the routines specific to your profession.

2.10 Vectorizing for Speed

In this lesson, we will introduce you to the looping statements, and then discuss when loops should and should not be used in **GAUSS**.

First, in your editor create a file called `lesson2`. Now type in the following program.

```
x = 1;
do until x+1 == 1;
    eps = x;
    x = x/2;
endo;

format /rz 1,8;
print "Machine epsilon: " eps;
```

This program finds the smallest number **eps** such that

$$eps + 1 > 1$$

Now suppose you want to recode a vector, based on which elements fall into specific categories.

Create a file called `lesson3`.

Next, create the vector you might want to recode:

```
r = 20;
v = ceil(100*rndu(r,1))/10;
```

Now, suppose you want to assign to a new vector, **v1**, a 1 if the corresponding element in **v** is less than 3, a 2 if the corresponding element in **v** is between 3 and 7, and a 3 otherwise. You could do so with this loop:

```
v1 = zeros(r,1);
i = 1;
do until i > r;
    if v[i] < 3;
        v1[i] = 1;
    elseif v[i] > 7;
        v1[i] = 3;
    else;
        v1[i] = 2;
    endif;
    i = i + 1;
endo;

format /rd 8,1;
print v~v1;
```

This program illustrates the use of the **do** loop and the **if** statement for conditional branching. Also, notice that in order to index into the vector **v1**, we had to first initialize it. In this program, we initialized it as a vector of zeros.

This isn't a particularly efficient way to accomplish the desired task, however. Tasks like these can be *vectorized*. This one in particular can be written in one line, using logical expressions.

Get back into the program. Now add the following statement to your program:

```
v2 = (v .< 3) +
     ( 3 .<= v .and v .<= 7 ) * 2 +
     (v .> 7) * 3;
```

Remember that the dot operators create a vector or matrix of 1's and 0's.

Execute this program. Now, in the command line, we can check to make sure that **v1** and **v2** are identical. From the command line, type:

```
maxc(abs(v1-v2));
```

This will return the maximum difference between any of the corresponding elements in **v1** and **v2**. This difference should be 0.

Finally, let's time the different procedures for recoding our vector **v**. To time a segment of code, we'll use the **hsec** function. Here is how you can time a segment of code:

```
et = hsec;              /* Start timer */

/* Segment of code to be timed here */

et = (hsec - et)/100;  /* Stop timer, convert to seconds  */
```

The function **hsec** uses your computer's clock to return the hundredths of seconds that have passed since midnight. Add timer statements and change the value of **r** to 1000 at the top of your program, so that `lesson3` looks like:

```
r = 1000;
v = ceil(100*rndu(r,1))/10;

et1 = hsec;
v1 = zeros(r,1);
i = 1;
do until i > r;
    if v[i] < 3;
        v1[i] = 1;
    elseif v[i] > 7;
        v1[i] = 3;
    else;
        v1[i] = 2;
    endif;
    i = i + 1;
endo;
et1 = (hsec - et1)/100;

et2 = hsec;
v2 = (v .< 3) +
     ( 3 .<= v .and v .<= 7 ) * 2 +
     (v .> 7) * 3;
et2 = (hsec - et2)/100;
```

```
format /rd 6,2;
print "Loop:       " et1;
print "Vectorized: " et2;
print "Ratio L/V:  " et1/et2;
```

The vectorized process should be over 10 times as fast as the loop. In general, any time you can vectorize a process, do so. The vectorized process will usually be substantially faster.

Another function you can use to recode data in the way we did above is **subscat**. See the *COMMAND REFERENCE* for details on using this function.

Chapter 3

Windows

The current release of **GAUSS** for UNIX includes a simple programming interface to Version 11 of the X Window System. It provides a wide variety of new window support functions, allowing you to easily develop X Window compatible **GAUSS** programs. Also, many previously existing commands have been modified to work within the new window system.

Three types of **GAUSS** windows are currently supported:

- PQG Windows - graphics windows for display of Publication Quality Graphics
- Text Windows - text I/O windows with fixed-sized rectangular buffers
- TTY Windows - text I/O windows with scrolling output history logs

These window types share certain common properties (each can be opened, closed, moved, resized, panned, cleared, and refreshed), but they differ greatly in purpose and functionality. (Many of the new **GAUSS** window functions are defined for a specific window type only, and will generate errors if passed the handle of an incorrect window type.)

This chapter is divided into five sections. The first section contains an overview of general window operations. In the next three sections, each window type is examined in more detail. Section five describes the command and help windows, and the concept of the active window and its role in **GAUSS** window programs.

3.1 General Window Operations

The following operations are basic window control functions, and can be done on any window type. The details of the function calls vary by window type. The most

recognizable difference being that arguments for PQG windows are usually given in pixels, while arguments for Text and TTY windows are usually given in rows and columns. Several operations refer to the special purpose windows; see Section 3.5 for definitions of those.

3.1.1 Opening a Window

WinOpenPQG opens (creates) a PQG window, **WinOpenText** opens a Text window, **WinOpenTTY** opens a TTY window. See Sections 3.2.1, 3.3.1 and 3.4.1 for information on window attributes and defaults.

3.1.2 Moving a Window

WinMove repositions a window on the screen. You can move a window partially or completely off-screen. Arguments are in pixels. Nonintegral arguments are truncated.

3.1.3 Resizing a Window

WinResize resizes a window. There are no limits for sizing PQG windows or TTY windows, but Text windows are limited by the size of their text buffer. You can resize a Text window smaller than its buffer, but not larger. For PQG windows, arguments are in pixels; for Text windows and TTY windows, arguments are in rows and columns. Nonintegral arguments are truncated.

You can specify a window as nonresizeable when you open it. **WinResize** will return a failure code on a nonresizeable window.

3.1.4 Panning a Window

WinPan pans through the contents of a window. For PQG windows, arguments are in pixels; for Text and TTY windows, arguments are in rows and columns. Nonintegral arguments are truncated.

WinPan is not supported for PQG windows.

For Text windows, panning is limited by the buffer boundaries.

For TTY windows, you cannot pan above the first line or to the left of the first column. You can pan the current output line to the top of the window (this is how **cls** and **WinClear** clear TTY windows), and you can pan to the right as far as you like.

3.1.5 Clearing a Window

WinClear clears a specified window, **cls** clears the active window. Windows are cleared to the background color. For PQG windows, the graphics file is truncated to zero length; For Text windows, the window buffer is cleared; for TTY windows, the current output line is merely panned to the top of the window. To clear the output log of a TTY window, use **WinClearTTYLog**.

WinClearArea clears a specified region of a Text window to the background color. Arguments are in rows and columns. As with **Winclear**, that region of the window buffer is cleared. **WinClearArea** is only supported for Text windows.

Closely related to **WinClearArea**, the **scroll** function allows you to scroll a region of text in the active window. The area the text is scrolled away from is cleared to a specified color. **scroll** is only supported for Text windows.

3.1.6 Closing a Window

WinClose closes (destroys) a window. **WinCloseAll** closes all windows except window 1 (the command window).

Closing a window does not mean iconizing it; closing a window destroys it. Once a window is closed, there is no way to retrieve any information it contained.

3.1.7 Selecting a Font

Before using a font, it must be loaded from the system. When you are done with it you should unload it, to conserve memory resources. **FontLoad** loads a font, **FontUnload** unloads it.

Once a font is loaded, you have to let **GAUSS** know you want to use it for a particular window. You can set the font either upon opening a window or when it is the active window. **font** sets the font for the active window.

Changing the font in a window will generally result in a change in the number of rows and columns displayed. Changing to a smaller font in a Text window can result in the window being down-sized, since the window cannot be larger than its buffer.

Setting the font of a PQG window, while not supported, will not cause an error.

3.1.8 Getting Input

The active window always has the keyboard focus in a **GAUSS** program. **con**, **cons**, **key**, **keyw**, **wait** and **waitc** all get input from the active window.

3.2 PQG Window

A PQG window is specifically designed to display Publication Quality Graphics. Text input and output is not supported for PQG windows.

3.2.1 Opening a PQG Window

WinOpenPQG opens a PQG window. You can specify the following on opening:

- window position (pixels)
- window size (text rows and columns)
- colormap
- foreground and background color
- fixed or variable aspect ratio
- resizeability

Defaults are defined for each of these settings:

window position: (0,0) *(upper left corner of screen)*
window size: 640×480
16 color colormap (emulates PC EGA palette)
black background, white foreground
fixed aspect ratio
resize by function or mouse

You can retain any or all of the default values on opening a window.

3.2.2 Selecting Colors

Each PQG window has an associated colormap. You can examine the colors in the colormap with **WinGetColorCells**, and change them with **WinSetColorCells**. Out of the colormap you can select the foreground and background colors for window output. The foreground is the the default color of the lines, text, and other PQG output, and the background is the color of the field on which they are drawn. You can change the foreground and background through the commands provided in the Publications Quality Graphics interface.

Note: The graphics library used in Publication Quality Graphics is currently limited to using the 16 color palette. While the appearance of the screen can be changed by the selection of an alternate colormap, all output to the printer, and all file conversions are treated as if the active colormap is 16 color.

3.2.3 Aspect Ratio

PQG windows may be opened with either a fixed or variable aspect ratio. For windows opened with fixed aspect ratio, both resizing and zooming are affected. The system forces one dimension of the window or zoom box to change by an amount proportional to any change in the other dimension. For the zoom feature, the X size of the zoom box determines the Y size. For resizing, the window manager determines which dimension is dominant, if any. The default for PQG windows is fixed aspect ratio.

3.3 Text Window

A Text window is a text I/O window with an underlying text buffer. All operations take place with respect to the buffer. A window cannot be resized larger than its buffer.

3.3.1 Opening a Text Window

WinOpenText opens a Text window. You can specify the following on opening:

- window location (pixels)
- window size (text rows and columns)
- buffer size (text rows and columns)
- text font
- colormap
- foreground and background color
- text wrapping mode
- refresh mode
- resizeability

Defaults are defined for each of these settings:

window position: (0,0) *(upper left corner of screen)*
window size: 25×80
buffer size: 25×80
system font
system colormap
system foreground and background
character wrap
refresh on *newline* (\n)
resize by function or mouse

You can retain any or all of the default values on opening a window.

3.3.2 Selecting Colors

Each Text window has an associated colormap. You can examine the colors in the colormap with **WinGetColorCells**, and change them with **WinSetColorCells**. Out of the colormap you can select the foreground and background colors for Text window output. The foreground is the color of the text itself, and the background is the color of the field on which it is printed. You can change the foreground color at any time, the background color, however, may not be changed after the window has been opened.

The **color** command sets the foreground color for the active window.

3.3.3 Locating Text

Text windows give you full control over text location, allowing you to "paint" a screen and set up output fields wherever you like. All text location functions operate with respect to the window buffer, NOT the current window appearance. Thus, you may locate and print text in an area that is not currently visible.

WinSetCursor sets the cursor position in a specified window. **locate** sets the cursor position in the active window. **tab** sets the cursor position on the current line of the active window. You can tab forward and backward. You can also embed **tab** commands within a **print** statement.

When you locate and print in a region that already has text, the existing text is overwritten.

Nonintegral arguments to the text location functions are truncated; for example,

```
locate 13.25, 40.75;
```

is equivalent to:

```
locate 13, 40;
```

It is not necessary for location arguments to be located within the window buffer. For example,

```
locate 1,-5;
```

is allowed. You can think of the buffer as covering a small region in "text coordinate space"; text written within the buffer bounds is remembered, text written outside it is lost.

3.3.4 Wrapping Text

Three text wrapping modes are supported:

- word wrapping
- character wrapping
- clipping

Word wrapping: Lines are wrapped between words. A line wrap on the last row of the buffer forces an auto-scroll of the buffer contents. Outputting a *newline* (\n) on the last row also forces an auto-scroll.

Character wrapping: Lines are wrapped between characters. Auto-scrolling is the same as for word wrapping.

Clipping: Lines are clipped at the buffer edge. No auto-scrolling takes place, not even when a *newline* (\n) is output to the last row of the buffer. If you want to scroll the contents of the buffer, you must do it explicitly with the **scroll** command.

Again, all output commands in a Text window operate with respect to the buffer, not the window. So, in clipping mode, any text that goes beyond the width of the buffer is discarded and cannot be retrieved. Likewise, any time text is scrolled off the top of the buffer, it is permanently discarded.

3.4 TTY Window

A TTY window is a text I/O window that logs output for a user-specified number of lines. Each line of output is saved in its entirety, and the window size, text wrap mode, etc., merely determine how the output is displayed. The only time you lose information in a TTY window is when you reach the output log limit and lines begin to scroll off the top of the output log.

TTY windows have color support for foreground only. The color commands will run in a TTY window, however, they will only effect the foreground color.

There are no sizing limits for TTY windows.

3.4.1 Opening a TTY Window

WinOpenTTY opens a TTY window. You can specify the following on opening:

- window location (pixels)
- window size (text rows and columns)
- output log size (number of lines)
- text font
- colormap
- foreground and background color
- text wrapping mode
- refresh mode
- resizeability

Defaults are defined for each of these settings:

window position: (0,0) *(upper left corner of screen)*
window size: 25×80
output log size: 500 lines
system font
system colormap
system foreground and background
character wrap
refresh on *newline* (\n)
resize by function or mouse

You can retain any or all of the default values on opening a window.

3.4.2 Locating Text

TTY windows have minimal text location support. You can use **tab** and **locate** to tab the cursor forward and backward in the active window, or **WinSetCursor** to tab the cursor in a specified window. For **locate** and **WinSetCursor**, the row argument is ignored.

When you tab backward and then print, the existing text is overwritten.

The tabbing commands operate with respect to the length of the current output line, not the window appearance. This becomes significant when a line gets longer than the window width, and wraps. For example, if a window is 80 columns wide and the output line is 90 columns wide, and you tab to column 85, the cursor will appear to be in column 5. If you then widen the window to 100 columns, the output line will "unwrap" and the cursor will appear where you expect it, in column 85.

3.4.3 Wrapping Text

Three text wrapping modes are supported:

- word wrapping
- character wrapping
- clipping

Word wrapping: Lines are wrapped between words. A line wrap on the last row of the window forces an auto-scroll of the buffer contents. Outputting a *newline* (\n) on the last row also forces an auto-scroll.

Character wrapping: Lines are wrapped between characters. Auto-scrolling is the same as for word wrapping.

Clipping: Lines are clipped at the window edge. Auto-scrolling takes place only when a *newline* (\n) is output to the last row of the window. (Note the difference between this and a clipped Text window, which never auto-scrolls.)

The text wrap mode in a TTY window defines how output is displayed in the window; it has nothing to do with how output is stored, as is the case in a Text window. So, the word and character wrapping modes specify that each line of output should be completely displayed within the window boundaries; nothing is "hidden" off-screen (until it scrolls off the top of the window). If you resize the window, the entire display will change as output is rewrapped accordingly. Likewise, clipping mode specifies that the portions of a line beyond the edges of the window should be clipped from the display, but nothing discarded. If you widen the window, you will see more of the text.

A final note about auto-scrolling: if the current output line has been panned below the bottom edge of the window, additional output will not scroll the window contents. The output is still added to the output log; it just doesn't affect the display. This allows you to study previous text during ongoing output. Pan the output line back within view, and normal window updating and auto-scrolling will resume.

3.5 Special Purpose Windows

GAUSS recognizes three special purpose windows: the command window, the help window, and the active window.

3.5.1 Command Window

The *command window* ("window 1") is the main window used for controlling **GAUSS**. It contains both the control buttons, and a TTY text pane for running **GAUSS** interactively. (If **GAUSS** is running in terminal mode, the system terminal window is the command window.) Since the window handle for the command window is 1, the command window is sometimes referred to as "window 1".

The top of the command window contains buttons which are used to control a **GAUSS** session. (These replace the main interactive control commands used in the DOS version.) The following buttons appear in window 1:

Help	toggle **GAUSS** help system on/off
Status	display program status, processes executing
Config	display/modify Compile and Run options
Trans	toggle file translation on/off
Cancel	cancel any commands pending in the command buffer
Stop	stop the currently executing progam
Kill	terminate child processes
Quit	quit **GAUSS** and exit to system.

The text pane portion of the command window is a standard TTY window. Most of the **GAUSS** commands which are valid for user defined TTY windows are also valid for the command window. To execute a **GAUSS** function on the command window, simply set the window handle parameter of the command to 1. There are some functions which are not defined for the command window, such as **WinClose**, and **WinSetRefresh**. These are documented in the Command Reference. The command **!!** may be used to repeat the last typed command.

3.5.2 Help Window

The *help window* is the window used for displaying output from the interactive help system. The **Help** button on the command window toggles the help window on and off. The help system is available at any time during a **GAUSS** session, with output directed to a dedicated TTY window. The window handle for the help window is 2. (If **GAUSS** is running in terminal mode, the system terminal window is the help window.)

Most of the **GAUSS** commands which are valid for user defined TTY windows are also valid for the help window. To execute a **GAUSS** function on the help window, simply set the window handle parameter of the command to 2. There are some functions such as **WinClose** and **WinSetRefresh** which are either not defined for the help window, or which operate differently on the help window. These are documented in the Command Reference.

3.5.3 Active Window

The *active window* refers to the default I/O window. All the input functions (**con**, **cons**, **key**, **keyw**, **wait** and **waitc**) get their input from the active window, and all the output and output-related commands that do not take a window handle as an argument (**locate**, **output**, **print**, **scroll**, **tab**, etc.) operate on the active window. In addition, certain functions that set window or output attributes (**color**, **font**, etc.) operate on the active window.

WinSetActive and **WinGetActive** set and get the active window. Any window can be the active window, and you can change the active window whenever you want. Many windows can be visible simultaneously, but there is only one active window.

When you first start **GAUSS**, the command window (window 1) is the active window.

It is possible not to have an active window; you can remove the active window by unsetting it with **WinSetActive** or closing it with **WinClose**. If there is no active window, the commands that operate on it will still run—they just won't have any effect (for example, the output of a **print** statement will simply not be displayed anywhere). If there is no active window when a **GAUSS** program ends, the command window automatically becomes the active window.

Note: If you set the active window to a PQG window, no character I/O will be processed.

Chapter 4

Debugger

GAUSS for UNIX includes a source level debugger which executes a program one line or opcode at a time, enabling the user to examine and modify program variables during execution.

The debugger is a dynamically linked X Window application. The X libraries must be available on the system when it is invoked.

4.1 Starting the Debugger

To start the debugger, enter:

```
debug filename
```

at the **GAUSS** prompt.

The debugger initially displays two windows: the Command window, which contains buttons used to control program flow, and the Source window, which displays the source code to be executed. Section 4.3 describes the other windows the debugger uses.

Debugger commands are issued by clicking on the buttons in the Command window.

4.2 Commands

Next Line	Execute next source line
Step Line	Execute next source line, enter procedure calls
Next Opcode	Execute next opcode
Step Opcode	Execute next opcode, enter procedure calls
Go	Execute with no tracing
Verbose	Execute with opcode tracing
Brkpts On|Off	Toggle breakpoint checking on and off
Brkpts Set	Detailed information and control of breakpoints
Examine	Examine and modify variable contents
History	Toggle opcode history on and off
Stack	Examine and modify stack contents
Procs	List procedures and set breakpoints
Quit	Exit debugger

Next Line Executes the currently highlighted source line. If the current line contains any procedure calls, they are executed (along with any procedures which they call) before the debugger pauses for input.

Step Line Executes the currently highlighted source line. If the current line contains a procedure call, execution pauses at the first line within the procedure. If the line contains more than one procedure call, execution will pause at the first line in each procedure (assuming you continue pressing **Step Line**).

Next Opcode Executes the currently highlighted opcode. If the opcode is a procedure call, the entire procedure is executed (along with any procedures which it calls) before the debugger pauses for input.

Step Opcode Executes the currently highlighted opcode. If the opcode is a procedure call, execution pauses at the first opcode within the procedure.

Go Executes the program until reaching either a breakpoint or the end of the code. Selecting **Go** automatically turns off the Opcode History window.

Verbose Executes the program until reaching either a breakpoint or the end of the code. Selecting **Verbose** automatically turns on the Opcode History window.

Brkpts On|Off Toggles breakpoint checking on and off. When breakpoint checking is off, the debugger ignores all breakpoints. When breakpoint checking is on, the debugger stops at the first active breakpoint.

A small banner is displayed in the upper right corner of the Command window when breakpoint checking is on.

Brkpts Set Opens/closes the Breakpoint window. The Breakpoint window displays the data for all breakpoints currently specified in the program. You can modify breakpoint settings in this window. See Section 4.4 for details.

Examine Requests a variable name, then opens an Examine window to display it. The window is updated after each command. It remains open until it is explicitly closed or the variable is no longer valid. Global variables are always valid; local variables become invalid after a **retp**. The debugger will allow up to 100 variables to be monitored at any given time.

History Opens/closes the Opcode History window. When open, the Opcode History window displays and logs opcodes as they are executed. It can log up to 4096 opcodes. This window is automatically opened whenever the **Next Opcode**, **Step Opcode**, or **Verbose** command is issued, and closed whenever the **Go** command is issued. While immensely useful for tracking down program errors, generating the opcode history is very output intensive and greatly increases program execution time while active.

Stack Opens/closes the Stack window. The Stack window displays the contents of the stack for examination and modification. The window is automatically updated after each command.

You will only see stack activity when you're selecting **Next Opcode** or **Step Opcode**. If you're selecting **Next Line** or **Step Line**, the stack will appear to be always empty.

Procs Opens/closes the Procedure window. The Procedure window lists the names of all procedures and functions found in the source code. Procedure breakpoints can be set and cleared from this window. See Section 4.4 for details.

Quit Ends the debugging session and exits the debugger.

4.3 Windows

The debugger utilizes a variety of different windows to display and edit the various types of program data and debugger settings.

Window types include:

Command	Command buttons, status indicators, debugger messages
Source	Currently executing source code, highlight indicates next source line to be executed

History	Opcode history, highlight indicates next opcode to be executed
Breakpoint	Lists all defined breakpoints, allows editing of breakpoints, highlight indicates currently selected breakpoint
Stack	Displays current stack contents, highlight indicates currently selected stack element, clicking on selected item opens Examine window
Procedure	Displays the names of all procedures and functions defined in the current program, highlight indicates currently selected procedure name
Matrix	Examine Matrix window, displays the contents of matrix, highlight indicates currently selected element, clicking on selected element allows modification
String	Examine String window, displays the contents of string, no highlight is displayed (strings cannot be modified)
Edit	Allows text/numeric entry into the debugger, highlight indicates current cursor position

The windows have a given default size, but can be adjusted (within limits) to whatever size is desired. Since there is often more information available than can be displayed within the window, the debugger supports keystroke movement of the highlight or window contents as follows:

Up	move up one element
Down	move down one element
Left	move left one element
Right	move right one element
Shift+Up	move up one window height
Shift+Down	move down one window height
Shift+Left	move left one window width
Shift+Right	move right one window width
Ctrl+Up	go to top
Ctrl+Down	go to bottom
Ctrl+Left	go to far left
Ctrl+Right	go to far right
PageUp	move up one window height
PageDown	move down one window height
Ctrl+PageUp	go to top
Ctrl+PageDown	go to bottom

Home	go to upper left
End	go to lower right
Enter	select item or accept changes
Esc	discard changes, close window

While the definition of "element" may vary according to the window type, every effort has been made to keep the keystroke actions intuitive, and there should be no confusion.

4.4 Breakpoints

The debugger supports two types of breakpoints, procedure breakpoints and line number breakpoints. Procedure breakpoints pause execution when the specified procedure or function is entered. Line number breakpoints pause execution when the specified line is reached. In either case, the break occurs before any of the **GAUSS** code for the procedure or line is executed.

Procedure breakpoints can be set and cleared in the Procedure window by clicking on the desired procedure name. You can also use the cursor keys to move the highlight to the desired procedure, then press **Enter** to set/clear a breakpoint.

Line number breakpoints can be set and cleared in the Source window by clicking on the desired line.

Breakpoints can be set, cleared and modified in the Breakpoints window, which is opened and closed by the **Brkpts Set** command. This window displays all the breakpoints defined for the current debugging session. You can select a breakpoint by clicking on it with the mouse or moving the highlight with the cursor keys. Once selected, the breakpoint settings can be modified as follows:

1. Clicking on the **ON/OFF** field toggles the state of the breakpoint. Breakpoints which are off remain defined, but are inactive. Active breakpoints are indicated in the Procedure and Source windows with an inverse video tag **[BP]**, inactive breakpoints are indicated with the tag **(BP)** in normal video.

2. Clicking on the **Skip** field opens an Edit window, which allows the skip value to be changed. Breakpoint checking in the debugger skips over the breakpoint the specified number of times. For example, 0 = don't skip, 24 = skip 24 times, break on the 25th occurence. Skip counters are reset to 0 whenever the breakpoint causes a break. For example, a skip value of 24 on a breakpoint within a loop from 1 to 100 will break 4 times, at $i = 25$, 50, 75, and 100).

   Note: The **Skip** field is currently only valid for line number breakpoints.

3. Clicking on the **Line #** field opens an Edit window, which allows the line number to be changed.

Chapter 5

Foreign Language Interface

The Foreign Language Interface (FLI) allows users to create functions written in C, FORTRAN, or other languages and call them from a **GAUSS** program. The functions are placed in dynamic-link libraries (DLL's, also known as shared libraries or shared objects) and linked in at run-time as needed. The FLI support functions are:

dlibrary	Link and unlink dynamic libraries at run-time.
dllcall	Call functions located in dynamic libraries.

GAUSS recognizes a default dynamic library directory, a directory where it will look for your dynamic-link libraries when you call **dlibrary**. You can specify the default directory in `gauss.cfg` by setting **dlib_path**. As it is shipped, `gauss.cfg` specifies `$GAUSSHOME/dlib` as the default directory.

5.1 Creating Dynamic Libraries

Assume you want to build a dynamic library named **libmyfuncs.so**, containing the functions found in two source files, `myfunc1.c` and `myfunc2.c`. The following sections show the compile and link commands you would use on the various platforms **GAUSS** for Unix runs on, assuming you're using the ANSI C compiler offered for each platform. The compiler command is first, followed by the linker command, followed by remarks regarding that platform.

The ANSI C compiler is included on some platforms; on others it is a separate purchase.

The compiler is usually called **cc**, the linker is usually called **ld**. On some platforms, the linker is invoked by the compiler; on most platforms, we call it directly. Thus, the link command is sometimes done with **ld**, sometimes with **cc**.

For explanations of the various flags used, see the man pages on your system. Two flags are common to all platforms. The **-c** compiler flag means "compile only, don't link". Virtually all compilers will perform the link phase automatically unless you tell them not to. When building a dynamic library we want to compile the source code files to object (`.o`) files, then link the object files in a separate phase into a dynamic library.

The **-o** linker flag means "name the output file as follows", and must be followed by the name of the library you want to create, in this case `libmyfuncs.so`.

$(CFLAGS) indicates any optional compilation flags you might add.

You may have to use the **-l** flag to specify additional libraries on the link command line.

5.1.1 Solaris 2.x

```
cc -c $(CFLAGS) myfunc1.c myfunc2.c

ld -G -o libmyfuncs.so myfunc1.o myfunc2.o
```

5.1.2 SunOS 4.x

```
acc -pic -c $(CFLAGS) myfunc1.c myfunc2.c

ld -assert pure-text -assert definitions -o libmyfuncs.so
myfunc1.o myfunc2.o
```

5.1.3 IBM AIX

```
cc -pic -c $(CFLAGS) myfunc1.c myfunc2.c

cc -o libmyfuncs.so myfunc1.o myfunc2.o -bE:libmyfuncs.exp
-bM:SRE -e _nostart
```

The **-bE:libmyfuncs.exp** flag specifies the name of the **export** file. The AIX linker requires that you specify the names of the symbols that will be exported, i.e., made callable by other processes, from the dynamic library. The export file is just a text file that lists the exported symbols, one per line. Assume that `myfunc1.c` and `myfunc2.c` contain the functions **func1()**, **func2()**, and **func3()**, and a static function **func4()** that is called by the others, but never directly from **GAUSS**. For this example, the export file would look like this:

```
func1
func2
func3
```

5.1.4 DEC OSF1

```
cc -c $(CFLAGS) myfunc1.c myfunc2.c

ld -shared -o libmyfuncs.so myfunc1.o myfunc2.o -lc
```

Under OSF1, if your library calls functions in the standard C runtime library, `libc.so`, you have to explicitly link it in. The **-lc** flag does this.

5.1.5 HP-UX

```
cc -c +z $(CFLAGS) myfunc1.c myfunc2.c

ld -b -c libmyfuncs.exp -o libmyfuncs.so myfunc1.o myfunc2.o
```

Like AIX, the HP-UX linker requires that you indicate the symbols to be exported. In this case, however, the **-c libmyfuncs.exp** flag specifies the name of an indirect command file. Indirect command files can contain any linker switches you want to place in them; for our purposes, `libmyfuncs.exp` contains only **+e**, or *mark symbol for export*, switches. Assuming the example described in the AIX section, `libmyfuncs.exp` would look like this:

```
+e func1
+e func2
+e func3
```

5.1.6 SGI IRIX

```
cc -c $(CFLAGS) myfunc1.c myfunc2.c

ar cq myfuncs.a myfunc1.o myfunc2.o

ld -elf -rdata_shared -shared -all myfuncs.a -o libmyfuncs.so
```

IRIX requires an extra step. The object files produced by the compiler are combined into a standard archive file using **ar**; this archive file becomes the input from which the linker constructs the dynamic library.

5.1.7 Linux

```
gcc -fpic -c $(CFLAGS) myfunc1.c myfunc2.c

gcc -shared -o libmyfuncs.so myfunc1.o myfunc2.o -lc
```

Like the DEC, if your library calls functions in the standard C runtime library, `libc.so`, you have to explicitly link it in under Linux. Once again, the **-lc** flag does this.

5.2 Writing FLI Functions

Your FLI functions should be written to the following specifications:

1. Take 0 or more pointers to doubles as arguments.

   This doesn't mean you can't pass strings to an FLI function. Just recast the double pointer to a char pointer inside the function.

2. Take those arguments either in a list or a vector.

3. Return an integer.

In C syntax, then, your functions would take one of the following forms:

1. **int func(void);**

2. **int func(double *arg1⟦, double *arg2, etc.⟧);**

3. **int func(double *argv[]);**

Functions can be written to take a list of up to 100 arguments, or a vector (in C terms, a 1-dimensional array) of up to 1000 arguments. This does not affect how the function is called from **GAUSS**; the **dllcall** statement will always appear to pass the arguments in a list. That is, the **dllcall** statement will always look as follows:

dllcall func(a,b,c,d⟦,e...⟧);

See **dllcall** in chapter 6 for details on calling your function, passing arguments to it and getting data back, and what the return value means.

Chapter 6

FLI Function Reference

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 fully 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 prefix `lib` and the suffix `.so`. 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.

-a append flag, the DLL's listed are added to the current set of DLL's rather than replacing them. For search purposes, the new DLL's follow the already active ones. Without the **-a** flag, any previously linked libraries are dumped.

-d dump flag, **ALL** DLL's are unlinked and the functions they contain are no longer available to your programs. If you use **dllcall** to call one of your functions after executing a **dlibrary -d**, your program will terminate with an error.

Remarks

If no flags are used, the DLL's listed are linked into **GAUSS** and any previously linked libraries are dumped. When you call **dllcall**, the DLL's will be searched in the order listed for the specified function. The first instance of the function found will be called.

dlibrary with no arguments prints out a list of the currently linked DLL's. The order in which they are listed is the order in which they are searched for functions.

dlibrary recognizes a default directory in which to look for dynamic libraries. You can specify this by setting the variable **dlib_path** in `gauss.cfg`. Set it to point to a single directory, not a sequence of directories. A new case (case 24) has also been added to **sysstate** for getting and setting this default.

GAUSS maintains its own DLL, `libgauss.so`. `libgauss.so` is listed when you execute **dlibrary** with no arguments, and searched when you call **dllcall**. By default,

`libgauss.so` is searched last, after all other DLL's, but you can force it to be searched earlier by listing it explicitly in a **dlibrary** statement. `libgauss.so` is always active. It is not unlinked when you execute **dlibrary -d**. `libgauss.so` is located in the $GAUSSHOME directory.

See also

dllcall, **sysstate–case 24**

Purpose

Calls functions located in dynamic libraries.

Format

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

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

Input

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

arg# arguments to be passed to *func*; optional. These must be elementary variables; they cannot be expressions.

-r optional flag. If **-r** is specified, **dllcall** examines the value returned by *func*, and fails if it is nonzero.

-v optional flag. Normally, **dllcall** passes parameters to *func* in a list. If **-v** is specified, **dllcall** passes them in a vector. See below for more details.

Remarks

func should be written to:

1. Take 0 or more pointers to doubles as arguments.
2. Take arguments either in a list or a vector.
3. Return an integer.

In C syntax, *func* should take one of the following forms:

1. **int** *func***(void);**
2. **int** *func***(double *** *arg1***[,double *** *arg2***...]);**
3. **int** *func***(double *** *argv***[]);**

dllcall can pass a list of up to 100 arguments to *func*; if it requires more arguments than that, you MUST write it to take a vector of arguments, and you MUST specify the **-v** flag when calling it. **dllcall** can pass up to 1000 arguments in vector format. In addition, in vector format **dllcall** appends a null pointer to the vector, so you can write *func* to take a variable number of arguments and just test for the null pointer.

Arguments are passed to *func* by reference. This means you can send back more than just the return value, which is usually just a success/failure code. (It also means that you need to be careful not to overwrite the contents of matrices or strings you want to preserve.) To return data from *func*, simply set up one or more of its arguments as return matrices (basically, by making them the size of what you intend to return), and inside *func* assign the results to them before returning.

See also

dlibrary, **sysstate–case 24**

Chapter 7

Language Fundamentals

GAUSS is a compiled language. **GAUSS** is also an interpreter. These may seem contradictory but it is necessary to understand each of these ideas to understand how **GAUSS** works and how it can be used most effectively.

GAUSS is a compiled language because it scans the entire program once and translates it into a binary code before it starts to execute the program. **GAUSS** is an interpreter because the binary code is not the native code of the CPU. When **GAUSS** executes the binary pseudocode it must "interpret" each instruction for the computer rather than just turning loose the CPU to execute the binary code by itself.

How then can **GAUSS** be so fast if it is an interpreter? There are two reasons. The first is that **GAUSS** has a fast interpreter, and the binary compiled code is compact and efficient. The second reason, and the most significant one, is that **GAUSS** is a matrix language. It is designed to tackle problems that can be solved in terms of matrix or vector equations. Much of the time lost in interpreting the pseudocode is made up in the matrix or vector operations.

In the following pages it will be necessary to understand the distinction between "compile time" and "execution time" because these are two very different stages in the life of a **GAUSS** program.

7.1 Expressions

An expression is a matrix, string, constant, function or procedure reference or any combination of these joined by operators. An expression returns a result that can be assigned to a variable with the assignment operator '='.

7.2 Statements

A statement is a complete expression or command. Statements end with a semicolon.

```
y = x*3;
```

If an expression has no assignment operator (=), it will be assumed to be an implicit **print** statement.

```
print x*3;

x*3;
```

The two statements above are equivalent.

```
output on;
```

This is an example of a statement that is a command rather than an expression. Commands cannot be used as a part of an expression.

There can be multiple statements on the same line as long as each statement is terminated with a semicolon.

7.2.1 Executable Statements

Executable statements are statements that can be "executed" over and over during the execution phase of a **GAUSS** program. As an executable statement is compiled, binary code is added to the program being compiled at the current location of the instruction pointer. This binary code will be executed whenever the interpreter passes through this section of the program. If this code is in a loop, it will be executed each iteration of the loop.

Here are some examples of executable statements.

```
y = 34.25;

print y;

x = { 1 3 7 2 9 4 0 3 };
```

7.2.2 Nonexecutable Statements

Nonexecutable statements are statements that have an effect only at compile time. They generate no executable code at the current location of the instruction pointer. Here are some examples:

```
declare matrix x = { 1 2 3 4 };

external matrix ybar;
```

Procedure **definitions** are nonexecutable. They do not generate executable code **at the current location of the instruction pointer**. Here is an example:

```
zed = rndn(3,3);

proc sqrtinv(x);
    local y;
    y = sqrt(x);
    retp(y+inv(x));
endp;

zsi = sqrtinv(zed);
```

The two executable statements are the first line and the last line. In the binary code that is generated, the last line will follow immediately after the first line. Everything in the procedure definition is tucked away somewhere else. The last line is the **call** to the procedure. This generates executable code. The procedure definition generates no code at the current location of the instruction pointer.

There is code generated in the procedure definition, but it is isolated from the rest of the program. It is executable only within the scope of the procedure and can be reached only by calling the procedure.

7.3 Programs

A program is any set of statements that are run together at one time. This is a rather loose definition, but it should suffice. If the **run** statement is used from within a program, it starts a new program. There are two distinct sections within a program which need to be understood.

7.3.1 Main Section

The first is the main section. The main section of the program is all of the code that is compiled together WITHOUT relying on the autoloader. This means code that is in the main file or is included in the compilation of the main file with an **#include** statement. ALL executable code should be in the main section.

There is always a main section even if it only consists of a call to the one and only procedure that is called in the program. The main program code is stored in an area of memory that can be adjusted in size with the **new** command.

7.3.2 Secondary Sections

Secondary sections of the program are files that are neither run directly nor included in the main section with **#include** statements.

The secondary sections of the program can be left to the autoloader to locate and compile when they are needed. Secondary sections must have only procedure definitions and other nonexecutable statements. If you violate this, you will have problems. **GAUSS** does not currently enforce this, but it will in later versions.

#include statements are okay in secondary sections as long as the file being included does not violate the above criteria. Here is an example of a secondary section.

```
declare matrix tol = 1.0e-15;

proc feq(a,b);
    retp(abs(a-b) <= tol);
endp;
```

7.4 Compiler Directives

Compiler directives are commands that tell **GAUSS** how to process a program during compilation. Directives determine what the final compiled form of a program will be. They can affect part or all of the source code for a program. Directives are not executable statements and have no effect at run-time.

The **#include** statement mentioned above is actually a compiler directive. It tells **GAUSS** to compile code from a separate file as though it were actually part of the file being compiled. This code is compiled in at the position of the **#include** statement. Here are the compiler directives available in **GAUSS**:

#define	Define a case-insensitive text-replacement or flag variable.
#definecs	Define a case-sensitive text-replacement or flag variable.
#undef	Undefine a text-replacement or flag variable.
#ifdef	Compile code block if a variable has been **#define**'d.
#ifndef	Compile code block if a variable has not been **#define**'d.
#iflight	Compile code block if running **GAUSS Light**.
#ifdos	Compile code block if running DOS.
#ifos2win	Compile code block if running OS/2 or Windows.
#ifunix	Compile code block if running Unix.
#else	Else clause for **#if-#else-#endif** code block.
#endif	End of **#if-#else-#endif** code block.
#include	Include code from another file in program.
#lineson	Compile program with line number and file name records.
#linesoff	Compile program without line number and file name records.
#srcfile	Insert source file name record at this point (currently used when doing data loop translation).
#srcline	Insert source file line number record at this point (currently used when doing data loop translation).

The **#define** statement can be used to define abstract constants. For example, you could define the default graphics page size as:

```
#define hpage       9.0
#define vpage       6.855
```

You can then write your program using **hpage** and **vpage**, and **GAUSS** will replace them with **9.0** and **6.855** when it compiles the program. This can make programs much more readable.

The **#ifdef**–**#else**–**#endif** directives allow you to conditionally compile sections of a program, depending on whether a particular flag variable has been **#define**'d. For example:

```
#ifdef log_10
    y = log(x);
#else
    y = ln(x);
#endif
```

This allows the same program to calculate answers using different base logarithms, depending on whether or not the program has a **#define log_10** statement at the top.

#undef allows you to undefine text-replacement or flag variables, so they no longer affect a program, or so you can **#define** them again with a different value for a different section of the program. If you use **#definecs** to define a case-sensitive variable, you must use the right case when **#undef**'ing it.

With **#lineson**, **#linesoff**, **#srcline** and **#srcfile** you can include line number and file name records in your compiled code, so that run-time errors will be easier to track down. **#srcline** and **#srcfile** are currently used by **GAUSS** when doing data loop translation (see Chapter 13). For a discussion of line number tracking see the *Debugger*, or *Error Handling and Debugging* chapter in your platform supplement. See also **#lineson** in the *COMMAND REFERENCE*.

The syntax for **#srcfile** and **#srcline** is a little different than for the other directives that take arguments. Typically, directives do not take arguments in parentheses (i.e. they look like keywords).

```
#define red 4
```

#srcfile and **#srcline**, however, do take their arguments in parentheses (like procedures).

```
#srcline(12)
```

This allows you to place **#srcline** statements in the middle of **GAUSS** commands, so that line numbers are reported precisely as you want them. For example:

```
#srcline(1)
print "Here is a multi-line "
#srcline(2)
"sentence--if it contains a run-time error, "
#srcline(3) "you will know exactly "
#srcline(4) "which part of the sentence has the problem.";
```

The argument supplied to **#srcfile** does not need quotes.

```
#srcfile(/gauss/test.e)
```

7.5 Procedures

A procedure allows you to define a new function which you can then use as if it were an intrinsic function. It is called in the same way as an intrinsic function.

```
y = myproc(a,b,c);
```

Procedures are isolated from the rest of your program and cannot be entered except by calling them. Some or all of the variables inside a procedure can be local variables. **local** variables are variables that exist only when the procedure is actually executing and then disappear. Local variables cannot get mixed up with other variables of the same name in your main program or in other procedures.

See Chapter 9 for details on defining and calling procedures.

7.6 Data Types

There are only two basic data types in **GAUSS**, matrices and strings. It is not necessary to declare the type of a variable. The data type and size can change in the course of a program.

The **declare** statement, used for compile-time initialization, enforces type checking and it is good programming practice to respect the types of variables whenever possible.

Short strings of up to 8 bytes can be entered into elements of matrices, to form "character matrices". These are discussed in Section 7.6.4.

7.6.1 Constants

The following constant types are supported.

Decimal

Decimal constants can be either integer or floating point values.

```
1.34e-10
1.34e123
-1.34e+10
-1.34d-10
1.34d10
1.34d+10
123.456789345
```

Up to 18 consecutive digits before and after the decimal point(depending on the platform) are significant, but the final result will be rounded to double precision if necessary. The range is the same as for matrices. See Section 7.6.2.

String

String constants are enclosed in quotation marks.

```
"This is a string."
```

Hexadecimal Integer

Hexadecimal integer constants are prefixed with **0x**.

```
0x0ab53def2
```

Hexadecimal Floating Point

Hexadecimal floating point constants are prefixed with **0v**. This allows you to input a double precision value exactly as you want using 16 hexadecimal digits. The highest order byte is to the left.

```
0vfff8000000000000
```

7.6.2 Matrices

Matrices are 2-dimensional arrays of double precision numbers. All matrices are implicitly complex, although if it consists only of zeros, the imaginary part may take up no space. Matrices are stored in row major order. A 2x3 real matrix will be stored in the following way from the lowest addressed element to the highest addressed element.

$$[1,1] \quad [1,2] \quad [1,3] \quad [2,1] \quad [2,2] \quad [2,3]$$

A 2x3 complex matrix will be stored in the following way from the lowest addressed element to the highest addressed element.

(*real part*)	$[1,1]$	$[1,2]$	$[1,3]$	$[2,1]$	$[2,2]$	$[2,3]$
(*imaginary part*)	$[1,1]$	$[1,2]$	$[1,3]$	$[2,1]$	$[2,2]$	$[2,3]$

Conversion between complex and real matrices occurs automatically and is transparent to the user in most cases. Functions are provided to provide explicit control when necessary.

All numbers in **GAUSS** matrices are stored in double precision floating point format, and each takes up 8 bytes of memory. This is the IEEE 754 format.

Bytes	Data Type	Significant Digits	Range
8	floating point	15–16	$4.19\text{x}10^{-307} \leq \lvert X \rvert \leq 1.67\text{x}10^{+308}$

Matrices with only one number (1x1 matrices) are referred to as scalars, and matrices with only one row or column (1xN or Nx1 matrices) are referred to as vectors.

Any matrix or vector can be indexed with 2 indices. Vectors can be indexed with one index. Scalars can be indexed with one or two indices also, because scalars, vectors, and matrices are the same data type to **GAUSS**

The majority of functions and operators in **GAUSS** take matrices as arguments. The following functions and operators are used for defining, saving, and loading matrices:

[ ]	indexing matrices
=	assignment operator
|	vertical concatenation
~	horizontal concatenation
con	numeric input from keyboard
cons	character input from keyboard
declare	compile-time matrix or string initialization
let	matrix definition statement
load	load matrix (same as **loadm**)
readr	read from a **GAUSS** file
save	save matrices, procedures and strings to disk
saved	convert a matrix to a **GAUSS** data set
stof	convert string to matrix
submat	extract a submatrix
writer	write data to a **GAUSS** data set

Examples of matrix definition statements are:

```
let x = { 1 2 3, 4 5 6, 7 8 9 };
```

or

```
x = { 1 2 3, 4 5 6, 7 8 9 };
```

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

An assignment statement followed by data enclosed in braces is an implicit **let** statement. Only constants are allowed in **let** statements; operators are illegal. When braces are used in **let** statements, commas are used to separate rows.

```
let x[3,3] = 1 2 3 4 5 6 7 8 9;
```

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

---

let x[3,3] = 1;

$$x = \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

---

let x[3,3];

$$x = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

---

let x = 1 2 3 4 5 6 7 8 9;

$$x = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{matrix}$$

---

let x[2,2] = 1+2i 3-4 5 6i;

$$x = \begin{matrix} 1 + 2i & 3 - 4i \\ 5 & 0 + 6i \end{matrix}$$

Complex constants can be entered in a **let** statement. Note that in this case, the + or - is not a mathematical operator, but connects the two parts of a complex number. There can be no spaces between the + or - and the parts of the number. If a number

has both real and imaginary parts, the trailing 'i' is not necessary. If a number has no real part, you can indicate that it is imaginary by appending the 'i'.

Complex constants can also be used with the **declare**, **con** and **stof** statements.

An "empty matrix" is a matrix that contains no data. Empty matrices are created with the **let** statement.

```
x = {};
```

You must use the form of the **let** statement with curly braces to create an empty matrix.

Empty matrices are currently supported ONLY by the **rows** and **cols** functions and the concatenation (~,|) operators.

```
x = {};
hsec0 = hsec;
do until  hsec-hsec0 > 6000;
    x = x ~ data_in(hsec-hsec0);
endo;
```

You can test whether a matrix is empty by entering **rows(**x**)**, **cols(**x**)** and **scalerr(**x**)**. If the matrix is empty **rows** and **cols** will return a 0, and **scalerr** will return 65535.

```
y = 1~2|3~4;
```

The ~ is the horizontal concatenation operator and the | is the vertical concatenation operator. This statement would be evaluated as:

```
y = (1~2)|(3~4);
```

because horizontal concatenation has precedence over vertical concatenation, and would result in a 2x2 matrix.

```
1   2
3   4
```

```
y = 1+1~2*2|3-2~6/2;
```

This statement would be evaluated as:

```
y = ((1+1)∼(2*2))|((3-2)∼(6/2));
```

because the arithmetic operators have precedence over concatenation, and would result in a 2x2 matrix. See also section 7.7.

```
2  4
1  3
```

```
let x[2,2] = 1 2 3 4;
```

The **let** command is used to initialize matrices with constant values. Unlike the concatenation operators, it cannot be used to define matrices in terms of expressions such as:

```
y = x1-x2∼x2|x3*3∼x4;
```

```
y = x[1:3,5:8];
```

will put the intersection of the first 3 rows and the fifth through eighth columns of x into the matrix y.

```
y = x[1 3 1,5 5 9];
```

will create a 3x3 matrix y with the intersection of the specified rows and columns pulled from x (in the indicated order).

```
let r = 1 3 1;
let c = 5 5 9;
y = x[r,c];
```

will have the same effect as the example above, but is more general.

```
y[2,4] = 3;
```

will set the 2,4 element of the existing matrix y to 3. This statement is illegal if y does not have at least 2 rows and 4 columns.

```
x = con(3,2);
```

will cause a ? to be printed on the screen, and will prompt the user until 6 numbers have been entered from the keyboard.

```
load x[] = b:mydata.asc
```

will load data contained in an ASCII file into an Nx1 vector x (use **rows(**x**)** to find out how many numbers were loaded, and use **reshape(**x**,N,K)** to reshape it to an NxK matrix).

```
load x;
```

will load the matrix **x.fmt** from disk (using the current load path) into the matrix x in memory.

```
open d1 = dat1;
x = readr(d1,100);
```

will read the first 100 rows of the **GAUSS** data set `dat1.dat`.

7.6.3 Strings and String Arrays

Strings

Strings can be used to store the names of files to be opened, messages to be printed, entire files, or whatever else you might need. Any byte value is legal in a string from 0–255. The buffer where a string is stored always contains a terminating byte of ASCII 0. This allows passing strings as arguments to C functions through the foreign language interface. Here is a partial list of the functions for manipulating strings.

$+	combines two strings into one long string.
^	interpret following name as a variable, not a literal.
chrs	convert vector of ASCII codes to character string.
ftocv	character representation of numbers in NxK matrix.
ftos	character representation of numbers in 1x1 matrix.
getf	load ASCII or binary file into string.
indcv	find index of element in character vector.
lower	convert to lowercase.
stof	convert string to floating point.
strindx	find index of a string within a second string.
strlen	length of a string.
strsect	extract substring of string.
upper	convert to uppercase.
vals	convert from string to numeric vector of ASCII codes.

Strings can be created like this:

```
x = "example string";
```

or

```
x = cons;                  /* keyboard input */
```

or

```
x = getf("myfile",0);      /* read a file into a string */
```

They can be printed like this:

```
print x;
```

A character matrix must have a '**$**' prefixed to it in a **print** statement.

```
print $x;
```

A string can be saved to disk with the **save** command in a file with a `.fst` extension and then loaded with the **load** command as follows:

```
save x;
loads x;
```

or

```
loads x=x.fst;
```

The backslash is used as the escape character inside double quotes to enter special characters. *backslash escape_character*

"\b"	backspace (ASCII 8)
"\e"	escape (ASCII 27)
"\f"	formfeed (ASCII 12)
"\g"	beep (ASCII 7)
"\l"	line feed (ASCII 10)
"\r"	carriage return (ASCII 13)
"\t"	tab (ASCII 9)
"\\"	a backslash
"\###"	the ASCII character whose decimal value is "###".

When entering DOS pathnames in double quotes two backslashes must be used to insert one backslash.

```
st = "c:\\gauss\\myprog.prg";
```

An important use of strings (and also character elements of matrices; see below) is with the substitution, or caret (^), operator.

In a command like the following:

```
create f1 = olsdat with x,4,2;
```

by default **GAUSS** will interpret the **olsdat** as a literal, i.e., the literal name of the **GAUSS** data file that you want to create. It will also interpret the **x** as the literal prefix string for the variable names: **x1 x2 x3 x4**.

If you want to get the data set name from a string variable, the (^) caret operator could be used as:

```
dataset="olsdat";
create f1=^dataset with x,4,2;
```

If you want to get the data set name from a string variable and the variable names from a character vector, then use the following:

```
dataset="olsdat";
let vnames=age pay sex;
create f1=^dataset with ^vnames,0,2;
```

The (^) caret operator works with **load**, **save**, and **dos** also.

```
lpath="/gauss/procs";
name="mydata";
load path=^lpath x=^name;
command="dir *.fmt";
dos ^command;
```

The general syntax is:

^*variable_name*

Expressions are not allowed. The following commands are supported with the substitution operator.

```
create f1=^dataset with ^vnames,0,2;
create f1=^dataset using ^cmdfile;
open f1=^dataset;
output file=^outfile;
load x=^datafile;
load path=^lpath x,y,z,t,w;
loadexe buf=^exefile;
save ^name=x;
save path=^spath;
dos ^cmdstr;
run ^prog;
msym ^mstring;
```

String Arrays

String arrays are NxK matrices of strings. Here is a partial list of the functions for manipulating string arrays:

$|	vertical string array concatenation operator.
$~	horizontal string array concatenation operator.
[ ]	extract subarrays or individual strings from their corresponding array, or assign their values.
'	transpose operator.
.'	bookkeeping transpose operator.
declare	initializes variables at compile time.
delete	deletes specified global symbols.
fgetsa	reads multiple lines of text from a file.
fgetsat	reads multiple lines of text from a file, discarding newlines.
format	defines output format for matrices, string arrays, and strings.
fputs	writes strings to a file.
fputst	writes strings to a file, appending newlines.
let	initializes matrices, strings, and string arrays.
loads	loads a string or string array file (`.fst file`).
lprint	prints expressioins to the printer.
lshow	prints global symbol table to the printer.
print	prints expressions on screen and/or auxiliary output.
reshape	reshapes a matrix or string array to new dimensions.
save	saves matrix, string array, string, procedure, function or keyword to disk and gives the disk file either a `.fmt`, `.fst` or `.fcg` extension.
show	displays global symbol table.
sortcc	quick-sorts rows of matrix or string array based on character column.
type	indicates whether variable passed as argument is matrix, string, or string array.
typecv	indicates whether variables named in argument are strings, string arrays, matrices, procedures, functions or keywords.
varget	accesses the global variable named by a string array.
varput	assigns the global variable named by a string array.
vec	stacks columns of a matrix or string array to form a column vector.
vecr	stacks rows of a matrix or string array to form a column vector.

String arrays are created through the use of the string array concatenation operators. Below is a contrast of the horizontal string and horizontal string array concatenation operators.

```
x = "age";
y = "pay";
n = "sex";
s = x $+ y $+ n;
sa =  x $~ y $~ n;


s = agepaysex

sa = age      pay      sex
```

7.6.4 Character Matrices

Matrices can have either numeric or character elements. For convenience we refer to a matrix containing character elements as a character matrix.

A character matrix is not a separate data type but just gives you the ability to store and manipulate data elements that are composed of ASCII characters as well as floating point numbers. For example, you may want to concatenate a column vector containing the names of the variables in an analysis onto a matrix containing the coefficients, standard errors, t-statistic, and p-value. You can then print out the whole matrix with a separate format for each column with one call to the function **printfm**.

The logic of the programs will dictate the type of data that is assigned to a matrix, and the increased flexibility allowed by being able to bundle both types of data together in a single matrix can be very powerful. You could, for instance, create a moment matrix from your data, concatenate a new row onto it containing the names of the variables and save it to disk with the **save** command.

Numeric matrices are double precision, which means that each element is stored in 8 bytes. A character matrix can thus have elements of up to 8 characters. Character matrices are ideal for the manipulation of the names of the columns in a data matrix, which are limited to 8 characters.

GAUSS does not automatically keep track of whether a matrix contains character or numeric information. The ASCII to **GAUSS** conversion program **atog** will record the types of variables in a data set when it creates it. The **create** command will, too. The function **vartypef** gets a vector of variable type information from a data set. This vector of ones and zeros can be used by **printfm** when printing your data. Since **GAUSS** does not know whether a matrix has character or numeric information, it is up to the user to specify which type of data it contains when printing the contents of the matrix. See **print** and **printfm** for details.

Most functions that take a string argument will take an element of a character matrix also, interpreting it as a string of up to 8 characters.

7.6.5 Special Data Types

The IEEE floating point format has many encodings that have special meaning. The **print** command will print them accurately so that you can tell if your calculation is producing results that are meaningful.

NaN

There are many floating point encodings which do not correspond to a real number. These encodings are referred to as NaN's. NaN stands for *Not A Number*.

Certain numerical errors will cause the math coprocessor to create a NaN called an **indefinite**. This will be printed as a **-NaN** when using the **print** command. These values are created by the following operations.

- $+\infty$ plus $-\infty$
- $+\infty$ minus $+\infty$
- $-\infty$ minus $-\infty$
- $0 \times \infty$
- ∞/∞
- 0 / 0
- Operations where one or both operands is a NaN
- Trigonometric functions involving ∞

INF

When the math coprocessor overflows, the result will be a properly signed infinity. Most subsequent calculations will not deal with infinities very well, and it is usually a signal of an error in your program. The result of an operation involving an infinity is usually a NaN.

DEN, UNN

When some math coprocessors underflow, they may do so gradually by shifting the significand of the number as necessary to keep the exponent in range. The result of this

is a denormal (DEN). When denormals are used in calculations, they are usually handled automatically in an appropriate way. The result will either be an unnormal (UNN), which like the denormal represents a number very close to zero, or a normal, depending on how significant the effect of the denormal was in the calculation. In some cases the result will be a NaN. This can happen in **GAUSS** for DOS when the value that is in a math coprocessor register in 80-bit format is an unnormal and has an exponent that is within the range of the 64-bit double precision format. Therefore, sometimes multiplying an unnormal by another number will result in a NaN. Following are some procedures for dealing with these values.

The procedure **isindef** will return 1 (true) if the matrix passed to it contains any NaN's that are the **indefinite** mentioned above. The **GAUSS** missing value code as well as **GAUSS** scalar error codes are NaN's, but this procedure tests only for **indefinite**.

```
proc isindef(x);
    retp(not x $/= __INDEFn);
endp;
```

Be sure to call **gausset** before calling **isindef**. **gausset** will initialize the value of the global **__INDEFn** to a platform specific encoding.

The procedure **normal** will return a matrix with all denormals and unnormals set to zero.

```
proc normal(x);
    retp(x .* (abs(x) .> 4.19e-307));
endp;
```

This procedure, **isinf**, will return 1 (true) if the matrix passed to it contains any infinities.

```
proc isinf(x);
    local plus,minus;
    plus = __INFp;
    minus = __INFn;
    retp(not x /= plus or not x /= minus);
endp;
```

Be sure to call **gausset** before calling **isinf**. **gausset** will initialize the value of the globals **__INFn** and **__INFp** to platform specific encodings.

7.7 Operator Precedence

The order in which an expression is evaluated is determined by the precedence of the operators involved and the order in which they are used. For example, the ***** and **/**

operators have a higher precedence than the **+** and **-** operators. In expressions which contain the above operators, the operand pairs associated with the ***** or **/** operator are evaluated first. Whether ***** or **/** is evaluated first depends on which comes first in the particular expression.

The precedence of all operators is listed in Appendix D.

The expression

-5+3/4+6*3

is evaluated as

$$(-5) + (3/4) + (6 * 3)$$

Within a term, operators of equal precedence are evaluated from left to right.

The term

2^3^7

is evaluated

$$(2^3)^7$$

In the expression

f1(x)*f2(y)

the function **f1** is evaluated before **f2**.

Here are a few examples:

expression	evaluation
a+b*c+d	$(a + (b * c)) + d$
-2+4-6*inv(8)/9	$((-2) + 4) - ((6 * inv(8))/9)$
3.14^5*6/(2+sqrt(3)/4)	$((3.14^5) * 6)/(2 + (sqrt(3)/4))$
-a+b*c^2	$(-a) + (b * (c^2))$
a+b-c+d-e	$(((a + b) - c) + d) - e$
a^b^c*d	$((a^b)^c) * d$
a*b/d*c	$((a * b)/d) * c$
a^b+c*d	$(a^b) + (c * d)$
2^4!	$(2^4)!$
2*3!	$2 * (3!)$

7.8 Flow Control

A computer language needs facilities for decision making and looping to control the order in which computations are done. **GAUSS** has several kinds of flow control statements.

7.8.1 Looping

do loop

The **do** statement can also be used in **GAUSS** to control looping.

do while *scalar_expression*; Loop if expression is true.

⋮
statements
⋮

endo;

do until *scalar_expression*; Loop if expression is false.

⋮
statements
⋮

endo;

The *scalar_expression* is any expression that returns a scalar result. The expression will be evaluated as $TRUE$ if its real part is nonzero and $FALSE$ if it is zero. There is no counter variable that is automatically incremented in a **do** loop. If one is used, it must be set to its initial value before the loop is entered and explicitly incremented or decremented inside the loop. The following example illustrates nested **do** loops that use counter variables.

```
format /rdn 1,0;
space = "     ";
comma = ",";
i = 1;
do while i <= 4;
    j = 1;
    do while j <= 3;
        print  space i comma j;;
        j = j+1;
    endo;
    i = i+1;
    print;
endo;
```

The example above will print the following.

```
1,1     1,2     1,3
2,1     2,2     2,3
3,1     3,2     3,3
4,1     4,2     4,3
```

Use the relational and logical operators without the dot '.' in the expression that controls a **do** loop. These operators always return a scalar result.

break and **continue** are used within **do** loops to control execution flow. When **break** is encountered, the program will jump to the statement following the **endo**. This terminates the loop. When **continue** is encountered, the program will jump up to the top of the loop and reevaluate the **while** or **until** expression. This allows you to reiterate the loop without executing any more of the statements inside the loop.

```
do until eof(fp);       /* continue jumps here */
    x = packr(readr(fp,100));
    if scalmiss(x);
        continue;       /* iterate again */
    endif;
    s = s + sumc(x);
    count = count + rows(x);
    if count >= 10000;
        break;          /* break out of loop */
    endif;
endo;
mean = s / count;       /* break jumps here */
```

for loop

The fastest looping construct in **GAUSS** is the **for** loop.

for *counter* **(***start***,** *stop***,** *step***);**

⋮

statements

⋮

endfor;

counter is the literal name of the counter variable. *start*, *stop* and *step* are scalar expressions. *start* is the initial value, *stop* is the final value and *step* is the increment. **break** and **continue** are also supported by **for** loops. For more information see **for** in the *COMMAND REFERENCE*.

7.8.2 Conditional Branching

The **if** statement controls conditional branching.

if *scalar_expression*;
    ⋮
    statements
    ⋮
elseif *scalar_expression*;
    ⋮
    statements
    ⋮
else;
    ⋮
    statements
    ⋮
endif;

The *scalar_expression* is any expression that returns a scalar result. The expression will be evaluated as $TRUE$ if its real part is nonzero and $FALSE$ if it is zero.

GAUSS will test the expression after the **if** statement. If it is $TRUE$, then the first list of statements is executed. If it is $FALSE$, 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.

Use the relational and logical operators without the dot '**.**' in the expression that controls an **if** or **elseif** statement. These operators always return a scalar result.

if statements can be nested.

One **endif** is required per **if** clause. If an **else** statement is used, there may be only one per **if** clause. There may be as many **elseif**'s as are required. There need not be any **elseif**'s or any **else** statement within an **if** clause.

7.8.3 Unconditional Branching

The **goto** and **gosub** statements control unconditional branching. The target of both a **goto** and a **gosub** is a label.

goto

A **goto** is an unconditional jump to a label with no return.

```
label:
 .
 .
goto label;
```

Parameters can be passed with a **goto**. The number of parameters is limited by available stack space. This is handy for common exit routines.

```
        .
        .
    goto errout("Matrix singular");
        .
        .
    goto errout("File not found");
        .
        .
errout:
    pop errmsg;
    errorlog errmsg;
    end;
```

gosub

With a **gosub**, the address of the **gosub** statement is remembered and when a **return** statement is encountered, the program will resume executing at the statement following the **gosub**.

Parameters can be passed with a **gosub** in the same way as a **goto**. With a **gosub** it is also possible to return parameters with the **return** statement.

Subroutines are not isolated from the rest of your program and the variables referred to between the label and the **return** statement can be accessed from other places in your program.

Since a subroutine is only an address marked by a label, there can be subroutines inside of procedures. The variables that are used in these subroutines are the same variables that are known inside the procedure. They will not be unique to the subroutine, but they may be locals that are unique to the procedure that the subroutine is in. See **gosub** in the *COMMAND REFERENCE* for the details.

7.9 Functions

Single line functions that return one item can be defined with the **fn** statement.

```
fn area(r) = pi * r * r;
```

These functions can be called in the same way as intrinsic functions. The above function could be used in the following program sequence.

```
diameter = 3;
radius = 3 / 2;
a = area(radius);
```

7.10 Rules of Syntax

This section lists the general rules of syntax for **GAUSS** programs.

7.10.1 Statements

A **GAUSS** program consists of a series of statements. A statement is a complete expression or command. Statements in **GAUSS** end with a semicolon. See your supplement for the maximum statement length in this implementation.

There is one exception to the rule that statements must end with a semicolon. From the **GAUSS** command line, the final semicolon in an interactive program is implicit if it is not explicitly given.

```
(gauss) x=5; z=rndn(3,3); y=x+z
```

Column position is not significant. Blank lines are allowed. Inside a statement and outside of double quotes, the carriage return/line feed at the end of a physical line will be converted to a space character as the program is compiled.

A statement containing a quoted string can be continued across several lines with a backslash as follows.

```
s = "This is one really long string that would be "\
    "difficult to assign in just a single line.";
```

7.10.2 Case

GAUSS does not distinguish between uppercase and lowercase except inside double quotes.

With some operating systems, file names are case sensitive.

7.10.3 Comments

```
/* This kind of comment can be nested */

@ This kind cannot be nested @
```

7.10.4 Extraneous Spaces

Extraneous spaces are significant in **print** and **lprint** statements where the space is a delimiter between expressions.

```
print x y z;
```

In **print** and **lprint** statements, spaces can be used in expressions that are in parentheses.

```
print (x * y)  (x + y);
```

7.10.5 Symbol Names

The names of matrices, strings, procedures, and functions can be up to 32 characters long. The characters must be alphanumeric or the underscore. The first character must be alphabetic or an underscore.

7.10.6 Labels

A label is used as the target of a **goto** or a **gosub**. The rule for naming labels is the same as for matrices, strings, procedures, and functions. A label is followed immediately by a colon.

```
here:
```

The reference to a label does not use a colon.

```
goto here;
```

7.10.7 Assignment Statements

The equal sign '=' is the assignment operator.

```
y = x + z;
```

Multiple assignments must be enclosed in braces.

```
{ mant,pow } = base10(x);
```

The (equal to) comparison operator is two equal signs '==='.

```
if x == y;
    print "x is equal to y";
endif;
```

7.10.8 Function Arguments

The arguments to functions are enclosed in parentheses.

```
y = sqrt(x);
```

7.10.9 Indexing Matrices

Brackets '**[ ]**' are used to index matrices.

```
x = { 1 2 3,
      3 7 5,
      3 7 4,
      8 9 5,
      6 1 8 };

y = x[3,3];
z = x[1 2:4,1 3];
```

Vectors can be indexed with either 1 or 2 indices.

```
v = { 1 2 3 4 5 6 7 8 9 };
k = v[3];
j = v[1,6:9];
```

x[2,3] returns the element in the second row and the third column of **x**.

x[1 3 5,4 7] returns the submatrix that is the intersection of rows 1, 3 and 5 and columns 4 and 7.

x[.,3] returns the third column of **x**.

x[3:5,.] returns the submatrix containing the third through the fifth rows of **x**.

The indexing operator will take vector arguments for submatrix extraction or submatrix assignments.

y = x[rv,cv];

y[rv,cv] = x;

rv and **cv** can be any expressions returning vectors or matrices. The elements of **rv** will be used as the row indices and the elements of **cv** will be used as the column indices. If **rv** is a scalar 0, then all rows will be used and if **cv** is a scalar 0, then all columns will be used. If a vector is used in an index expression, then it is illegal to use the space operator or the colon operator on the same side of the comma as the vector.

7.10.10 Arrays of Matrices and Strings

It is possible to index sets of matrices or strings using the **varget** function. Consider the following example.

```
mvec = { x y z a };
i = 2;
g = inv(varget(mvec[i]));
```

In this example, a set of matrix names is assigned to **mvec**. The name **y** is indexed from **mvec** and passed to **varget** which will return the global matrix **y**. The returned matrix is inverted and assigned to **g**.

The following procedure can be used to index the matrices in **mvec** more directly.

```
proc imvec(i);
    retp(varget(mvec[i]));
endp;
```

Then **imvec(i)** will equal the matrix whose name is in the i^{th} element of **mvec**.

In the example above, the procedure **imvec()** was written so that it always operates on the vector **mvec**. The following procedure makes it possible to pass in the vector of names being used.

```
proc get(array,i);
    retp(varget(array[i]));
endp;
```

Then **get(mvec,3)** will return the 3^{rd} matrix listed in **mvec**.

```
proc put(x,array,i);
    retp(varput(x,array[i]));
endp;
```

And **put(x,mvec,3)** will assign **x** to the 3^{rd} matrix listed in **mvec** and return a 1 if successful or a 0 if it fails.

7.10.11 Arrays of Procedures

It is also possible to index procedures. The ampersand (**&**) operator is used to return a pointer to a procedure.

As an example, assume that **f1**, **f2**, and **f3** are procedures that take a single argument. The following code defines a procedure **fi** that will return the value of the i^{th} procedure, evaluated at **x**.

```
nms = &f1 | &f2 | &f3;

proc fi(x,i);
    local f;
    f = nms[i];
    local f:proc;
    retp( f(x) );
endp;
```

For instance, **fi(x,2)** will return **f2(x)**. The ampersand is used to return the pointers to the procedures. **nms** is just a numeric vector that contains a set of pointers. The **local** statement is used twice. The first one tells the compiler that **f** is a local matrix. The i^{th} pointer (which is just a number) is assigned to **f**. Then the second **local** statement tells the compiler to treat **f** as a procedure from this point on, and so the subsequent statement **f(x)** is interpreted as a procedure call.

Chapter 8

Operators

8.1 Element-by-Element Operators

The operators described as **element-by-element** operators share common rules of conformability. Some functions that have two arguments also operate according to the same rules.

The element-by-element operators handle those situations in which matrices are not conformable according to standard rules of matrix algebra. When a matrix is said to be **ExE conformable**, it refers to this element-by-element conformability. The following cases are supported.

matrix	**op**	*matrix*
matrix	**op**	*scalar*
scalar	**op**	*matrix*
matrix	**op**	*vector*
vector	**op**	*matrix*
vector	**op**	*vector*

In a typical expression involving an element-by-element operator:

$$z = x + y;$$

conformability is defined as follows.

- If x and y are the same size, the operations are carried out corresponding element by corresponding element.

$$x = \begin{matrix} 1 & 3 & 2 \\ 4 & 5 & 1 \\ 3 & 7 & 4 \end{matrix}$$

$$y = \begin{matrix} 2 & 4 & 3 \\ 3 & 1 & 4 \\ 6 & 1 & 2 \end{matrix}$$

$$z = \begin{matrix} 3 & 7 & 5 \\ 7 & 6 & 5 \\ 9 & 8 & 6 \end{matrix}$$

- If x is a matrix and y is a scalar, or vice versa, then the scalar is operated on with respect to every element in the matrix. For example, *x + 2* will add 2 to every element of x.

$$x = \begin{matrix} 1 & 3 & 2 \\ 4 & 5 & 1 \\ 3 & 7 & 4 \end{matrix}$$

$$y = \quad 2$$

$$z = \begin{matrix} 3 & 5 & 4 \\ 6 & 7 & 3 \\ 5 & 9 & 6 \end{matrix}$$

- If x is an Nx1 column vector and y is an NxK matrix, or vice versa, the vector is swept "across" the matrix.

vector		matrix		
1	$\longrightarrow$	2	4	3
4	$\longrightarrow$	3	1	4
3	$\longrightarrow$	6	1	2

result		
3	5	4
7	5	8
9	4	5

- If x is an 1xK column vector and y is an NxK matrix, or vice versa, then the vector is swept "down" the matrix.

vector	2	4	3
	↓	↓	↓
	2	4	3
matrix	3	1	4
	6	1	2
	4	8	6
result	5	5	7
	8	5	5

- When one argument is a row vector and the other is a column vector, the result of an element-by-element operation will be the **table** of the two.

row vector		2	4	3	1
	3	5	7	6	4
column vector	2	4	6	5	3
	5	7	9	8	6

If x and y are such that none of these conditions apply, then the matrices are not conformable to these operations and an error message will be generated.

8.2 Matrix Operators

The following operators work on matrices. Some assume numeric data and others will work on either character or numeric data.

8.2.1 Numeric Operators

See Section 8.1 for details on how matrix conformability is defined for element-by-element operators.

\+ Addition:

$$y = x + z;$$

Performs element-by-element addition.

\- Subtraction or negation:

$y = x - z;$

$y = -k;$

Performs element-by-element subtraction or the negation of all elements, depending on context.

* Matrix multiplication or multiplication:

$y = x * z;$

When z has the same number of rows as x has columns, this will perform matrix multiplication (inner product). If x or z are scalar, this performs standard element-by-element multiplication.

/ Division or linear equation solution.

$x = b / A;$

If A and b are scalars, it performs standard division. If one of the operands is a matrix and the other is scalar, the result is a matrix the same size with the results of the divisions between the scalar and the corresponding elements of the matrix. Use **./** for element-by-element division of matrices.

If b and A are conformable, this operator solves the linear matrix equations.

$$Ax = b$$

Linear equation solution is performed in the following cases.

- If A is a square matrix and has the same number of rows as b, then this statement will solve the system of linear equations using an LU decomposition.
- If A is rectangular with the same number of rows as b, then this statement will produce the least squares solutions by forming the normal equations and using the Cholesky decomposition to get the solution.

  $$y = \frac{A'b}{A'A}$$

  If **trap 2** is set, missing values will be handled with pairwise deletion.

  For least squares solutions on OS/2 and DOS, all computations and storage of intermediate results are done in 80-bit precision. **prcsn 64** can be specified to force 64-bit precision for smaller matrices or when using these platforms.

% Modulo division:

$y = x$ **%** z**;**

For integers, this returns the integer value that is the remainder of the integer division of x by z. If x or z are noninteger, they will first be rounded to the nearest integer. This is an element-by-element operator.

! Factorial:

$y = x$**!;**

Computes the factorial of every element in the matrix x. Nonintegers are rounded to the nearest integer before the factorial operator is applied. This will not work with complex matrices. If x is complex, a fatal error will be generated.

.* Element-by-element multiplication:

$y = x$ **.*** z**;**

If x is a column vector, and z is a row vector (or vice versa), then the "outer product" or "table" of the two will be computed. See Section 8.1 for conformability rules.

./ Element-by-element division:

$y = x$ **./** z**;**

^ Element-by-element exponentiation.

$y = x$**^**z**;**

If x is negative, z must be an integer.

.^ Same as **^**

.*. Kronecker (tensor) product:

$y = x$ **.*.** z**;**

This results in a matrix in which every element in x has been multiplied (scalar multiplication) by the matrix z. For example:

```
x = { 1 2,
      3 4 };
z = { 4 5 6,
      7 8 9 };
y = x .*. z;
```

$$x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

$$z = \begin{matrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

$$y = \begin{matrix} 4 & 5 & 6 & 8 & 10 & 12 \\ 7 & 8 & 9 & 14 & 16 & 18 \\ 12 & 15 & 18 & 16 & 20 & 24 \\ 21 & 24 & 27 & 28 & 32 & 36 \end{matrix}$$

$*\sim$ Horizontal direct product.

$z = x *\sim y;$

$$x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

$$y = \begin{matrix} 5 & 6 \\ 7 & 8 \end{matrix}$$

$$z = \begin{matrix} 5 & 6 & 10 & 12 \\ 21 & 24 & 28 & 32 \end{matrix}$$

The input matrices x and y must have the same number of rows. The result will have **cols(**x**) * cols(**y**)** columns.

8.2.2 Other Matrix Operators

′ Transpose operator

$y = x';$

In the expression above the columns of y will contain the same values as the rows of x and the rows of y will contain the same values as the columns of x. For complex matrices this computes the complex conjugate transpose.

If an operand immediately follows the transpose operator, the ′ will be interpreted as **′***. Thus **y = x′x** is equivalent to **y = x′*x**.

.′ Bookkeeping transpose operator

$y = x.';$

This is provided primarily as a matrix handling tool for complex matrices. For all matrices, the columns of y will contain the same values as the rows of x and the rows of y will contain the same values as the columns of x. The complex conjugate transpose is NOT computed when you use .′.

If an operand immediately follows the bookkeeping transpose operator, the .′ will be interpreted as **.′***. Thus **y = x.′x** is equivalent to **y = x.′*x**.

| Vertical concatenation

$z = x \mid y;$

$$x = \begin{matrix} 1 & 2 & 3 \\ 3 & 4 & 5 \end{matrix}$$

$$y = \begin{matrix} 7 & 8 & 9 \end{matrix}$$

$$z = \begin{matrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 7 & 8 & 9 \end{matrix}$$

~ Horizontal concatenation

$z = x \sim y;$

$$x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

$$y = \begin{matrix} 5 & 6 \\ 7 & 8 \end{matrix}$$

$$z = \begin{matrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{matrix}$$

8.3 Relational Operators

See Section 8.1 for details on how matrix conformability is defined for element-by-element operators.

Each of these operators has two equivalent representations. Either can be used (e.g., < or **lt**), depending only upon preference. The alphabetic form should be surrounded by spaces.

A third form of these operators has a '**$**' and is used for comparisons between character data and for comparisons between strings or string arrays. The comparisons are done byte by byte starting with the lowest addressed byte of the elements being compared.

The equality comparison operators (<=, ==, >=, /=) and their dot equivalents can be used to test for missing values and the NaN that is created by floating point exceptions. Less than and greater than comparisons are not meaningful with missings or NaN's, but equal and not equal will be valid. These operators are sign-insensitive for missings, NaN's, and zeros.

The string '**$**' versions of these operators can also be used to test missings, NaN's and zeros. Because they do a strict byte-to-byte comparison, they are sensitive to the sign

bit. Missings, NaN's, and zeros can all have the sign bit set to 0 or 1, depending on how they were generated and have been used in a program.

If the relational operator is NOT preceded by a dot '**.**', then the result is always a scalar 1 or 0, based upon a comparison of all elements of x and y. All comparisons must be true for the relational operator to return $TRUE$.

By this definition, then

if x **/=** y**;**

is interpreted as: "is every element of x not equal to the corresponding element of y". To check if two matrices are not identical, use:

if not x **==** y**;**

For complex matrices, the **==**, **/=**, **.==** and **./=** operators compare both the real and imaginary parts of the matrices; all other relational operators compare only the real parts.

- Less than

  $z = x$ **<** y**;**

  $z = x$ **lt** y**;**

  $z = x$ **$<** y**;**

- Less than or equal to

  $z = x$ **<=** y**;**

  $z = x$ **le** y**;**

  $z = x$ **$<=** y**;**

- Equal to

  $z = x$ **==** y**;**

  $z = x$ **eq** y**;**

  $z = x$ **$==** y**;**

- Not equal

  $z = x$ **/=** y**;**

  $z = x$ **ne** y**;**

  $z = x$ **$/=** y**;**

- Greater than or equal to

  $z = x >= y;$

  $z = x$ **ge** $y;$

  $z = x$ **$>=** $y;$

- Greater than

  $z = x > y;$

  $z = x$ **gt** $y;$

  $z = x$ **$>** $y;$

If the relational operator IS preceded by a dot '**.**', then the result will be a matrix of 1's and 0's, based upon an element-by-element comparison of x and y.

- Element-by-element less than

  $z = x$ **.<** $y;$

  $z = x$ **.lt** $y;$

  $z = x$ **.$<** $y;$

- Element-by-element less than or equal to

  $z = x$ **.<=** $y;$

  $z = x$ **.le** $y;$

  $z = x$ **.$<=** $y;$

- Element-by-element equal to

  $z = x$ **.==** $y;$

  $z = x$ **.eq** $y;$

  $z = x$ **.$==** $y;$

- Element-by-element not equal to

  $z = x$ **./=** $y;$

  $z = x$ **.ne** $y;$

  $z = x$ **.$/=** $y;$

- Element-by-element greater than or equal to

  $z = x$ **.>=** $y;$

$z = x$ **.ge** y**;**

$z = x$ **.$>=** y**;**

- Element-by-element greater than

  $z = x$ **.>** y**;**

  $z = x$ **.gt** y**;**

  $z = x$ **.$>** y**;**

8.4 Logical Operators

The logical operators perform logical or Boolean operations on numeric values. On input a nonzero value is considered TRUE and a zero value is considered $FALSE$. The logical operators return a 1 if $TRUE$ and a 0 if $FALSE$. Decisions are based on the following truth table.

Complement

X	**not** X
T	F
F	T

Conjunction

X	Y	X **and** Y
T	T	T
T	F	F
F	T	F
F	F	F

Disjunction

X	Y	X **or** Y
T	T	T
T	F	T
F	T	T
F	F	F

Exclusive Or

X	Y	X **xor** Y
T	T	F
T	F	T
F	T	T
F	F	F

Equivalence

X	Y	X **eqv** Y
T	T	T
T	F	F
F	T	F
F	F	T

For complex matrices, the logical operators consider only the real part of the matrices.

The following operators require scalar arguments. These are the ones to use in **if** and **do** statements.

- Complement

  $z =$ **not** x**;**

- Conjunction

  $z = x$ **and** y**;**

- Disjunction

  $z = x$ **or** y**;**

- Exclusive or

  $z = x$ **xor** y**;**

- Equivalence

  $z = x$ **eqv** y**;**

If the logical operator is preceded by a dot '**.**', the result will be a matrix of 1's and 0's based upon an element-by-element logical comparison of x and y.

- Element-by-element logical complement

  $z =$ **.not** x**;**

- Element-by-element conjunction

  $z = x$ **.and** y**;**

- Element-by-element disjunction

  $z = x$ **.or** y**;**

- Element-by-element exclusive or

  $z = x$ **.xor** y**;**

- Element-by-element equivalence

  $z = x$ **.eqv** y**;**

8.5 Other Operators

Assignment Operator

Assignments are done with one equal sign.

```
y = 3;
```

(comma)

Commas are used to delimit lists.

```
clear x,y,z;
```

Commas are also used to separate row indices from column indices within brackets.

```
y = x[3,5];
```

And to separate arguments of functions within parentheses.

```
y = momentd(x,d);
```

(period)

Dots are used in brackets to signify "all rows" or "all columns".

```
y = x[.,5];
```

(space)

Spaces are used inside of index brackets to separate indices.

```
y = x[1 3 5,3 5 9];
```

No extraneous spaces are allowed immediately before or after the comma, or immediately after the left bracket or before the right bracket.

Spaces are also used in **print** and **lprint** statements to separate the separate expressions to be printed.

```
print x/2 2*sqrt(x);
```

Therefore, no extraneous spaces are allowed within expressions in **print** or **lprint** statements unless the expression is enclosed in parentheses.

```
print (x / 2) (2 * sqrt(x));
```

(colon)

A colon is used within brackets to create a continuous range of indices.

$y = x$**[1:5,.];**

(ampersand)

The (**&**) ampersand operator will return a pointer to a procedure (**proc**) or function (**fn**). It is used when passing procedures or functions to other functions and for indexing procedures. See Section 9.5.

String Concatenation

```
x = "dog";
y = "cat";
z = x $+ y;
print z;

dogcat
```

If the first argument is of type STRING, the result will be of type STRING. If the first argument is of type MATRIX, the result will be of type MATRIX. See the following examples:

```
y = 0 $+ "caterpillar";
```

The result will be a 1x1 matrix containing **caterpil**.

```
y = zeros(3,1) $+ "cat";
```

The result will be a 3x1 matrix, each element containing **cat**.

If we use the y created above in the following:

```
k = y $+ "fish";
```

The result will be a 3x1 matrix with each element containing **catfish**.

If we then use k created above:

```
t = "" $+ k[1,1];
```

The result will be a STRING containing **catfish**. If we used the same k to create z as follows:

```
z = "dog" $+ k[1,1];
```

The resulting z will be a string containing **dogcatfish**.

String Array Concatenation

$| vertical string array concatenation.

```
x = "dog";
y = "fish";
k = x $| y;
print k;

dog
fish
```

$~ horizontal string array concatenation.

```
x = "dog";
y = "fish";
k = x $~ y;
print k;

dog      fish
```

String Variable Substitution

In a command like the following:

```
create f1 = olsdat with x,4,2;
```

by default **GAUSS** will interpret **olsdat** as the literal name of the GAUSS data file that you want to create. It will also interpret **x** as the literal prefix string for the variable names: **x1 x2 x3 x4**.

If you want to get the data set name from a string variable, the substitution (^) operator could be used as follows:

```
dataset = "olsdat";
create f1 = ^dataset with x,4,2;
```

If you want to get the data set name from a string variable and the variable names from a character vector, use the following:

```
dataset = "olsdat";
vnames = { age, pay, sex };
create f1 = ^dataset with ^vnames,0,2;
```

The general syntax is:

^*variable_name*

Expressions are not allowed.

The following commands are supported with the substitution operator in the current version.

```
create f1 = ^dataset with ^vnames,0,2;
create f1 = ^dataset using ^cmdfile;
open f1 = ^dataset;
output file = ^outfile;
load x = ^datafile;
load path = ^lpath x,y,z,t,w;
loadexe buf = ^exefile;
save ^name = x;
save path = ^spath;
dos ^cmdstr;
run ^prog;
msym ^mstring;
```

8.6 Using Dot Operators with Constants

When you use those operators preceded by a '**.**' (the *dot operators*) with a scalar integer constant, make sure to put a space between the constant and any following dot operator. Otherwise, the dot will be interpreted as part of the scalar, i.e., the decimal point, and you may get results other than you expect. For example:

```
let y = 1 2 3;
x = 2.<y;
```

will return x as a scalar 0, not a vector of 0's and 1's, because

```
x = 2.<y;
```

is interpreted as

```
x = 2. < y;
```

not

```
x = 2 .< y;
```

Most of the time, it will be the dot relational (**.<**, **.<=**, **.==**, **./=**, **.>**, **.>=**) operators that will trip you up. The same problem can occur with other dot operators, though, too. For example:

```
let x = 1 1 1;
y = x./2./x;
```

will return y as a scalar .5 rather than a vector of .5's, because

```
y = x./2./x;
```

is interpreted as

```
y = (x ./ 2.) / x;
```

not

```
y = (x ./ 2) ./ x;
```

The second division, then, is handled as a matrix division rather than an element-by-element division.

Chapter 9

Procedures and Keywords

Procedures are multiple-line, recursive functions that can have either local or global variables. Procedures allow a large computing task to be written as a collection of smaller tasks. These smaller tasks are easier to work with and can hide the details of their operation from the other parts of the program that do not need to know them. This makes programs easier to understand and easier to maintain.

A procedure in **GAUSS** is basically a user-defined function that can be used as if it were an intrinsic part of the language. They can be as small and simple or as large and complicated as necessary to perform whatever task you need done. Procedures allow people to build on their previous work and on the work of others rather than starting over again and again to perform related tasks.

Any intrinsic command or function may be used in a procedure, as well as any user-defined function or other procedure. Procedures can refer to any global variable (that is, any variable that is in the main symbol table and that can be shown with the **show** command). It is also possible to declare local variables within a procedure. These variables are known only inside the procedure they are defined in and cannot be accessed from other procedures or from the main level program code.

All labels and subroutines inside a procedure are local to that procedure and will not be confused with labels of the same name in other procedures.

9.1 Defining a Procedure

A procedure definition consists of five parts, four of which are denoted by explicit **GAUSS** commands:

1.	Procedure declaration	**proc** statement
2.	Local variable declaration	**local** statement
3.	Body of procedure	
4.	Return from procedure	**retp** statement
5.	End of procedure definition	**endp** statement

There is always one **proc** statement and one **endp** statement in a procedure definition. Any statements that come between these two statements are part of the procedure. Procedure definitions cannot be nested. **local** and **retp** statements are optional. There can be multiple **local** and **retp** statements in a procedure definition. Here is an example of a procedure definition:

```
proc (3) = regress(x, y);
    local xxi,b,ymxb,sse,sd,t;
    xxi = invpd(x'x);
    b = xxi * (x'y);
    ymxb = y-xb;
    sse = ymxb'ymxb/(rows(x)-cols(x));
    sd = sqrt(diag(sse*xxi));
    t = b./sd;
    retp(b,sd,t);
endp;
```

This could be used as a function that takes two matrix arguments and returns three matrices as a result. An example of how it might be used is:

```
{ b,sd,t } = regress(x,y);
```

The five parts of the procedure definition will now be discussed in more detail:

9.1.1 Procedure Declaration

The **proc** statement is the procedure declaration statement. The format is as follows:

proc ⟦**(***rets***)** **=**⟧ *name***(**⟦*arg1***,***arg2***,...***argN*⟧**);**

rets Optional constant, number of values returned by the procedure. Acceptable values here are 0-1023, the default is 1.

name The name of the procedure, up to 32 alphanumeric characters or an underscore, beginning with an alpha or an underscore.

arg# Names that will be used inside of the procedure for the arguments that are passed to the procedure when it is called. There can be 0-1023 arguments. These names will be known only in the procedure being defined. Other procedures can use the same names, but they will be separate entities.

9.1.2 Local Variable Declarations

The **local** statement is used to declare local variables. Local variables are variables known only to the procedure being defined. The names used in the argument list of the **proc** statement are always local. The format of the **local** statement is:

local x**,**y**,**f**:proc,**g**:fn,**z**,**h**:keyword;**

Local variables can be matrices or strings. If **:proc**, **:fn**, or **:keyword** follows the variable name in the **local** statement, the compiler will treat the symbol as if it were a procedure, function, or keyword respectively. This allows passing procedures, functions, and keywords to other procedures. See Section 9.4.

Variables that are global to the system (that is, variables that are listed in the global symbol table and can be shown with the **show** command) can be accessed by any procedure without any redundant declaration inside of the procedure. If you want to create variables that are known only to the procedure being defined, the names of these local variables must be listed in a **local** statement. Once a variable name is encountered in a **local** statement, further references to that name inside of the procedure will be to the local rather than to a global having the same name. See **clearg**, **varget**, and **varput** for ways of accessing globals from within procedures that have locals with the same name.

The **local** statement does not initialize (set to a value) the local variables. If they are not passed in as parameters, they must be assigned some value before they are accessed or the program will terminate with a "Variable not initialized" error.

All local and global variables are dynamically allocated and sized automatically during execution. Local variables, including those that were passed as parameters, can change in size during the execution of the procedure.

Local variables exist only when the procedure is executing and then disappear. Local variables cannot be listed with the **show** command.

The maximum number of locals is limited by stack space and the size of workspace memory. The limiting factor applies to the total number of active local symbols at any one time during execution. If **cat** has 10 locals and it calls **dog** which has 20 locals, then there are 30 active locals whenever **cat** is called. See your supplement for the maximum number of locals allowed in this implementation.

There can be multiple **local** statements in a procedure. They will affect only the code in the procedure that follows. Therefore, for example, it is possible to refer to a global **x** in a procedure and follow that with a **local** statement that declares a local **x**. In that case all subsequent references to **x** would be to the local **x**. This is not good programming practice, but it demonstrates the principle that the **local** statement affects only the code that is physically below it in the procedure definition. Another example of this is a symbol that is declared as a local and then declared as a local procedure or function later in the same procedure definition. This allows doing arithmetic on local function pointers before calling them, see Section 9.5.

9.1.3 Body of Procedure

The body of the procedure can have any **GAUSS** statements necessary to perform the task the procedure is being written for. Other user-defined functions and other procedures can be referenced as well as any global matrices and strings.

GAUSS procedures are recursive, so the procedure can call itself as long as there is logic in the procedure to prevent an infinite recursion. The process would otherwise terminate with either an "Insufficient workspace memory" message or a "Procedure calls too deep" message, depending on the space necessary to store the locals for each separate invocation of the procedure.

9.1.4 Returning from the Procedure

The return from the procedure is accomplished with the **retp** statement.

retp;

retp(*expression1***,***expression2***,...***expressionN***);**

The **retp** statement can have multiple arguments. The number of items returned must coincide with the number of *rets* in the **proc** statement.

If the procedure is being defined with no items returned, then the **retp** statement is optional. The **endp** statement which ends the procedure will generate an implicit **retp** with no objects returned. If the procedure returns one or more objects, there must be an explicit **retp** statement.

There can be multiple **retp** statements in a procedure, and they can be anywhere inside the body of the procedure.

9.1.5 End of Procedure Definition

The **endp** statement marks the end of the procedure definition.

endp;

An implicit **retp** statement that returns nothing is always generated here so it is impossible to run off the end of a procedure without returning. If the procedure was defined to return one or more objects, then executing this implicit return will result in a "Wrong number of returns" error message and the program will terminate.

9.2 Calling a Procedure

Procedures are called like this:

```
dog(i,j,k);               /* no returns */

y = cat(i,j,k);           /* one return */

{ x,y,z } = bat(i,j,k);   /* multiple returns */

call bat(i,j,k);          /* ignore any returns */
```

Procedures are called in the same way that intrinsic functions are called. The procedure name is followed by a list of arguments in parentheses. The arguments must be separated by commas.

If there is to be no return value then use:

```
proc (0) = dog(x,y,z);
```

when defining the procedure and use:

```
dog(ak,4,3);
```

or

```
call dog(ak,4,3);
```

when calling it.

The arguments passed to procedures can be complicated expressions involving calls to other functions and procedures. This calling mechanism is completely general. For example:

```
y = dog(cat(3*x,bird(x,y))-2,1);
```

is legal.

9.3 Keywords

A keyword, like a procedure, is a subroutine that can be called interactively or from within a **GAUSS** program. A keyword differs from a procedure in that a keyword accepts exactly one string argument, and returns nothing. Keywords can perform many tasks not as easily accomplished with procedures.

9.3.1 Defining a Keyword

A keyword definition is much like a procedure definition. Keywords always are defined with 0 returns and 1 argument. The beginning of a keyword definition is the **keyword** statement.

keyword *name*(*strarg*);

name The name of the keyword, up to 32 alphanumeric characters or an underscore, beginning with an alpha or an underscore.

strarg Name that will be used inside of the keyword for the argument that is passed to the keyword when it is called. There is always 1 argument. The name is known only in the keyword being defined. Other keywords can use the same name, but they will be separate entities. This will always be a string. If the keyword is called with no characters following the name of the keyword, this will be a null string.

The rest of the keyword definition is the same as a procedure definition as in Section 9.1 above. Keywords always return nothing. Any **retp** statements, if used, should be empty.

retp;

9.3.2 Calling a Keyword

When a keyword is called, every character up to the end of the statement, excluding the leading spaces, is passed to the keyword as one string argument.

```
keyword what(str);
    print "The argument is: ‘" str "’";
endp;
```

The keyword above will print the string argument passed to it. The argument will be printed enclosed in single quotes. For example if you type:

```
what is the argument;
```

The keyword will respond:

```
The argument is: ‘is the argument’
```

Here is another example:

```
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;
```

To use this keyword, type:

```
add 1 2 3 4 5;
```

It will respond with:

```
Sum is: 15.00
```

9.4 Passing Procedures to Procedures

Procedures and functions can be passed to procedures in the following way:

```
proc max(x,y);  /* procedure to return maximum */
    if x>y;
        retp(x);
    else;
        retp(y);
    endif;
endp;

proc min(x,y);  /* procedure to return minimum */
    if x<y;
        retp(x);
    else;
        retp(y);
    endif;
endp;

fn lgsqrt(x) = ln(sqrt(x));  /* function to return
                             :: log of square root
                             */
```

```
proc myproc(&f1,&f2,x,y);
    local f1:proc, f2:fn, z;
    z = f1(x,y);
    retp(f2(z));
endp;
```

The procedure **myproc** takes 4 arguments. The first is a procedure **f1** that has 2 arguments. The second is a function **f2** that has one argument. It also has two other arguments that must be matrices or scalars. In the **local** statement, **f1** is declared to be a procedure and **f2** is declared to be a function. They can be used inside the procedure in the usual way. **f1** will be interpreted as a procedure inside of **myproc**, and **f2** will be interpreted as a function. The call to **myproc** is made as follows:

```
k = myproc(&max,&lgsqrt,5,7);  /* log of square root of 7 */

k = myproc(&min,&lgsqrt,5,7);  /* log of square root of 5 */
```

The ampersand (**&**) in front of the function or procedure name in the call to **myproc** causes a pointer to the function or procedure to be passed. No argument list should follow the name when it is preceded by the ampersand.

Inside of **myproc**, the symbol that is declared as a procedure in the **local** statement is assumed to contain a pointer to a procedure. It can be called exactly like a procedure is called. It cannot be **save**'d but it can be passed on to another procedure. If it is to be passed on to another procedure, use the ampersand in the same way.

9.5 Indexing Procedures

This example assumes there are a set of procedures named **f1-f5** which are already defined. A 1x5 vector **procvec** is defined by horizontally concatenating pointers to these procedures. A new procedure, **g(**x**,**i**)** is then defined which will return the value of the i^{th} procedure evaluated at x:

```
procvec = &f1 ~ &f2 ~ &f3 ~ &f4 ~ &f5;

proc g(x,i);
    local f;
    f = procvec[i];
    local f:proc;
    retp( f(x) );
endp;
```

Notice that the **local** statement is used twice. The first time, **f** is declared to be a local matrix. After **f** has been set equal to the i^{th} pointer, **f** is declared to be a procedure and is called as a procedure in the **retp** statement.

9.6 Multiple Returns from Procedures

Procedures can return multiple items, up to 1023. The procedure is defined like this example of a complex inverse:

```
proc (2) = cminv(xr,xi); /* (2) specifies number of
                         :: return values
                         */
    local ixy, zr, zi;
    ixy = inv(xr)*xi;
    zr = inv(xr+xi*ixy); /* real part of inverse. */
    zi = -ixy*zr;        /* imaginary part of inverse. */
    retp(zr,zi);    /* return: real part, imaginary part */
endp;
```

It can then be called like this:

```
{ zr,zi } = cminv(xr,xi);
```

To make the assignment, the list of targets must be enclosed in braces.

Also, a procedure that returns more than one argument can be used as input to another procedure or function that takes more than one argument:

```
proc (2) = cminv(xr,xi);
    local ixy, zr, zi;
    ixy = inv(xr)*xi;
    zr = inv(xr+xi*ixy);  /* real part of inverse. */
    zi = -ixy*zr;         /* imaginary part of inverse. */
    retp(zr,zi);
endp;

proc (2) = cmmult(xr,xi,yr,yi);
    local zr,zi;
    zr = xr*yr-xi*yi;
    zi = xr*yi+xi*yr;
    retp(zr,zi);
endp;

{ zr,zi } = cminv( cmmult(xr,xi,yr,yi) );
```

The two returned matrices from **cmmult()** are passed directly to **cminv()** in the statement above. This is equivalent to the following statements:

```
{ tr,ti } = cmmult(xr,xi,yr,yi);
{ zr,zi } = cminv(tr,ti);
```

This is completely general so the following program is legal:

```
proc (2) = cmcplx(x);
    local r,c;
    r = rows(x);
    c = cols(x);
    retp(x,zeros(r,c));
endp;

proc (2) = cminv(xr,xi);
    local ixy, zr, zi;
    ixy = inv(xr)*xi;
    zr = inv(xr+xi*ixy);  /* real part of inverse. */
    zi = -ixy*zr;         /* imaginary part of inverse. */
    retp(zr,zi);
endp;

proc (2) = cmmult(xr,xi,yr,yi);
    local zr,zi;
    zr = xr*yr-xi*yi;
    zi = xr*yi+xi*yr;
    retp(zr,zi);
endp;

{ xr,xi } = cmcplx(rndn(3,3));
{ yr,yi } = cmcplx(rndn(3,3));

{ zr,zi } = cmmult( cminv(xr,xi),cminv(yr,yi) );
{ qr,qi } = cmmult( yr,yi,cminv(yr,yi) );

{ wr,wi } = cmmult(yr,yi,cminv(cmmult(cminv(xr,xi),yr,yi)));
```

Chapter 10

Libraries

The **GAUSS** library system allows for the creation and maintenance of modular programs. The user can create *libraries* of frequently used functions that the **GAUSS** system will automatically find and compile whenever they are referenced in a program.

10.1 Autoloader

The autoloader resolves references to procedures, keywords, matrices, and strings that are not defined in the program from which they are referenced. The autoloader automatically locates and compiles the files containing the symbol definitions that are not resolved during the compilation of the main file. The search path used by the autoloader is first the current directory, and then the paths listed in the **src_path** configuration variable in the order they appear. **src_path** can be defined in the **GAUSS** configuration file.

10.1.1 Forward References

When the compiler encounters a symbol that has not previously been defined, that is called a *forward reference*. **GAUSS** handles forward references in two ways, depending on whether they are *left-hand side* or *right-hand side* references.

Left-Hand Side

A left-hand side reference is usually a reference to a symbol on the left-hand side of the equal sign in an expression.

```
x = 5;
```

Left-hand side references, since they are assignments, are assumed to be matrices. In the statement above, **x** is assumed to be a matrix and the code is compiled accordingly. If, at execution time, the expression actually returns a string, the assignment is made and the type of the symbol **x** is forced to string.

Some commands are implicit left-hand side assignments. There is an implicit left-hand side reference to **x** in each statement below:

```
clear x;

load x;

open x = myfile;
```

Right-Hand Side

A right-hand side reference is usually a reference to a symbol on the right-hand side of the equal sign in an expression.

```
z = 6;
y = z + dog;
print y;
```

In the program above, since **dog** is not previously known to the compiler, the autoloader will search for it in the active libraries. If it is found, the file containing it will be compiled. If it is not found in a library, the autoload/autodelete state will determine how it is handled.

10.1.2 The Autoloader Search Path

If the autoloader is OFF, no forward references are allowed. Every procedure, matrix, and string referenced by your program must be defined before it is referenced. An **external** statement can be used above the first reference to a symbol, but the definition of the symbol must be in the main file or in one of the files that are **#include**'d. No global symbols are automatically deleted.

If the autoloader is ON, **GAUSS** searches for unresolved symbol references during compilation using a specific search path as outlined below. If the autoloader is OFF, an "Undefined symbol" error will result for right-hand side references to unknown symbols.

When autoload is ON, the autodelete state controls the handling of references to unknown symbols. The following search path will be followed to locate any symbols not previously defined:

Autodelete ON

1. **user** library
2. User-specified libraries.
3. **gauss** library
4. Current directory, then **src_path** for files with a `.g` extension.

Forward references are allowed and `.g` files need not be in a library. If there are symbols that cannot be found in any of the places listed above, an "Undefined symbol" error will be generated and all uninitialized variables and all procedures with global references will be deleted from the global symbol table. This autodeletion process is transparent to the user, since the symbols are automatically located by the autoloader the next time the program is run. This process results in more compile time, which may or may not be significant, depending on the speed of the computer and the size of the program.

Autodelete OFF

1. **user** library
2. User-specified libraries.
3. **gauss** library

All `.g` files must be listed in a library. Forward references to symbols that are not listed in an active library are not allowed. For example:

```
x = rndn(10,10);
y = sym(x);    /* Forward reference to sym */

proc sym(x);
    retp(x+x');
endp;
```

Use an **external** statement for anything referenced above its definition if autodelete is OFF.

```
external proc sym;

x = rndn(10,10);
y = sym(x);

proc sym(x);
    retp(x+x');
endp;
```

When autodelete is OFF, symbols not found in an active library will not be added to the symbol table. This prevents the creation of uninitialized procedures in the global symbol table. No deletion of symbols from the global symbol table will take place.

Libraries

The first place **GAUSS** looks for a symbol definition is in the *active* libraries. A **GAUSS** library is a text file that serves as a dictionary to the source files that contain the symbol definitions. When a library is active, **GAUSS** will look in it whenever it is looking for a symbol it is trying to resolve. The **library** statement is used to make a library active. Library files should be located in the subdirectory listed in the **lib_path** configuration variable. Library files have a `.lcg` extension.

Suppose you have several procedures that are all related and you would like to have them all defined in the same file. You can create such a file, and, with the help of a library, the autoloader will be able to find the procedures defined in that file whenever they are called.

First, create the file that is to contain your desired procedure definitions. By convention, we usually name these files with an `.src` extension, but you can use any name and any file extension you wish. In that file, put all the definitions of related procedures you wish to use. Here is an example of such a file. It is called `norm.src`.

```
/*
** norm.src
**
** This is a file containing the definitions of three
** procedures which return the norm of a matrix x.
** The three norms calculated are the 1-norm, the
** inf-norm and the E-norm.
*/

proc onenorm(x);
    retp(maxc(sumc(abs(x))));
endp;

proc infnorm(x);
    retp(maxc(sumc(abs(x'))));
endp;

proc Enorm(x);
    retp(sumc(sumc(x.*x)));
endp;
```

Next, create a library file that contains the name of the file you want access to, and the list of symbols defined in it. This can be done with the **lib** command. See **lib** in the *COMMAND REFERENCE*.

A library file entry has a filename that is flush left. The drive and path can be included to speed up the autoloader. Indented below the filename are the symbols included in

the file. There can be multiple symbols listed on a line with spaces between. The symbol type follows the symbol name with a colon delimiting it from the symbol name. The valid symbol types are:

fn	user-defined single line function.
keyword	keyword.
proc	procedure.
matrix	matrix, numeric or character.
string	string.

If the symbol type is missing, the colon must not be present and the symbol type is assumed to be **proc**. Both library files below are valid.

```
/*
** math
**
** This library lists files and procedures for mathematical routines.
*/

norm.src
    onenorm:proc infnorm:proc Enorm:proc
complex.src
    cmmult:proc cmdiv:proc cmadd:proc cmsoln:proc
poly.src
    polychar:proc polyroot:proc polymult:proc
```

```
/*
** math
**
** This library lists files and procedures for mathematical routines.
*/

c:\gauss\src\norm.src
    onenorm : proc
    infnorm : proc
    Enorm : proc
c:\gauss\src\complex.src
    cmmult : proc
    cmdiv : proc
    cmadd : proc
    cmsoln : proc
c:\gauss\src\fcomp.src
    feq : proc
    fne : proc
    flt : proc
    fgt : proc
    fle : proc
    fge : proc
c:\gauss\src\fcomp.dec
    _fcmptol : matrix
```

Once the autoloader finds, via the library, the file containing your procedure definition, everything in that file will be compiled. For this reason, you will probably want to combine related procedures in the same file to minimize the compiling of procedures not needed by your program. In other words, do not combine unrelated functions in one `.src` file because if one function in an `.src` file is needed, the whole file will be compiled. This does not apply to library (`.lcg`) files.

user Library

This is a library for user-created procedures. If the autoloader is ON, the **user** library is the first place **GAUSS** looks when trying to resolve symbol references.

You can update the **user** library with the **lib** command as follows:

```
lib user myfile.src
```

This will update the **user** library by adding a reference to `myfile.src`.

No **user** library is shipped with **GAUSS**. It will be created the first time you use the **lib** command to update it.

See the *COMMAND REFERENCE* for a detailed discussion of the parameters available with the **lib** command.

G Files

If autoload and autodelete are ON and a symbol is not found in a library, the autoloader will assume it is a procedure and look for a file that has the same name as the symbol and a `.g` extension. For example, if you have defined a procedure called **square**, you could put the definition in a file called `square.g` on one of the subdirectories listed in your **src_path**. If autodelete is OFF, the `.g` file needs to be listed in an active library, for example, in the **user** library. See **user Library** above.

10.2 Global Declaration Files

If your application makes use of several global variables, you will probably want to create a file containing **declare** statements. In our code, we use files with the extension `.dec` to assign default values to global matrices and strings with **declare** statements. A file with an `.ext` extension containing the same symbols in **external** statements can also be created and **#include**'d at the top of any file that references these global variables. An appropriate library file should contain the name of the `.dec` files and the names of the globals they declare. The example below demonstrates more specifically how these files work together.

Here is an example that illustrates the way in which `.dec`, `.ext`, `.lcg` and `.src` files work together. We always begin the names of global matrices or strings with '_' to distinguish them easily from procedures.

```
/*
** fcomp.src
**
**  These functions use _fcmptol to fuzz the comparison operations
**  to allow for roundoff error.
**
**  The statement:      y = feq(a,b);
**
**  is equivalent to:   y = a eq b;
**
**  Returns a scalar result, 1 (true) or 0 (false)
**
**      y = feq(a,b);
**      y = fne(a,b);
*/

#include fcomp.ext;

proc feq(a,b);
    retp(abs(a-b) <= _fcmptol);
endp;

proc fne(a,b);
    retp(abs(a-b) > _fcmptol);
endp;
```

```
/*
** fcomp.dec - global declaration file for fuzzy comparisons.
*/

declare matrix _fcmptol != 1e-14;
```

```
/*
** fcomp.ext - external declaration file for fuzzy comparisons.
*/

external matrix _fcmptol;
```

```
/*
** fcomp.lcg - fuzzy compare library
```

```
*/

fcomp.dec
    _fcmptol:matrix
fcomp.src
    feq:proc
    fne:proc
```

With the exception of the library (`.lcg`) files, these files must be located along your **src_path**. The library files must be on your **lib_path**. With these files in place, the autoloader will be able to find everything needed to run the following programs:

```
library fcomp;
x = rndn(3,3);
xi = inv(x);
xix = xi*x;
if feq(xix,eye(3));
    print "Inverse within tolerance.";
else;
    print "Inverse not within tolerance.";
endif;
```

If the default tolerance of $1e-14$ is too tight, the tolerance could be relaxed.

```
library fcomp;
x = rndn(3,3);
xi = inv(x);
xix = xi*x;
_fcmptol = 1e-12;     /* reset tolerance */
if feq(xix,eye(3));
    print "Inverse within tolerance.";
else;
    print "Inverse not within tolerance.";
endif;
```

10.3 Troubleshooting

Below is a partial list of errors you may encounter in using the library system, followed by the most probable cause of these errors.

```
(4) : error G0290 : '/gauss/lib/prt.lcg' : Library not found
```

The autoloader is looking for a library file called tt prt.lcg, because it has been activated in a **library** statement. Check the subdirectory listed in your **lib_path** configuration variable for a file called `prt.lcg`.

```
(0) : error G0292 : 'prt.dec' : File listed in library not found
```

The autoloader can't find a file called `prt.dec`. Check for this file. It should exist somewhere along your **src_path**, if you have it listed in `prt.lcg`.

```
Undefined symbols:
    PRTVEC /gauss/src/tstprt.g(2)
```

The symbol **prtvec** could not be found. Check if the file containing **prtvec** is in the **src_path**. You may have not activated the library which contains your symbol definition. Activate the library in a **library** statement.

```
/gauss/src/prt.dec(3) : Redefinition of '__vnames'
(proc)__vnames being declared external matrix
```

You are trying to illegally force a symbol to another type. You probably have a name conflict that needs to be resolved by renaming one of the symbols.

```
/gauss/lib/prt.lcg(5) : error G0301 : 'prt.dec' : Syntax error in library
Undefined symbols:
__VNAMES /gauss/src/prt.src(6)
```

Check your library to see that all filenames are flush left and that all the symbols defined in that file are indented by at least one space.

10.3.1 Using dec Files

Below is some advice you are encouraged to follow when constructing your own library system:

- Whenever possible, declare variables in a file that contains only **declare** statements. When your program is run again without clearing the workspace, the file containing the variable declarations will not be compiled and **declare** warnings will be prevented.
- Provide a function containing regular assignment statements to reinitialize the global variables in your program if they ever need to be reinitialized on the fly or between runs. Put this in a separate file from the declarations.

```
proc (0) = globset;
    _vname = "X";
    _con = 1;
    _row = 0;
    _title = "";
endp;
```

- Never declare any global in more than one file.
- Never declare a global more than once in any one file. A global should only be declared ONCE. By observing this advice, you will avoid meaningless redefinition errors and **declare** warnings. Redefinition error messages and **declare** warnings are meant to help you prevent name conflicts, and will be useless to you if your code generates them normally.

The point of the above is to make sure that any **declare** warnings and redefinition errors you get are *meaningful*. By knowing that such warnings and errors are significant, you will be able to debug your programs more efficiently.

Chapter 11

Compiler

GAUSS allows you to compile your large frequently used programs to a file that can be run over and over with no compile time. The compiled image is usually smaller than the uncompiled source. This is not a native code compiler; rather, it compiles to a form of pseudo-code. The file will have a `.gcg` extension.

The **compile** command will compile an entire program to a compiled file, or a file can be compiled directly from the **GAUSS** editor. An attempt to edit a compiled file will cause the source code to be loaded into the editor if it is available to the system. The **run** command assumes a compiled file if no extension is given and a file with a `.gcg` extension is in the **src_path**. A **saveall** command is available to save the current contents of memory in a compiled file for instant recall later. The **use** command will instantly load a compiled program or set of procedures at the beginning of an ASCII program before compiling the rest of the ASCII program file.

Since the compiled files are encoded binary files, the compiler is useful for developers who do not want to distribute their source code.

11.1 Compiling Programs

Programs can be compiled with the **compile** command or directly from the **GAUSS** editor.

11.1.1 Compiling a File

Source code program files that can be run with the **run** command can be compiled to `.gcg` files with the **compile** command.

```
compile qxy.e;
```

All procedures, global matrices and strings, and the main program segment will be saved in the compiled file. The compiled file can be run later using the **run** command. Any libraries used in the program will need to be present and active during the compile, but will not need to be present or active when the program is run. If the program uses the **loadexe** command, the `.rex` files will need to be present when the program is run and the **loadexe path** will need to be set to the correct subdirectory. This will usually be automatically handled in your startup file, but should be kept in mind if the program is run on a different computer than it was compiled on.

11.1.2 Compiling from EDIT Mode, DOS Only

Press **Alt-X**,**C** to get the **Compile Options** menu. The **compile to file** mode (among other things) can be set here. There are four **compile** modes:

OFF	the compiled program is not saved when the program is executed (default).
ONCE	the compiled program is saved when the program is executed and the compile mode is toggled to OFF.
ON	the compiled program will be saved when the program is executed.
ONLY	the program will be compiled and saved, but it will not be executed.

Press **Alt-X**,**R** to get the **Run Options** menu. From here, you can select whether to compile the program with or without line number records. A program compiled with line number records will be a little slower and 10-20% larger than it would be if compiled without line number records.

When you run the program by pressing **F2** or **Alt-X**,**X**, a compiled version of the program will be saved on the same subdirectory the source is located on.

If you use the compile ON or ONLY option, don't forget to turn it off or you will be creating lots of compiled files you don't want.

11.2 Saving the Current Workspace

The simplest way to create a compiled file containing a set of frequently used procedures is to use **saveall** and an **external** statement.

```
library pgraph;
external proc xy,logx,logy,loglog,hist;
saveall pgraph;
```

Just list the procedures you will be using in an **external** statement and follow it with a **saveall** statement. You do not need to list procedures that you do not explicitly call but are called from another procedure because the autoloader will find them automatically before the **saveall** command is executed. You also do not need to be exhaustive in listing every procedure you will be calling unless the source will not be available when the compiled file is **use**'d.

Remember that the list of active libraries is NOT saved in the compiled file so you may still need a **library** statement in a program that is **use**'ing a compiled file.

11.3 Debugging

If you are using compiled code in a development situation where debugging is important, you should compile the file with line number records. After the development is over you can recompile without line number records if the maximum possible execution speed is important. If you want to guarantee that all procedures contain line number records, put a **new** statement at the top of your program and turn line number tracking on.

Chapter 12

File I/O

Following is a partial list of the I/O commands in the **GAUSS** programming language.

close	close a file.
closeall	close all open files.
colsf	number of columns in a file.
create	create **GAUSS** data set.
dfree	space remaining on disk.
eof	test for end of file.
fcheckerr	check error status of a file.
fclearerr	check error status of a file and clear error flag.
fflush	flush a file's output buffer.
fgets	read a line of text from a file.
fgetsa	read multiple lines of text from a file.
fgetsat	read multiple lines of text from a file, discarding newlines.
fgetst	read a line of text from a file, discarding newline.
fileinfo	returns names and info of files matching a specification.
files	returns a list of files matching a specification.
filesa	returns a list of files matching a specification.
fopen	open a file.
fputs	write strings to a file.
fputst	write strings to a file, appending newlines.
fseek	reposition file pointer.
fstrerror	get explanation of last file I/O error.
ftell	get position of file pointer.
getf	load a file into a string.
getname	get variable names from data set.

iscplxf	returns whether a data set is real or complex.
load	load matrix file or small ASCII file (same as **loadm**).
loadd	load a small **GAUSS** data set into a matrix.
loadm	load matrix file or small ASCII file.
loads	load string file.
open	open a **GAUSS** data set.
output	control printing to an auxiliary output file or device.
readr	read a specified number of rows from a file.
rowsf	number of rows in file.
save	save matrices, strings, procedures.
saved	save a matrix in a **GAUSS** data set.
seekr	reset read/write pointer in a data set.
sortd	sort a data set.
typef	get type of data set (bytes per element).
writer	write data to a data set.

12.1 ASCII Files

GAUSS has facilities for reading and writing ASCII files. Most other software can also read and write ASCII files so this provides a way of sharing data between **GAUSS** and many other kinds of programs.

12.1.1 Matrix Data

Reading

Files containing numeric data that are delimited with spaces or commas and are small enough to fit into a single matrix or string can be read with **load**. Larger ASCII data files can be converted to **GAUSS** data sets with the **atog** utility program which is documented in the *UTILITIES* section of this manual. **atog** can convert packed ASCII files as well as delimited files.

For small delimited data files, the **load** statement can be used to load the data directly into a **GAUSS** matrix. The limit here is that the resulting **GAUSS** matrix must be no larger than the limit for a single matrix.

```
load x[] = dat1.asc;
```

This will load the data in the file `dat1.asc` into an Nx1 matrix **x**. This is the preferred method because **rows(x)** can be used to determine how many elements were actually loaded and the matrix can be **reshape**'d to the desired form.

```
load x[] = dat1.asc;
if rows(x) eq 500;
    x = reshape(x,100,5);
else;
    errorlog "Read Error";
    end;
endif;
```

For quick interactive loading without error checking you can use:

```
load x[100,5] = dat1.asc;
```

This will load the data into a 100x5 matrix. If there are more or fewer than 500 numbers in the data set, the matrix will be automatically reshaped to 100x5.

Writing Using Auxiliary Output

To write data to an ASCII file the **print** or **printfm** commands are used to print to the auxiliary output. The resulting files are standard ASCII files, and can be edited with **GAUSS**'s editor or another text editor.

The **output** and **outwidth** commands are used to control the auxiliary output. **print** or **printfm** commands are used to control what is sent to the output file.

The screen can be turned on and off using **screen**. When printing a large amount of data to the auxiliary output, the screen can be turned off using the command:

```
screen off;
```

This will speed things up considerably, especially if the auxiliary output is a disk file.

It is easy to forget to turn the screen on again, and you may think your computer is broken. It is therefore good practice to use the **end** statement to terminate your programs. **end** will automatically perform **screen on** and **output off**.

The following commands can be used to control printing to the auxiliary output:

format	controls format for printing a matrix.
output	open, close, rename auxiliary output file or device.
outwidth	auxiliary output width.
printfm	formatted matrix print.
print	print matrix or string.
screen	Turn printing to the screen on and off.

The following example illustrates printing a matrix to a file:

```
format /rd 8,2;
outwidth 132;
output file = myfile.asc reset;
screen off;
print x;
output off;
screen on;
```

The numbers in the matrix **x** will be printed with a field width of 8 spaces per number, and with 2 places beyond the decimal point. The resulting file will be an ASCII data file. It will have 132 column lines maximum.

A more extended example is given below. This program will write the contents of the **GAUSS** file `mydata.dat` into an ASCII file called `mydata.asc`. If there were an existing file by the name of `mydata.asc`, it would be overwritten:

```
output file = mydata.asc reset;
screen off;
format /rd 1,8;
open fp = mydata;
do until eof(fp);
    print readr(fp,200);;
endo;
fp = close(fp);
end;
```

The **output ... reset** command will create an auxiliary output file called `mydata.asc` to receive the output. The screen is turned off to speed up the process. The **GAUSS** data file `mydata.dat` is opened for read and 200 rows will be read per iteration until the end of the file is reached. The data that are read will be printed to the auxiliary output `mydata.asc` only, because the screen is off.

12.1.2 General File I/O

getf will read a file and return it in a string variable. Any kind of file can be read in this way as long as it will fit into a single string variable.

To read files sequentially, use **fopen** to open the file and use **fgets**, **fputs** and associated functions to read and write the file. The current position in a file can be determined with **ftell**. The procedure below uses these functions to copy an ASCII text file.

```
proc copy(src, dest);
    local fin, fout, str;
```

```
    fin = fopen(src, "rb");
    if not fin;
        retp(1);
    endif;

    fout = fopen(dest, "wb");
    if not fin;
        call close(fin);
        retp(2);
    endif;

    do until eof(fin);
        str = fgets(fin, 1024);
        if fputs(fout, str) /= 1;
            call close(fin);
            call close(fout);
            retp(3);
        endif;
    endo;

    call close(fin);
    call close(fout);
    retp(0);
endp;
```

12.2 Data Sets

GAUSS data sets are the preferred method of storing data for use within **GAUSS**. Use of these data sets allows extremely fast reading and writing of data. Many library functions are designed to read data from these data sets.

12.2.1 Layout

GAUSS data sets are arranged as matrices. That is, they are organized in terms of rows and columns. The columns in a data file are assigned names, and these names are stored in the header, or in the case of the v89 format, in a separate header file.

The limit on the number of rows in a **GAUSS** data set is determined by disk size. The limit on the number of columns is listed in your supplement. Data can be stored in 2, 4, or 8 bytes per number, rather than just 8 bytes as in the case of **GAUSS** matrix files.

The ranges of the different formats is as follows:

Bytes	*Type*	*Significant Digits*	*Range*
2	*integer*	4	$-32768 <= X <= 32767$
4	*single*	$6-7$	$8.43E-37 <= \|X\| <= 3.37E+38$
8	*double*	$15-16$	$4.19E-307 <= \|X\| <= 1.67E+308$

12.2.2 Creating Data Sets

Data sets can be created with the **create** command. The names of the columns, the type of data, etc., can be specified. See **create** in the *COMMAND REFERENCE* for complete details.

Data sets, unlike matrices, cannot change from real to complex or vice-versa. Data sets are always stored a row at a time. The rows of a complex data set, then, have the real and imaginary parts interleaved, element by element. For this reason, you cannot write rows from a complex matrix to a real data set—there is no way to interleave the data without rewriting the whole data set. If you need to do so, explicitly convert the rows of data first using the **real** and **imag** functions (see the *COMMAND REFERENCE*), then write them to the data set. Rows from a real matrix CAN be written to a complex data set; **GAUSS** simply supplies 0's for the imaginary part.

To create a complex data set, include the **complex** flag in your **create** command.

12.2.3 Reading and Writing

The basic functions in **GAUSS** for reading data files are **open** and **readr**:

```
open f1 = dat1;
x = readr(f1,100);
```

The **readr** function here will read in 100 rows from `dat1.dat`. The data will be assigned to a matrix **x**.

loadd and **saved** can be used for loading and saving small data sets.

The following code illustrates the creation of a **GAUSS** data file by merging (horizontally concatenating) two existing data sets:

```
file1 = "dat1";
file2 = "dat2";
```

```
outfile = "daty";
open fin1 = ^file1 for read;
open fin2 = ^file2 for read;
varnames = getname(file1)|getname(file2);
otyp = maxc(typef(fin1)|typef(fin2));
create fout = ^outfile with ^varnames,0,otyp;
nr = 400;
do until eof(fin1) or eof(fin2);
     y1 = readr(fin1,nr);
     y2 = readr(fin2,nr);
     r = maxc(rows(y1)|rows(y2));
     y = y1[1:r,.] ~ y2[1:r,.];
     call writer(fout,y);
endo;
closeall fin1,fin2,fout;
```

In this example, data sets `dat1.dat` and `dat2.dat` are opened for reading. The variable names from each data set are read using **getname** and combined in a single vector called **varnames**. A variable called **otyp** is created which will be equal to the larger of the two data types of the input files. This will insure that the output is not rounded to less precision than the input files. A new data set `daty.dat` is created using the **create ... with ...** command. Then, on every iteration of the loop, 400 rows are read in from each of the two input data sets, horizontally concatenated, and written out to `daty.dat`. When the end of one of the input files is reached, reading and writing will stop. The **closeall** command is used to close all the files.

12.2.4 Distinguishing Character and Numeric Data

Although **GAUSS** itself does not distinguish between numeric and character columns in a matrix or data set, some of the **GAUSS** Application programs do, so it is important when creating a data set to indicate the type of data in the various columns. The following discusses two ways of doing this.

Using Type Vectors

The **v89** data set format distinguished between character and numeric data in data sets by the case of the variable names associated with the columns. The **v96** data set format, however, stores this type information separately, resulting in a much cleaner and more robust method of tracking variable types, and greater freedom in the naming of data set variables.

When you create a data set, you can supply a vector indicating the type of data in each column of the data set. For example:

```
data = { M 32  21500,
         F 27  36000,
         F 28  19500,
         M 25  32000 };
vnames = { "Sex" "Age" "Pay" };
vtypes = { 0 1 1 };
create f = mydata with ^vnames, 3, 8, vtypes;
call writer(f,data);
f = close(f);
```

To retrieve the type vector you use **vartypef**.

```
open f = mydata for read;
vn = getnamef(f);
vt = vartypef(f);
print vn';
print vt';

  Sex   Age   Pay
    0     1     1
```

Note that the function **getnamef** in this example returns a string array rather than a character vector, so you can print it without the "**$**" prefix.

Using the Uppercase/Lowercase Convention

The following method for distinguishing character/numeric data is in the process of being obsoleted; you should use the approach described above.

To distinguish numeric variables from character variables in **GAUSS** data sets, some **GAUSS** Application programs recognize an "uppercase/lowercase" convention: if the variable name is uppercase, the variable is assumed to be numeric, and if it is lowercase, the variable is assumed to be character. The **atog** utility program implements this convention when you use the # and $ operators to toggle between character and numeric variable names listed in the **invar** statement and you have specified **nopreservecase**.

It should be remembered that **GAUSS** does not make this distinction internally and it is up to the program to keep track of and make use of the information recorded in the case of the variable names in a data set.

When creating a data set using the **saved** command, this convention can be established as follows:

```
data = { M 32  21500,
         F 27  36000,
         F 28  19500,
         M 25  32000 };
dataset = "mydata";
vnames = { "sex" AGE PAY };
call saved(data,dataset,vnames);
```

It is necessary to put "sex" into quotes in order to prevent it from being forced to uppercase.

The procedure **getname** can be used to retrieve the variable names:

```
print $getname("mydata");
```

The names are:

```
sex
AGE
PAY
```

When you are writing or creating a data set (as the above example using **saved** does) the case of the variable names is important. This is especially true if the **GAUSS** Applications programs will be used on the data set.

12.3 Matrix Files

GAUSS matrix files are files created by the **save** command.

The **save** command takes a matrix in memory, adds a header that contains information on the number of rows and columns in the matrix, and stores it on disk. Numbers are stored in double precision just as they are in matrices in memory. These files have the extension `.fmt`.

Matrix files can be no larger than a single matrix. No variable names are associated with matrix files.

GAUSS matrix files can be **load**'ed into memory using the **load** or **loadm** commands or they can be opened with the **open** command and read with the **readr** command. With the **readr** command a subset of the rows can be read. With the **load** command, the entire matrix is **load**'ed.

GAUSS matrix files can be **open**'ed **for read**, but not **for append**, or **for update**.

If a matrix file has been opened and assigned a file handle, **rowsf** and **colsf** can be used to determine how many rows and columns it has without actually reading it into memory. **seekr** and **readr** can be used to jump to particular rows and to read them into memory. This can be useful when only a subset of rows is needed at any time. This procedure will save memory and can be much faster than **load**'ing the entire matrix into memory.

12.4 File Formats

This section discusses the **GAUSS** binary file formats.

There are 4 currently supported matrix file formats:

Version	Extension	Support
Small Matrix v89	`.fmt`	Obsolete, read on Intel x86 processors only
Extended Matrix v89	`.fmt`	Obsolete, read on Intel only
Matrix v92	`.fmt`	Obsolete, read on Unix only
Universal Matrix v96	`.fmt`	Supported for read/write

There are 4 currently supported string file formats:

Version	Extension	Support
Small String v89	`.fst`	Obsolete, read on Intel x86 processors only
Extended String v89	`.fst`	Obsolete, read on Intel only
String v92	`.fst`	Obsolete, read on Unix only
Universal String v96	`.fst`	Supported for read/write

There are 4 currently supported data set formats:

Version	Extension	Support
Small Data Set v89	`.dat`, `.dht`	Obsolete, read on Intel x86 processors only
Extended Data Set v89	`.dat`, `.dht`	Obsolete, read on Intel only
Data Set v92	`.dat`	Obsolete, read on Unix only
Universal Data Set v96	`.dat`	Supported for read/write

12.4.1 Small Matrix v89

Matrix files are binary files, and cannot be read with a text editor. They are created with **save**. Matrix files have a `.fmt` extension and a 16-byte header formatted as follows:

offset	description
0-1	DDDD hex, identification flag
2-3	rows, unsigned 2-byte integer
4-5	columns, unsigned 2-byte integer
6-7	size of file minus 16-byte header, unsigned 2-byte integer
8-9	type of file, 0086 hex for real matrices, 8086 hex for complex matrices
10-15	reserved, all 0's

The body of the file starts at offset 16 and consists of IEEE format double precision floating point numbers or character elements of up to 8 characters. Character elements take up 8 bytes and are padded on the right with zeros. The size of the body of the file is 8*rows*cols rounded up to the next 16-byte paragraph boundary. Numbers are stored row by row. A 2x3 real matrix will be stored on disk in the following way from the lowest addressed element to the highest addressed element:

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

For complex matrices, the size of the body of the file is 16*rows*cols. The entire real part of the matrix is stored first, then the entire imaginary part. A 2x3 complex matrix will be stored on disk in the following way from the lowest addressed element to the highest addressed element:

(*real part*)	$[1,1]$	$[1,2]$	$[1,3]$	$[2,1]$	$[2,2]$	$[2,3]$
(*imaginary part*)	$[1,1]$	$[1,2]$	$[1,3]$	$[2,1]$	$[2,2]$	$[2,3]$

12.4.2 Extended Matrix v89

In older 32-bit versions, matrices with more than 8190 elements were saved in a format that cannot be read by the 16-bit version. These extended files have a 16-byte header formatted as follows:

offset	description
0-1	EEDD hex, identification flag
2-3	type of file, 0086 hex for real matrices, 8086 hex for complex matrices
4-7	rows, unsigned 4-byte integer
8-11	columns, unsigned 4-byte integer
12-15	size of file minus 16-byte header, unsigned 4-byte integer

The size of the body of an extended matrix file is 8*rows*cols (not rounded up to a paragraph boundary). Aside from this, the differences in the file header, and the number of elements allowed in the file, extended matrix files conform to the matrix file description specified above.

12.4.3 Small String v89

String files are created with **save**. String files have a 16-byte header formatted as follows:

offset	description
0-1	DFDF hex, identification flag
2-3	1, unsigned 2-byte integer
4-5	length of string plus null byte, unsigned 2-byte integer
6-7	size of file minus 16-byte header, unsigned 2-byte integer
8-9	001D hex, type of file
10-15	reserved, all 0's

The body of the file starts at offset 16. It consists of the string terminated with a null byte. The size of the file is the 16-byte header plus the length of the string and null byte rounded up to the next 16-byte paragraph boundary.

12.4.4 Extended String v89

In the 32-bit version, strings with more than 65519 characters are saved in a format that cannot be read by the 16-bit version. These extended files have a 16-byte header formatted as follows:

offset	description
0-1	EEDF hex, identification flag
2-3	001D hex, type of file
4-7	1, unsigned 4-byte integer
8-11	length of string plus null byte, unsigned 4-byte integer
12-15	size of file minus 16-byte header, unsigned 4-byte integer

The body of the file starts at offset 16. It consists of the string terminated with a null byte. The size of the file is the 16-byte header plus the length of the string and null byte rounded up to the next 8-byte boundary.

12.4.5 Small Data Set v89

The `.dat` file is used with the `.dht` descriptor file. The format of the `.dat` file is binary data. It will be in one of three types:

8-byte IEEE floating point

4-byte IEEE floating point

2-byte signed binary integer, twos complement

Numbers are stored row by row.

The `.dht` file is used in conjunction with the `.dat` file as a descriptor file and as a place to store names for the columns in the `.dat` file. Its format is as follows:

offset	description
0-1	DADA hex, identification flag
2-5	reserved, all 0's
6-7	columns, unsigned 2-byte integer
8-9	row size in bytes, unsigned 2-byte integer
10-11	header size in bytes, unsigned 2-byte integer
12-13	data type in `.dat` file (2 4 8), unsigned 2-byte integer
14-17	reserved, all 0's
18-21	reserved, all 0's
22-23	control flags, unsigned 2-byte integer
24-127	reserved, all 0's

Column names begin at offset 128 and are stored 8 bytes each in ASCII format. Names with less than 8 characters are padded on the right with bytes of 0.

The number of rows in the `.dat` file is calculated in **GAUSS** using the file size, columns, and data type. This means that users can modify the `.dat` file by adding or deleting rows with other software without updating the header information.

Names for the columns should be lowercase for character data if you want to be able to distinguish them from numeric data with **vartype**.

GAUSS currently examines only the 4's bit of the control flags. This bit is set to 0 for real data sets, 1 for complex data sets. All other bits are 0.

Data sets are always stored a row at a time. A real data set with 2 rows and 3 columns (not an overlarge data set!) will be stored on disk in the following way from the lowest addressed element to the highest addressed element:

$$\begin{matrix}[1,1] & [1,2] & [1,3] \\ [2,1] & [2,2] & [2,3]\end{matrix}$$

The rows of a complex data set are stored with the real and imaginary parts interleaved, element by element. A 2x3 complex data set, then, will be stored on disk in the following way from the lowest addressed element to the highest addressed element:

$$\begin{matrix}[1,1]r & [1,1]i & [1,2]r & [1,2]i & [1,3]r & [1,3]i \\ [2,1]r & [2,1]i & [2,2]r & [2,2]i & [2,3]r & [2,3]i\end{matrix}$$

12.4.6 Extended Data Set v89

In the 32-bit version, data sets with more than 8175 columns are saved in a format that cannot be read by the 16-bit version. These extended files have a `.dht` descriptor file formatted as follows:

offset	description
0-1	EEDA hex, identification flag
2-3	data type in `.dat` file (2 4 8), unsigned 2-byte integer
4-7	reserved, all 0's
8-11	columns, unsigned 4-byte integer
12-15	row size in bytes, unsigned 4-byte integer
16-19	header size in bytes, unsigned 4-byte integer
20-23	reserved, all 0's
24-27	reserved, all 0's
28-29	control flags, unsigned 2-byte integer
30-127	reserved, all 0's

Aside from the differences in the descriptor file and the number of columns allowed in the data file, extended data sets conform to the v89 data set description specified above.

12.4.7 Matrix v92

offset	description
0-3	always 0
4-7	always 0xEECDCDCD
8-11	reserved
12-15	reserved
16-19	reserved
20-23	0 - real matrix, 1 - complex matrix
24-27	number of dimensions 0 - scalar 1 - row vector 2 - column vector, matrix
28-31	header size, 128 + number of dimensions * 4, padded to 8-byte boundary
32-127	reserved

If the data is a scalar, the data will directly follow the header.

If the data is a row vector, an unsigned integer equaling the number of columns in the vector will precede the data, along with 4 padding bytes.

If the data is a column vector or a matrix, there will be two unsigned integers preceding the data. The first will represent the number of rows in the matrix and the second will represent the number of columns.

The data area always begins on an even 8-byte boundary. Numbers are stored in double precision (8 bytes per element, 16 if complex). For complex matrices, all of the real parts are stored first, followed by all imaginary parts.

12.4.8 String v92

offset	description
0-3	always 0
4-7	always 0xEECFCFCF
8-11	reserved
12-15	reserved
16-19	reserved
20-23	size of string in units of eight bytes
24-27	length of string plus null terminator in bytes
28-127	reserved

The size of the data area is always divisible by 8, and is padded with nulls if the length of the string is not evenly divisible by 8. If the length of the string is evenly divisible by 8 the data area will be the length of the string plus 8. The data area follows immediately after the 128-byte header.

12.4.9 Data Set v92

offset	description
0-3	always 0
4-7	always 0xEECACACA
8-11	reserved
12-15	reserved
16-19	reserved
20-23	rows in data set
24-27	columns in data set
28-31	0 for real data set, 1 for complex data set
32-35	type of data in data set, 2, 4, or 8
36-39	header size in bytes is 128 + columns * 9
40-127	reserved

The variable names begin at offset 128 and are stored 8 bytes each in ASCII format. Each name corresponds to one column of data. Names less than 8 characters are padded on the right with bytes of zero.

The variable type flags immediately follow the variable names. They are 1-byte binary integers, one per column, padded to an even 8-byte boundary. A 1 indicates a numeric variable and a 0 indicates a character variable.

The contents of the data set follow the header and start on an 8-byte boundary. Data is either 2-byte signed integer, 4-byte single precision floating point or 8-byte double precision floating point.

12.4.10 Matrix v96

offset	description
0-3	always 0xFFFFFFFF
4-7	always 0
8-11	always 0xFFFFFFFF
12-15	always 0
16-19	always 0xFFFFFFFF
20-23	0xFFFFFFFF for forward byte order, 0 for backward byte order
24-27	0xFFFFFFFF for forward bit order, 0 for backward bit order
28-31	always 0xABCDEF01
32-35	currently 1
36-39	reserved
40-43	floating point type, 1 for IEEE 754
44-47	1008 (double precision data)
48-51	8, the size in bytes of a double matrix
52-55	0 for real matrix, 1 for complex matrix
56-59	1 - imaginary part of matrix follows real part (standard GAUSS style) 2 - imaginary part of each element immediately follows real part (FORTRAN style)
60-63	number of dimensions 0 scalar 1 row vector 2 column vector or matrix
64-67	1 for row major ordering of elements, 2 for column major
68-71	always 0
72-75	header size, 128 + dimensions * 4, padded to 8-byte boundary
76-127	reserved

If the data is a scalar, the data will directly follow the header.

If the data is a row vector, an unsigned integer equaling the number of columns in the vector will precede the data, along with 4 padding bytes.

If the data is a column vector or a matrix, there will be two unsigned integers preceding the data. The first will represent the number of rows in the matrix and the second will represent the number of columns.

The data area always begins on an even 8-byte boundary. Numbers are stored in double precision (8 bytes per element, 16 if complex). For complex matrices, all of the real parts are stored first, followed by all the imaginary parts.

12.4.11 Data Set v96

offset	description
0-3	always 0xFFFFFFFF
4-7	always 0
8-11	always 0xFFFFFFFF
12-15	always 0
16-19	always 0xFFFFFFFF
20-23	0xFFFFFFFF for forward byte order, 0 for backward byte order
24-27	0xFFFFFFFF for forward bit order, 0 for backward bit order
28-31	0xABCDEF02
32-35	version, currently 1
36-39	reserved
40-43	floating point type, 1 for IEEE 754
44-47	12 for signed 2-byte integer 1004 for single precision floating point 1008 for double precision float
48-51	2, 4, or 8, the size of an element in bytes
52-55	0 for real matrix, 1 for complex matrix
56-59	1 - imaginary part of matrix follows real part (standard GAUSS style) 2 - imaginary part of each element immediately follows real part (FORTRAN style)
60-63	always 2
64-67	1 for row major ordering of elements, 2 for column major
68-71	always 0
72-75	header size, 128 + columns * 33, padded to 8-byte boundary
76-79	reserved
80-83	rows in data set
84-87	columns in data set
88-127	reserved

The variable names begin at offset 128 and are stored 32 bytes each in ASCII format. Each name corresponds to one column of data. Names less than 32 characters are padded on the right with bytes of zero.

The variable type flags immediately follow the variable names. They are 1-byte binary integers, one per column, padded to an even 8-byte boundary. A 1 indicates a numeric variable and a 0 indicates a character variable.

Contents of the data set follow the header and start on an 8-byte boundary. Data is either 2-byte signed integer, 4-byte single precision floating point or 8-byte double precision floating point.

Chapter 13

Data Transformations

GAUSS allows expressions that directly reference variables (columns) of a data set. This is done within the context of a data loop.

```
dataloop infile outfile;
    drop wagefac wqlec shordelt foobly;
    csed = ln(sqrt(csed));
    select csed > 0.35 and married $== "y";
    make chfac = hcfac + wcfac;
    keep csed chfac stid recsum voom;
endata;
```

GAUSS translates the data loop into a procedure that does the required operations and then calls the procedure automatically at the location (in your program) of the data loop. It does this by translating your main program file into a temporary file and then executing the temporary file.

A data loop may be placed only in the main program file. Data loops in files that are **#include**'d or autoloaded are not recognized.

13.1 Data Loop Statements

A data loop begins with a **dataloop** statement and ends with an **endata** statement. Inside of a data loop, the following statements are supported:

code	create variable based on a set of logical expressions.
delete	delete rows (observations) based on a logical expression.
drop	specify variables NOT to be written to data set.
extern	allows access to matrices and strings in memory.
keep	specify variables to be written to output data set.
lag	lag variables a number of periods.
listwise	controls deletion of missing values.
make	create new variable.
outtyp	specify output file precision.
recode	change variable based on a set of logical expressions.
select	select rows (observations) based on a logical expression.
vector	create new variable from a scalar returning expression.

In any expression inside of a data loop, all text symbols not immediately followed by a left parenthesis '**(**' are assumed to be data set variable (column) names. Text symbols followed by a left parenthesis are assumed to be procedure names. Any symbol listed in an **extern** statement is assumed to be a matrix or string already in memory.

13.2 Using Other Statements

All program statements in the main file and not inside of a data loop are passed through to the temporary file without modification. Program statements within a data loop that are preceded by a '**#**' are passed through to the temporary file without modification. The adept user who is familiar with the code generated in the temporary file can use this to do out-of-the-ordinary operations inside the data loop.

13.3 Debugging Data Loops

The translator that processes data loops can be turned on and off. When the translator is on, there are three distinct phases in running a program.

Translation	translation of main program file to temporary file.
Compilation	compilation of temporary file.
Execution	execution of compiled code.

13.3.1 Translation Phase

In the translation phase, the main program file is translated into a temporary file. Each data loop is translated into a procedure and a call to this procedure is placed in the

temporary file at the same location as the original data loop. The data loop itself is commented out in the temporary file. All the data loop procedures are placed at the end of the temporary file.

Depending upon the status of line number tracking, error messages encountered in this phase will be printed with the file name and line numbers corresponding to the main file.

13.3.2 Compilation Phase

In the compilation phase, the temporary file **$xrun$.tmp** is compiled. Depending upon the status of line number tracking, error messages encountered in this phase will be printed with the file name and line numbers corresponding to both the main file and the temporary file.

13.3.3 Execution Phase

In the execution phase, the compiled program is executed and error messages will include line number references from both the original file and the temporary file, depending upon the status of line number tracking.

13.4 Reserved Variables

The following local variables are created by the translator and used in the produced code:

x_cv	**x_iptr**	**x_ncol**	**x_plag**
x_drop	**x_keep**	**x_nlag**	**x_ptrim**
x_fpin	**x_lval**	**x_nrow**	**x_shft**
x_fpout	**x_lvar**	**x_ntrim**	**x_tname**
x_i	**x_n**	**x_out**	**x_vname**
x_in	**x_name**	**x_outtyp**	**x_x**

These variables are reserved, and should not be used within a **dataloop**... **endata** section.

Chapter 14

Data Loop Reference

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

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.

Purpose

Removes specific rows in a data loop based on a logical expression.

Format

delete *logical expression*;

Remarks

Deletes only those rows for which *logical expression* is *TRUE*. Any variables referenced must already exist, either as elements of the source data set, as externs, or as the result of a previous **make**, **vector**, or **code** statement.

GAUSS expects *logical expression* to return a row vector of 1's and 0's. The relational and other operators (e.g. $<$) 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

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

Purpose

Allows access to matrices or strings in memory from inside a data loop.

Format

extern *variable_list*;

Remarks

Commas in *variable_list* are optional.

extern tells the translator not to generate local code for the listed variables, and not to assume they are elements of the input data set.

extern statements should be placed before any reference to the symbols listed. The specified names should not exist in the input data set, or be used in a **make** statement.

Example

This example shows how to assign the contents of an external vector to a new variable in the data set, by iteratively assigning a range of elements to the variable. The reserved variable **x_x** contains the data read from the input data set on each iteration. The external vector must have at least as many rows as the data set.

```
base = 1;       /* used to index a range of elements from exvec */
dataloop oldata newdata;
    extern base, exvec;
    make ndvar = exvec[seqa(base,1,rows(x_x))];
    # base = base + rows(x_x);  /* execute command literally */
endata;
```

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

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, `_lagtmp_.dat`, on the default directory. This temporary data set becomes the input data set for the dataloop, and is then automatically deleted.

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**.

Purpose

Specifies the creation of a new variable within a data loop.

Format

make [[#]] *numvar* = *numeric_expression*;

make $ *charvar* = *character_expression*;

Remarks

A *numeric_expression* is any valid expression returning a numeric vector. A *character_expression* is any valid expression returning a character vector. If neither '**$**' nor '**#**' is specified, '**#**' is assumed.

The expression may contain explicit variable names and/or **GAUSS** commands. Any variables referenced must already exist, either as elements of the source data set, as externs, or as the result of a previous **make**, **vector**, or **code** statement. The variable name must be unique. A variable cannot be made more than once, or an error is generated.

Example

```
make sqvpt =  sqrt(velocity * pressure * temp);
make $ sex = lower(sex);
```

See also

vector

Purpose

Specifies the precision of the output data set.

Format

outtyp *num_constant*;

Remarks

num_constant must be 2, 4, or 8, to specify integer, single precision, or double precision, respectively.

If **outtyp** is not specified, the precison of the output data set will be that of the input data set. If character data is present in the data set the precision will be coerced to double.

Example

```
outtyp 8;
```

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 externs, 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

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 externs, or as the result of a previous **make**, **vector**, or **code** statement.

Example

```
select age > 40 AND sex $== 'MALE';
```

See also

delete

Purpose

Specifies the creation of a new variable within a data loop.

Format

vector [[#]] *numvar* = *numeric_expression*;

vector $ *charvar* = *character_expression*;

Remarks

A *numeric_expression* is any valid expression returning a numeric value. A *character_expression* is any valid expression returning a character value. If neither '**$**' nor '**#**' is specified, '**#**' is assumed.

vector is used in place of **make** when the expression returns a scalar rather than a vector. **vector** forces the result of such an expression to a vector of the correct length. **vector** could actually be used anywhere that **make** is used, but would generate slower code for expressions that already return vectors.

Any variables referenced must already exist, either as elements of the source data set, as externs, or as the result of a previous **make**, **vector**, or **code** statement.

Example

```
vector const =  1;
```

See also

make

Chapter 15

Graphics Categorical Reference

This chapter summarizes all of the procedures and global variables available within the Publication Quality Graphics System. A general usage description and command reference may be found in Chapters 16 and 17 respectively.

15.1 Graph Types

xy	Graph X,Y using Cartesian coordinate system.
logx	Graph X,Y using logarithmic X axis.
logy	Graph X,Y using logarithmic Y axis.
loglog	Graph X,Y using logarithmic X and Y axes.
bar	Generate bar graph.
hist	Compute and graph frequency histogram.
histf	Graph a histogram given a vector of frequency counts.
histp	Graph a percent frequency histogram of a vector.
box	Graph data using the box graph percentile method.
xyz	Graph X, Y, Z using 3-D Cartesian coordinate system.
surface	Graph a 3-D surface.
contour	Graph contour data.
draw	Supply additional graphic elements to graphs.

15.2 Axes Control and Scaling

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

15.3 Text, Labels, Titles, and Fonts

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

15.4 Main Curve Lines and Symbols

_pboxctl	Control box plotter.
_pboxlim	Output percentile matrix from box plotter.

_plctrl	Control main curve and frequency of data symbols.
_pcolor	Control line color for main curves.
_pltype	Control line style for main curves.
_plwidth	Control line thickness for main curves.
_pstype	Control symbol type for main curves.
_psymsiz	Control symbol size for main curves.
_pzclr	Z level color control for **contour** and **surface**.

15.5 Extra Lines and Symbols

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

15.6 Window, Page, and Plot Control

_pagesiz	Control size of graph for printer output.
_pageshf	Shift the graph for printer output.
_plotsiz	Control plot area size.
_plotshf	Control plot area position.
_protate	Rotate the graph 90 degrees.
margin	Control graph margins.
axmargin	Control axes margins and plot size.
begwind	Window initialization procedure.
endwind	End window manipulation, display graphs.
window	Create tiled windows of equal size.
makewind	Create window with specified size and position.
setwind	Set to specified window number.
nextwind	Set to next available window number.
getwind	Get current window number.
savewind	Save window configuration to a file.
loadwind	Load a window configuration from a file.

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

15.7 Output Options

_pnotify	Control interactive mode and progress reports.
_pscreen	Control graphics output to screen.
_psilent	Control final beep.
_pzoom	Specify zoom parameters.
_ptek	Control creation and name of graphics .tkf file.
graphprt	Generate print, conversion file.

15.8 Miscellaneous

_pbox	Draw a border around window/screen.
_pframe	Draw a frame around 2-D, 3-D plots.
_pcrop	Control cropping of graphics data outside axes area.
_pmcolor	Control colors to be used for axes, title, x. and y labels, date, box and background.
_pxmem	Memory control for graphics `.rex` files.
view	Set 3-D observer position in workbox units.
viewxyz	Set 3-D observer position in plot coordinates.
volume	Set length, width and height ratios of 3-D workbox.
rerun	Display most recently created graph.
graphset	Reset all PQG globals to default values.

Chapter 16

Publication Quality Graphics

GAUSS Publication Quality Graphics is a set of routines built on the graphics functions in **GraphiC** by Scientific Endeavors Corp.

The main graphics routines include xy, xyz, surface, polar and log plots, as well as histograms, bar, and box graphs. Users can enhance their graphs by adding legends, changing fonts, and adding extra lines, arrows, symbols and messages.

The user can create a single full size graph, inset a smaller graph into a larger one, tile a screen with several equally sized graphs or place several overlapping graphs on the screen. Window size and location are all completely under the user's control.

16.1 Configuration

Before using the Publication Quality Graphics system, you may need to configure the system for the hardware. See the graphics configuration section in the platform supplement for your platform.

16.2 General Design

GAUSS Publication Quality Graphics consists of a set of main graphing procedures and several additional procedures and global variables for customizing the output.

bar	Bar graphs.
box	Box plots.
contour	Contour plots.
draw	Draws graphs using only global variables.
hist	Histogram.
histp	Percentage histogram.
histf	Histogram from a vector of frequencies.
loglog	Log scaling on both axes.
logx	Log scaling on X axis.
logy	Log scaling on Y axis.
polar	Polar plots.
surface	3-D surface with hidden line removal.
xy	Cartesian graph.
xyz	3-D Cartesian graph.

All of the actual output to the screen happens during the call to these main routines.

16.3 Using Publication Quality Graphics

16.3.1 Getting Started Right Away

There are four basic parts to a graphics program. These elements should be in any program that uses graphics routines. The four parts are header, data setup, graphics format setup, and graphics call.

1. **Header**

   In order to use the graphics procedures, the pgraph library must be active. This is done in the **library** statement at the top of your program or command file. The next line in your program will typically be a command to reset the graphics global variables to the default state. The top two lines of your graphics program should look something like this:

   ```
   library mylib, pgraph;
   graphset;
   ```

2. **Data Setup**

   The data to be graphed must be in matrices.

   ```
   x = seqa(1,1,50);
   y = sin(x);
   ```

3. **Graphics Format Setup**

   Most of the graphics elements contain defaults which allow the user to generate a plot without modification. These defaults, however, may be overridden by the

user through the use of global variables and graphics procedures. Some of the elements custom configurable by the user are axes numbering, labeling, cropping, scaling, line and symbol sizes and types, legends and colors.

4. **Calling Graphics Routines**

   It is in the main graphics routines where all of the work for the graphics functions get done. The graphics routines take as input the user data and global variables which have previously been set up. It is in these routines where the graphics file is created and displayed.

Here are three Publication Quality Graphics examples. The first two programs are different versions of the same graph. The variables that begin with "**_p**" are the global control variables used by the graphics routines. See Section 16.8 for a detailed description of the global control variables.

If using UNIX, a graphics window will have to be opened with the **WinOpenPQG** function, see the *COMMAND REFERENCE* section in the UNIX supplement.

Example 1:

The routine being called here is a simple XY plot. The entire screen will be used. Four sets of data will be plotted with the line and symbol attributes automatically selected. This graph will include a legend, title, and a time/date stamp (time stamp is on by default).

```
library pgraph;                 /* activate PGRAPH library       */
graphset;                       /* reset global variables        */
x = seqa(.1,.1,100);            /* generate data                 */
y = sin(x);
y = y ~ y*.8 ~ y*.6 ~ y*.4;     /* 4 curves plotted against x    */
_plegctl = 1;                   /* legend on                     */
title("Example xy Graph");      /* Main title    */
xy(x,y);                        /* Call to main routine  */
```

Example 2:

Here is the same graph with more of the graphics format controlled by the user. The first two data sets will be plotted using symbols at graph points only (observed data), and the data in the second two sets will be connected with lines (predicted results).

```
library pgraph;                 /* activate PGRAPH library       */
graphset;                       /* reset global variables        */
x = seqa(.1,.1,100);            /* generate data                 */
y = sin(x);
y = y ~ y*.8 ~ y*.6 ~ y*.4;     /* 4 curves plotted against x    */
_pdate = "";                    /* date is not printed           */
```

```
_plctrl = { 1, 1, 0, 0 };       /* 2 curves w/symbols, 2 without */
_pltype = { 1, 2, 6, 6 };       /* dashed, dotted, solid lines   */
_pstype = { 1, 2, 0, 0 };       /* symbol types circles, squares */
_plegctl= { 2, 3, 1.7, 4.5 };   /* legend size and locations     */
_plegstr= "Sin wave 1.\0"\      /* 4 lines legend text           */
          "Sin wave .8\0"\
          "Sin wave .6\0"\
          "Sin wave .4";
ylabel("Amplitude");            /* Y axis label  */
xlabel("X Axis");               /* X axis label  */
title("Example xy Graph");      /* main title    */
xy(x,y);                        /* call to main routine  */
```

Example 3:

In this example, two graphics windows are drawn. The first window is a full-sized surface representation, and the second is a half-sized inset containing a contour of the same data located in the lower left corner of the screen.

```
library pgraph;                 /* activate pgraph library */

/* Generate data for surface and contour plots */
x = seqa(-10,0.1,71)';          /* note x is a row vector    */
y = seqa(-10,0.1,71);           /* note y is a column vector */
z = cos(5*sin(x) - y);          /* z is a 71x71 matrix       */

begwind;                        /* initialize graphics windows  */
makewind(9,6.855,0,0,0);        /* first window full size       */
makewind(9/2,6.855/2,1,1,0);    /* second window inset to first */

setwind(1);                     /* activate first window */
  graphset;                          /* reset global variables */
  _pzclr = { 1, 2, 3, 4 };           /* set Z level colors     */
  title("cos(5*sin(x) - y)");        /* set main title         */
  xlabel("X Axis");                  /* set X axis label       */
  ylabel("Y Axis");                  /* set Y axis label       */
  scale3d(miss(0,0),miss(0,0),-5|5); /* scale Z axis           */
  surface(x,y,z);                    /* call surface routine   */

nextwind;                       /* activate second  window. */
  graphset;                          /* reset global variables */
  _pzclr = { 1, 2, 3, 4 };           /* set Z level colors     */
  _pbox = 15;                        /* white border           */
  contour(x,y,z);                    /* call contour routine   */

endwind;                  /* Display windows */
```

While the structure has changed somewhat, the four basic elements of the graphics program are all here. The additional routines **begwind**, **endwind**, **makewind**, **nextwind**, and **setwind** are all used to control the graphics windows.

As Example 3 illustrates, the code between window functions (i.e. **setwind** or **nextwind**) may include assignments to global variables, a call to **graphset**, or may set up new data to be passed to the main graphics routines.

At this point, you may want to look through Chapter 15 to get an overview of the capabilities of the Publication Quality Graphics, or proceed directly to Section 16.8 and Chapter 17 for in-depth information on the Publication Quality Graphics functions and global control variables. For more information on how to create graphics windows, continue reading the section below.

You are encouraged to run the example programs supplied with **GAUSS**. Analyzing these programs is perhaps the best way to learn how to use the Publication Quality Graphics system. The example programs are located on the examples subdirectory.

16.3.2 Graphics Coordinate System

The Publication Quality Graphics uses a 4190x3120 pixel resolution on a 9.0x6.855 inch hardcopy output screen or page. There are 3 units of measure supported with most of the graphics global elements.

Inch Coordinates

Inch coordinates are based on the dimensions of the full-size 9.0x6.855 inch output page. The origin is (0,0) at the lower left corner of the page. If the picture is rotated the origin is at the upper left. See **Inch Units in Graphics Windows** in Section 16.4.1.

Plot Coordinates

Plot coordinates refer to the coordinate system of the graph in the units of the user's X, Y and Z axes.

Pixel Coordinates

Pixel coordinates refer to the 4096x3120 pixel coordinates of the full-size output page. The origin is (0,0) at the lower left corner of the page. If the picture is rotated the origin is at the upper left.

16.4 Graphics Windows

Multiple windows for graphics are supported. These windows allow the user to display multiple graphs on one screen or page.

A *window* is any rectangular subsection of the screen or page. Windows may be any size and position on the screen and may be tiled or overlapping, transparent or nontransparent.

Tiled Windows

Tiled windows do not overlap. The screen can easily be divided into any number of tiled windows with the **window** command. **window** takes three parameters: number of rows, number of columns, and window attribute (1=transparent, 0=nontransparent).

```
window(nrows,ncols,attr);

window(2,3,0);
```

This command will divide the screen into 6 equally sized windows. There will be two rows of three windows, 3 windows in the upper half of the screen and 3 in the lower half. In this case, the attribute value of 0 is arbitrary since there are no other windows beneath them.

Overlapping Windows

Overlapping windows are laid on top of one another as they are created, much as if you were using the cut and paste method to place several graphs together on one page. An overlapping window is created with the **makewind** command:

```
makewind(hsize,vsize,hpos,vpos,attr);

window(2,3,0);
makewind(4,2.5,1,1.5,0);
```

This **makewind** command will create an overlapping window of size 4 inches horizontally by 2.5 inches vertically and positioned 1 inch from the left edge of the page and 1.5 inches from the bottom of the page. It will be nontransparent.

Nontransparent Windows

A nontransparent window is one which is blanked before graphics information is written to it. Therefore, information in any previously drawn windows that lie under it will not be visible.

Transparent Windows

A transparent window is one which is not blanked, allowing the window beneath it to "show through". Transparent windows are useful for adding information to the page without erasing information below it. For example, using a transparent window, the user can add an arrow which begins in one window and ends in another.

16.4.1 Using Window Functions

The following is a summary of the available window functions:

begwind	Window initialization procedure.
endwind	End window manipulations, display graphs.
window	Partition screen into tiled windows.
makewind	Create window with specified size and position.
setwind	Set to specified window number.
nextwind	Set to next available window number.
getwind	Get current window number.
savewind	Save window configuration to a file.
loadwind	Load window configuration from a file.

The following example creates 4 tiled windows and one window which overlaps the other four:

```
library pgraph;
graphset;
begwind;

window(2,2,0);      /* Create four tiled windows (2 rows, 2 columns) */

xsize = 9/2;        /* Create window that overlaps the tiled windows */
ysize = 6.855/2;
makewind(xsize,ysize,xsize/2,ysize/2,0);

x = seqa(1,1,1000);      /* Create X data */
y = (sin(x) + 1) * 10.;  /* Create Y data */

setwind(1);        /* Graph #1, upper left corner  */
    xy(x,y);
nextwind;          /* Graph #2, upper right corner */
    logx(x,y);
nextwind;          /* Graph #3, lower left corner  */
    logy(x,y);
nextwind;          /* Graph #4, lower right corner */
    loglog(x,y);
nextwind;          /* Graph #5, center, overlayed  */
    bar(x,y);
endwind;           /* End window processing, display graph */
```

Inch Units in Graphics Windows

Some global variables allow coordinates to be input in inches. If a coordinate value is in inches and is being used in a window, that value will be scaled to **window inches** and positioned relative to the lower left corner of the window. A **window inch** is a true inch in size only if the window is scaled to the full screen, otherwise **X** coordinates will be scaled relative to the **horizontal** window size and **Y** coordinates will be scaled relative to the **vertical** window size.

For help in finding the exact inch coordinates on the output page for positioning text and other items, see Section 16.5.

16.4.2 Transparent Windows

Lines, symbols, arrows, error bars and other graphics objects may extend from one window to the next by using transparent windows. First create the desired window configuration. Then create a full screen, transparent window using the **makewind** or the **window** command. Set the appropriate global variables to position the desired object on the transparent window. Use the **draw** procedure to draw it. This window will act as a transparent "overlay" on top of the other windows. Transparent windows can be used to add text or to superimpose one window on top of another.

16.4.3 Saving Window Configurations

The functions **savewind** and **loadwind** allow the user to save window configurations. Once windows are created (using **makewind** and **window**) **savewind** may be called. This will save to disk the global variables containing information about the current window configuration. To load this configuration again, call **loadwind**. See Chapter 17.

16.5 Interactive Draft Mode, DOS Only

Interactive Draft mode allows the user to find inch position coordinates on the graph interactively using a mouse or cursor keys. This is useful for locating coordinates for text, symbols, lines and other items which will be added to the graph.

Draft mode is selected by setting **_pnotify** to 2. This will cause the graph to be drawn while the graphics file is being created. When each window is complete, the program will pause before drawing any other windows. At this point, type '**P**'. A cursor will appear and the cursor location will be printed in the lower right corner of the screen.

The cursor may be positioned with the arrow keys (holding down the **Shift** key causes finer movement) or with a mouse.

If the cursor enters the most recent window, two sets of coordinates will be displayed. The upper coordinates are in window inches relative to the current window. The lower coordinates are absolute full-page coordinates. See **Inch Units in Graphics Windows** in Section 16.4.1.

To exit the interactive mode, press the **Esc** key.

This function is available on CGA, EGA, and VGA displays only. Thick lines show up as normal width.

16.6 Fonts and Special Characters

Graphics text elements, such as titles, messages, axes labels, axes numbering and legends can be modified and enhanced by changing fonts, adding superscripting, subscripting and special mathematical symbols.

To make these modifications and enhancements, the user can embed *escape codes* in the text strings which are passed to **title**, **xlabel**, **ylabel** and **asclabel** or assigned to **_pmsgstr** and **_plegstr**.

The escape codes used for graphics text are:

\000	String termination character (null byte).
[	Enter superscript mode, leave subscript mode.
]	Enter subscript mode, leave superscript mode.
@	Interpret next character as literal.
\20n	Select font number n. (see discussion on fonts below)

The escape code "\L" can be embedded into title strings to create a multiple line title:

```
title("This is the first line\lthis is the second line");
```

A null byte "\000" is used to separate strings in **_plegstr** and **_pmsgstr**:

```
_pmsgstr = "First string\000Second string\000Third string";
```

or

```
_plegstr = "Curve 1\000Curve 2";
```

Use the "[..]" to create the expression $M(t) = E(e^{tx})$:

```
_pmsgstr = "M(t) = E(e[tx])";
```

Use the "@" to use "[" and "]" in an X axis label:

```
xlabel("Data used for x is: data@[.,1 2 3@]");
```

16.6.1 Selecting Fonts

Four fonts are supplied with Publication Quality Graphics. They are Simplex, Complex, Simgrma, and Microb. To see the characters in these fonts, see Appendix A.

Fonts are loaded by passing to the **fonts** procedure a string containing the names of all fonts to be loaded. For example, this statement will load all four fonts:

```
fonts("simplex complex microb simgrma");
```

The **fonts** command must be called before any of the fonts may be used in text strings.

One of these fonts may then be selected by embedding an escape code of the form "\20n" in the string which is to be written in the new font. The n will be 1, 2, 3 or 4, depending on the order in which the fonts were loaded in **fonts**. If fonts were loaded as they are in the above example, then the escape characters for each of the fonts would be:

\201	Simplex
\202	Complex
\203	Microb
\204	Simgrma

Now select a font for each string to be written:

```
title("\201This is the title using Simplex font");
xlabel("\202This is the label for X using Complex font");
ylabel("\203This is the label for Y using Microb font");
```

Once a font is selected, all succeeding text will use that font until another font is selected. If no fonts are selected by the user, a default font (Simplex) is loaded and selected automatically for all text work.

16.6.2 Greek and Mathematical Symbols

The following examples illustrate the use of the Simgrma font. This example assumes that Simgrma was the fourth font loaded. The Simgrma font table in Appendix A lists the characters available and their numbers. The Simgrma characters are specified by either:

1. The character number, preceeded by a "\".
2. The regular text character with the same number.

For example, to get an integral sign "$\int$" in Simgrma, embed either a "\044" or a "**,**" in the string that has been currently set to use Simgrma font.

To produce the title $f(x) = sin^2(\pi x)$, the following title string should be used:

```
title("\201f(x) = sin[2](\204p\201x)");
```

The "p" (character 112) corresponds to π in Simgrma.

To number the major X axis tick marks with multiples of $\pi/4$, the following could be passed to **asclabel**:

```
lab = "\2010 \204p\201/4 \204p\201/2  3\204p\201/4 \204p";
asclabel(lab,0);
xtics(0,pi,pi/4,1);
```

xtics is used to make sure that major tick marks are placed in the appropriate places.

This example will number the X axis tick marks with the labels μ^{-2}, μ^{-1}, 1, μ, and μ^2:

```
lab = "\204m\201[-2] \204m\201[-1] 1 \204m m\201[2]";
asclabel(lab,0);
```

This example illustrates the use of several of the special Simgrma symbols:

```
_pmsgstr =  "\2041\2011/2\204p ,\201e[-\204m[\2012]\201/2]d\204m";
```

This produces:

$$\sqrt{1/2\pi} \int e^{-\mu^2/2} d\mu$$

16.7 Hardcopy and Exporting

When a graphics procedure is called, a graphics file is created. This file contains graphics commands which may be used later for viewing or printing.

When the graph is displayed on the screen, the user may, in DOS, press the **Enter** key to obtain a menu containing available options. If using OS/2 or Unix simply select one of the menu bar options. These options allow the user to print the graph, zoom in, or convert the file to another format.

16.7.1 Printing the Graph

The graph may be printed by selecting one of the interactive print options: **normal** or **draft**. The normal print uses a resolution level taken from the PQG configuration file. The fast draft print always uses a low resolution print. Most of the print parameters such as page size, margins, and orientation may be changed by editing the configuration file or by using the **graphprt** procedure. See Chapter 17 for more information on **graphprt**.

If you have purchased the optional DOS `play.exe` program you may choose to print your graph using the options available with it. `play.exe` is documented in the DOS supplement.

Zooming

To zoom in on a graph interactively, press **Z** from the graphics display, or select zoom from the menu bar. An arrow will appear in the center of the graph with a *rubberband* box. You may size the box using the the cursor keys or by moving the mouse. Pressing the right mouse button (or **Ctrl-Cursor** keys, DOS only) will allow you to move the box. Pressing the **Enter** key or left mouse button will select the box as the area to zoom.

Press **Esc** to cancel zoom mode.

The user may bypass the interactive menu by using a global variable to pass zoom parameters to the graphics routines. The parameters are set with a 3x1 matrix named **_pzoom**. See Section 16.8 for a description of this global.

16.7.2 Exporting Graphs

By pressing **C**, in DOS, from either the graphics menu or the displayed graph, or selecting File from the menu bar, in OS/2, the graphics file may be converted into other formats. These formats are widely used by other applications for importing graphs into word processing and desktop publishing programs. The currently available format types are:

Encapsulated PostScript file.
Lotus .PIC file
HP-GL Plotter file
PCX bitmap image

DOS only

Other export options are available with the optional program `play.exe`. For a complete description of command parameters to `play.exe`, see the DOS supplement.

UNIX only

The convert menu option will convert the graph to the file type specified by the `gauss*print.driver` variable located in the `.gaussrc` file. Other export file options are located in the `prndev.tbl` file.

16.8 Global Control Variables

The following global variables are used to control various graphics elements. Default values are provided. Any and all of these variables may be set before calling one of the main graphing routines. The default values may be modified by changing the declarations in `pgraph.dec` and the statements in the procedure **graphset** in `pgraph.src`. **graphset** may be called anytime the user wishes to reset these variables to their default values.

_pageshf 2x1 vector, the graph will be shifted to the right and up if this is not zero. If this is zero, the graph will be centered on the output page. The default is 0.

Note: Used internally, see **axmargin** for the same functionality. This is used by the graphics window routines. The user must not set this when using the window procedures.

_pagesiz 2x1 vector, size of the graph in inches on the printer output. Maximum size is 9.0 x 6.855 inches (unrotated) or 6.855 x 9.0 inches (rotated). If this is 0, the maximum size will be used. The default is 0.

Note: Used internally, see **axmargin** for the same functionality. This is used by the graphics window routines. The user must not set this when using the window procedures.

_parrow Mx11 matrix, draws one arrow per row **M** of the input matrix. If scalar zero, no arrows will be drawn.

[M,1] x starting point.

[M,2] y starting point.

[M,3] x ending point.

[M,4] y ending point.

[M,5] ratio of the length of the arrow head to half its width.

[M,6] size of arrow head in inches.

[M,7] type and location of arrow heads. This integer number will be interpreted as a decimal expansion mn, for example: if 10, then

$m = 1$, $n = 0$

m, form of arrow head

0 solid.
1 empty.
2 open.
3 closed.

n, location of arrow head

0 none.
1 at the final end.
2 at both ends.

[M,8] color of arrow.

[M,9] coordinate units for location.

1 x,y starting and ending locations in plot coordinates.
2 x,y starting and ending locations in inches.
3 x,y starting and ending locations in pixels.

[M,10] line type

1 dashed.
2 dotted.
3 short dashes.
4 closely spaced dots.
5 dots and dashes.
6 solid.

[M,11] Controls. thickness of lines used to draw arrow. This value may be zero or greater. A value of zero is normal line width.

The following statement could be used to create two single-headed arrows. They will be located using inches.

```
_parrow = {  1  1  2  2  3  0.2  11  10  2  6  0,
             3  4  2  2  3  0.2  11  10  2  6  0 };
```

_parrow3 Mx12 matrix, draws one 3-D arrow per row of the input matrix. If scalar zero, no arrows will be drawn.

[M,1] x starting point in 3-D plot coordinates.
[M,2] y starting point in 3-D plot coordinates.
[M,3] z starting point in 3-D plot coordinates.
[M,4] x ending point in 3-D plot coordinates.
[M,5] y ending point in 3-D plot coordinates.
[M,6] z ending point in 3-D plot coordinates.
[M,7] ratio of the length of the arrow head to its half width.
[M,8] size of arrow head in inches.

[M,9] type and location of arrow heads. This integer number will be interpreted as a decimal expansion mn, for example:
if 10, then m = 1, n = 0

m, form of arrow head

0 solid.
1 empty.
2 open.
3 closed.

n, location of arrow head

0 none.
1 at the final end.
2 at both ends.

[M,10] color of arrow.

[M,11] line type.

1 dashed.
2 dotted.
3 short dashes.
4 closely spaced dots.
5 dots and dashes.
6 solid.

[M,12] controls thickness of lines used to draw arrow. This value may be zero or greater. A value of zero is normal line width.

The following statement could be used to create two single-headed arrows. They will be located using plot coordinates.

```
_parrow3 = {  1  1  1  2  2  2  3  0.2  11  10  6  0,
              3  4  5  2  2  2  3  0.2  11  10  6  0 };
```

_paxes scalar, 2x1, or 3x1 vector for independent control for each axis. The first element controls the X axis, the second controls the Y axis, and the third (if set) will control the Z axis. If 0 the axis will not be drawn. Default is 1.

If this is a scalar, it will be expanded to that value.

For example:

```
_paxes = { 1, 0 }; /* turn X axis on, Y axis off */
_paxes = 0;        /* turn all axes off */
_paxes = 1;        /* turn all axes on */
```

_paxht scalar, size of axes labels in inches. If 0, a default size will be computed. Default 0.

_pbartyp global 1x2 or Kx2 matrix. Controls bar shading and colors in bar graphs and histograms.

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.

_pbarwid global scalar, width 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. The default is 0.5, so the bars take up about half the space open to them.

_pbox scalar, draws a box (border) around the entire graph. Set to desired color of box to be drawn. Use 0 if no box is desired. Default 0.

_pboxctl 5x1 vector, controls box plot style, width, and color. Used by procedure **box** only.

[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 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.

_pboxlim 5xM output matrix containing computed percentile results from procedure **box**. M corresponds to each column of input y data.

[1,M] minimum whisker limit according to **_pboxctl[3]**.

[2,M] 25th percentile (bottom of box).

[3,M] 50th percentile (median).

[4,M] 75th percentile (top of box).

[5,M] maximum whisker limit according to **_pboxctl[3]**.

_pcolor scalar or Kx1 vector, colors for main curves in **xy**, **xyz** and **log** graphs. To use a single color set for all curves set this to a scalar color value. If 0, use default colors. Default 0.

The default colors come from a global vector called **_pcsel**. This vector can be changed by editing `pgraph.dec` to change the default colors. **_pcsel** is not documented elsewhere.

_pcrop scalar or 1x5 vector, allows plot cropping for different graphic elements to be individually controlled. Valid values are 0 (disabled) or 1 (enabled). If cropping is enabled, any graphical data sent outside the axes area will not be drawn. If this is scalar, **_pcrop** is expanded to a 1x5 vector using the given value for all elements. All cropping is enabled by default.

[1] Crop main curves/symbols.

[2] Crop lines generated using **_pline**.

[3] Crop arrows generated using **_parrow**.

[4] Crop circles/arcs generated using **_pline**.

[5] Crop symbols generated using **_psym**.

```
_pcrop = { 1 1 0 1 0 };
```

This will crop main curves, and lines and circles drawn by **_pline**.

_pcross scalar. If 1, the axes will intersect at the (0,0) X-Y location if it is visible. The default is 0, meaning the axes will be at the lowest end of the X-Y coordinates.

_pdate date string. If this contains characters, the date will be appended and printed. The default is set as follows (note the first character is a font selection escape code):

```
_pdate = "\201GAUSS   ";
```

If this is set to a null string, no date will be printed. See Section 16.6 for more information on using fonts within strings.

_perrbar Mx9 matrix, draws one error bar per row of the input matrix. If scalar 0, no error bars will be drawn.

[M,1] x location in plot coordinates.

[M,2] left end of error bar.

[M,3] right end of error bar.

[M,4] y location in plot coordinates.

[M,5] bottom of error bar.

[M,6] top of error bar.

[M,7] line type.

- **1** dashed.
- **2** dotted.
- **3** short dashes.
- **4** closely spaced dots.
- **5** dots and dashes.
- **6** solid.

[M,8] color.

[M,9] line thickness.. This value may be zero or greater. A value of zero is normal line width.

The following statement could be used to create one error bar using solid lines:

```
_perrbar = { 1  0  2  2  1  3  6  2  0 };
```

_pframe 2x1 vector, controls frame around axes area. On 3-D plots this is a cube surrounding the 3-D workspace.

[1]

- **1** frame on.
- **0** frame off.

[2]

- **1** tick marks on frame.
- **0** no tick marks.

The default is a frame with tick marks.

_pgrid 2x1 vector to control grid.

[1] grid through tick marks.

- **0** no grid.
- **1** dotted grid.
- **2** fine dotted grid.
- **3** solid grid.

[2] grid subdivisions between major tick marks.

- **0** no subdivisions.
- **1** dotted lines at subdivisions.
- **2** tick marks only at subdivisions.

The default is no grid and tick marks at subdivisions.

_plctrl scalar or Kx1 vector to control whether lines and/or symbols will be displayed for the main curves. This also controls the frequency of symbols on main curves. The rows (K) is equal to the number of individual curves to be plotted in the graph. Default 0.

- **0** draw line only.
- >**0** draw line and symbols every **_plctrl** points.
- <**0** draw symbols only every **_plctrl** points.
- −1 all of the data points will be plotted with no connecting lines.

```
_plctrl = { 0, 10, -5 };
```

This example draws a line for the first curve, draws a line and plots a symbol every 10 data points for the second curve, and plots symbols only every 5 data points for the third curve.

_plegctl scalar or 1x4 vector, legend control variable.

If scalar 0, no legend is drawn (default). If nonzero scalar, create legend in the default location in the lower right of the page.

If 1x4 vector, set as follows:

[1] Legend position coordinate units.

- **1** Coordinates are in plot coordinates.
- **2** Coordinates are in inches.
- **3** Coordinates are in pixels.

[2] Legend text font size. $1 \leq \text{size} \leq 9$. Default 5.

[3] x coordinate of lower left corner of legend box.

[4] y coordinate of lower left corner of legend box.

```
_plegctl = 1;
```

This example puts a legend in the lower right corner.

```
_plegctl = { 2 3 2.5 1 };
```

This example creates a smaller legend and positions it 2.5 inches from the left and 1 inch from the bottom.

_plegstr string, legend entry text. Text for multiple curves is separated by a null byte ("\000").

For example:

```
_plegstr = "Curve 1\000Curve 2\000Curve 3";
```

_plev Mx1 vector, user-defined contour levels for **contour**. Default 0. See **contour**.

_pline Mx9 matrix, to draw lines, circles, or radii. Each row controls one item to be drawn. If this is a scalar zero, nothing will be drawn. The default is 0.

[M,1] item type and coordinate system.

- **1** line in plot coordinates.
- **2** line in inch coordinates.
- **3** line in pixel coordinates.
- **4** circle in plot coordinates.
- **5** circle in inch coordinates.
- **6** radius in plot coordinates.
- **7** radius in inch coordinates.

[M,2] line type.

- **1** dashed.
- **2** dotted.
- **3** short dashes.
- **4** closely spaced dots.
- **5** dots and dashes.
- **6** solid.

[M,3-7] coordinates and dimensions.

(1) Line in plot coordinates.

[M,3] x starting point in plot coordinates.

[M,4] y starting point in plot coordinates.

[M,5] x ending point in plot coordinates.

[M,6] y ending point in plot coordinates.

[M,7] 0 if this is a continuation of a curve, 1 if this begins a new curve.

(2) Line in inches.

[M,3] x starting point in inches.

[M,4] y starting point in inches.

[M,5] x ending point in inches.

[M,6] y ending point in inches.

[M,7] 0 if this is a continuation of a curve, 1 if this begins a new curve.

(3) Line in pixel coordinates.

[M,3] x starting point in pixels.

[M,4] y starting point in pixels.

[M,5] x ending point in pixels.

[M,6] y ending point in pixels.

[M,7] 0 if this is a continuation of a curve, 1 if this begins a new curve.

(4) Circle in plot coordinates.

[M,3] x center of circle in plot coordinates.

[M,4] y center of circle in plot coordinates.

[M,5] radius in x plot units.

[M,6] starting point of arc in radians.

[M,7] ending point of arc in radians.

(5) Circle in inches.

[M,3] x center of circle in inches.

[M,4] y center of circle in inches.

[M,5] radius in inches.

[M,6] starting point of arc in radians.

[M,7] ending point of arc in radians.

(6) Radius in plot coordinates.

[M,3] x center of circle in plot coordinates.

[M,4] y center of circle in plot coordinates.

[M,5] beginning point of radius in x plot units, 0 is the center of the circle.

[M,6] ending point of radius.

[M,7] angle in radians.

(7) Radius in inches.

[M,3] x center of circle in inches.

[M,4] y center of circle in inches.

[M,5] beginning point of radius in inches, 0 is the center of the circle

[M,6] ending point of radius in inches.

[M,7] angle in radians.

[M,8] color.

[M,9] Controls line thickness. This value may be zero or greater. A value of zero is normal line width.

_pline3d Mx9 matrix. Allows extra lines to be added to an **xyz** or **surface** graph in 3-D plot coordinates.

[M,1] x starting point.

[M,2] y starting point.

[M,3] z starting point.

[M,4] x ending point.

[M,5] y ending point.

[M,6] z ending point.

[M,7] color.

[M,8] line type.

- **1** dashed.
- **2** dotted.
- **3** short dashes.
- **4** closely spaced dots.
- **5** dots and dashes.
- **6** solid.

[M,9] Line thickness, 0 = normal width.

[M,10] Hidden line flag, 1 = obscured by surface, 0 = not obscured.

_plotshf 2x1 vector, distance of plot from lower left corner of output page in inches. [1] is x distance, [2] is y distance. If scalar 0, there will be no shift. Default 0.

Note: Used internally, see **axmargin** for the same functionality. This is used by the graphics window routines. The user must not set this when using the window procedures.

_plotsiz 2x1 vector, size of the axes area in inches. If scalar 0, the maximum size will be used.

Note: Used internally, see **axmargin** for the same functionality. This is used by the graphics window routines. The user must not set this when using the window procedures.

_pltype scalar or Kx1 vector, line type for the main curves. If this is a nonzero scalar, all lines will be this type. If scalar 0, line types will be default styles. The default is 0.

- **1** dashed.
- **2** dotted.
- **3** short dashes.
- **4** closely spaced dots.
- **5** dots and dashes.
- **6** solid.

The default line types come from a global vector called **_plsel**. This vector can be changed by editing `pgraph.dec` to change the default line types. **_plsel** is not documented elsewhere.

_plwidth scalar or Kx1 vector, line thickness for main curves. This value may be zero or greater. A value of zero is normal (single pixel) line width. Default 0.

_pmcolor 9x1 vector, color values to use for plot

- **[1]** axes.
- **[2]** axes numbers.
- **[3]** X axis label.
- **[4]** Y axis label.
- **[5]** Z axis label.
- **[6]** title.
- **[7]** box.
- **[8]** date.
- **[9]** background.

If this is scalar, it will be expanded to a 9x1 vector

_pmsgctl Lx7 matrix of control information for printing the strings contained in **_pmsgstr**.

- **[L,1]** horizontal location of lower left corner of string.
- **[L,2]** vertical location of lower left corner of string.
- **[L,3]** character height in inches.
- **[L,4]** angle in degrees to print string. This may be -180 to 180 relative to the positive X axis.
- **[L,5]** location coordinate system.
  - **1** location of string in plot coordinates.
  - **2** location of string in inches.
- **[L,6]** color.
- **[L,7]** Font thickness, may be zero or greater. If 0 use normal line width.

_pmsgstr string, contains a set of messages to be printed on the plot. Each message is separated from the next with a null byte (\000). The number of messages must correspond to the number of rows in the **_pmsgctl** control matrix. This can be created as follows:

```
_pmsgstr = "Message one.\000Message two.";
```

_pnotify scalar, controls screen output during the creation of the graph. Default 1.

0	No activity to the screen while writing `.tkf` file.
1	Display progress as fonts are loaded, and `.tkf` file is being generated.
2	Display graph while writing `.tkf` file and enter interactive draft mode.

_pnum scalar, 2x1 or 3x1 vector for independent control for axes numbering. The first element controls the X axis numbers, the second controls the Y axis numbers, and the third (if set) controls the Z axis numbers. Default is 1.

If this value is scalar, it will be expanded to a vector.

0	No axes numbers displayed.
1	Axes numbers displayed, vertically oriented on Y axis.
2	Axes numbers displayed, horizontally oriented on Y axis.

For example:

```
_pnum = { 0, 2 }; /* no X axis numbers, horizontal on Y axis */
```

_pnumht scalar, size of axes numbers in inches. If 0 (default), a size of .13 will be used.

_protate scalar, if 0, no rotation, if 1, plot will be rotated 90 degrees. Default 0.

_pscreen scalar, if 1, display graph on screen, if 0, do not display graph on screen. Default 1.

_psilent scalar, if 0, a beep will sound when the graph is finished drawing to the screen. Default 1 (no beep).

_pstype scalar or Kx1 vector, controls symbol used at graph points. To use a single symbol type for all points set this to one of the following scalar values:

1	Circle	**8**	Solid Circle
2	Square	**9**	Solid Square
3	Triangle	**10**	Solid Triangle
4	Plus	**11**	Solid Plus
5	Diamond	**12**	Solid Diamond
6	Inverted Triangle	**13**	Solid Inverted Triangle
7	Star	**14**	Solid Star

If this is a vector, each line will have a different symbol. Symbols will repeat if there are more lines than symbol types.

_psurf 2x1 vector, controls 3-D surface characteristics.

- **[1]** if 1, show hidden lines. Default 0.
- **[2]** Color for base (default 7). The base is an outline of the X-Y plane with a line connecting each corner to the surface. If 0 no base is drawn.

_psym Mx7 matrix, M extra symbols will be plotted.

- [**M,1**] x location.
- [**M,2**] y location.
- [**M,3**] symbol type. See **_pstype**.
- [**M,4**] symbol height. If this is 0, a default height of 5.0 will be used.
- [**M,5**] symbol color.
- [**M,6**] type of coordinates.
  - **1** plot coordinates.
  - **2** inch coordinates.
- [**M,7**] line thickness. A value of zero is normal line width.

_psym3d Mx7 matrix for plotting extra symbols on a 3-D (**surface** or **xyz**) graph.

- [**M,1**] x location in plot coordinates.
- [**M,2**] y location in plot coordinates.
- [**M,3**] z location in plot coordinates.
- [**M,4**] symbol type. See **_pstype**.
- [**M,4**] symbol height. If this is 0, a default height of 5.0 will be used.
- [**M,6**] symbol color.
- [**M,7**] Line thickness. A value of 0 is normal line width.

Use **_psym** for plotting extra symbols in inch coordinates.

_psymsiz scalar or Kx1 vector, symbol size for the symbols on the **main curves**. This is NOT related to **_psym**. If 0, a default size of 5.0 is used.

_ptek string, name of Tektronix format graphics file. This must have a `.tkf` extension. If this is set to a null string, the graphics file will be suppressed. The default is `graphic.tkf`.

_pticout scalar, if 1, tick marks point outward on graphs. Default 0.

_ptitlht scalar, the height of the title characters in inches. If this is 0, a default height of approx. 0.13" will be used.

_pversno string, the graphics version number.

_pxpmax scalar, the maximum number of places to the right of the decimal point for the X axis numbers. Default 12.

_pxsci scalar, the threshold in digits above which the data for the X axis will be scaled and a power of 10 scaling factor displayed. Default 4.

_pypmax scalar, the maximum number of places to the right of the decimal point for the Y axis numbers. Default 12.

_pysci scalar, the threshold in digits above which the data for the Y axis will be scaled and a power of 10 scaling factor displayed. Default 4.

_pzclr scalar, row vector, or Kx2 matrix, Z level color control for procedures **surface** and **contour**. See **surface**.

_pzoom 1x3 row vector, magnifies the graphics display for zooming in on detailed areas of the graph. If scalar 0 (default), no magnification is performed.

[1] Magnification value. 1 is normal size.

[2] Horizontal center of zoomed plot (0-100).

[3] Vertical center of zoomed plot (0-100).

To see the upper left quarter of the screen magnified 2 times use:

```
_pzoom = { 2 25 75 };
```

_pzpmax scalar, the maximum number of places to the right of the decimal point for the Z axis numbers. Default 3.

_pzsci scalar, the threshold in digits above which the data for the Z axis will be scaled and a power of 10 scaling factor displayed. Default 4.

Chapter 17

Graphics Reference

Purpose

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

Library

pgraph

Format

asclabel(xl,yl**);**

Input

xl	string or Nx1 character vector, labels for the tick marks on the X axis. Set to 0 if no character labels for this axis are desired.
yl	string or Mx1 character vector, labels for the tick marks on the Y axis. Set to 0 if no character labels for this axis are desired.

Example

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

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

This will also work:

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

If the string format is used, then escape characters may be embedded in the labels. For example, the following produces character labels that are multiples of λ. The font Simgrma must be previously loaded in a **fonts** command. See Section 16.6.

```
fonts("simplex simgrma");
lab = "\2010.25\202l \2010.5\202l \2010.75\202l 1";
asclabel(lab,0);
```

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

Source

pgraph.src

Globals

_pascx, **_pascy**

See also

xtics, **ytics**, **scale**, **scale3d**, **fonts**

Purpose

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

Library

pgraph

Format

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

Input

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

Remarks

axmargin sets an absolute distance from the axes to the edge of the window. 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 screen of 9 x 6.855 inches. If this procedure is used within a window, 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

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

This 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 window/page.

Source

pgraph.src

Globals

_paxmarx, **_plotshf**, **_plotsiz**

Purpose

Bar graph.

Library

pgraph

Format

bar(*val*,*ht*);

Input

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

ht NxK numeric vector, bar heights.

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.

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

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

_pbartyp Kx2 matrix.

The first column controls the bar shading:

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

The second column controls the bar color.

Remarks

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

Example

```
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. */
```

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

Source

pbar.src

Globals

rerun, **xtics**, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_pascx**, **_pascy**, **_paxes**, **_paxht**, **_paxnum**, **_pbartyp**, **_pbarwid**, **_pbox**, **_pcartx**, **_pcarty**, **_pcrop**, **_pcross**, **_pcwin**, **_pdate**, **_perrbar**, **_pfonts**, **_pgrid**, **_plegctl**, **_plegstr**, **_plotshf**, **_plotsiz**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_pqgtype**, **_protate**, **_pscreen**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pworld**, **_pwrscal**, **_pxfmt**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pxsci**, **_pyfmt**, **_pylabel**, **_pypmax**, **_pyscale**, **_pysci**, **_setpage**, **_txtfilt**

See also

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

Purpose

Initialize global window variables.

Library

pgraph

Format

begwind;

Remarks

This procedure must be called before any other window functions are called.

Source

pwindow.src

Globals

_pageshf, **_pagesiz**, **_pappend**, **_pcwin**, **_pdate**, **_pfirstw**, **_pncwin**, **_pscreen**, **_pwindmx**, **_pwindno**

See also

endwind, **window**, **makewind**, **setwind**, **nextwind**, **getwind**

Purpose

Graph data using the box graph percentile method.

Library

pgraph

Format

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

Input

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

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

Global Input

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

[1] box width between 0 and 1. If zero, the box plot is drawn as two vertical lines representing the quartile ranges with a filled circle representing the 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 1xM vector or scalar as follows:

0	Plot boxes only, no symbols.
1	Plot boxes and plot symbols which lie outside the *min* and *max* box values.
2	Plot boxes and all symbols.
-1	Plot symbols only, no boxes.

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

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

Remarks

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

Source

`pbox.src`

Globals

rerun, **xtics**, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_pascx**, **_pascy**, **_paxes**, **_paxht**, **_paxnum**, **_pbox**, **_pboxctl**, **_pcartx**, **_pcarty**, **_pcolor**, **_pcrop**, **_pcross**, **_pcsel**, **_pcwin**, **_pdate**, **_perrbar**, **_pfonts**, **_pgrid**, **_plctrl**, **_plegctl**, **_plegstr**, **_pline**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_plwidth**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_pqgtype**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pworld**, **_pwrscal**, **_pxfmt**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pxsci**, **_pyfmt**, **_pylabel**, **_pypmax**, **_pyscale**, **_pysci**, **_setpage**, **_txtfilt**

Purpose

To graph a matrix of contour data.

Library

pgraph

Format

contour(*x*,*y*,*z*);

Input

x	1xK vector, the X axis data. K must be odd.
y	Nx1 vector, the Y axis data. N must be odd.
z	NxK matrix, the matrix of height data to be plotted.
_plev	Kx1 vector, user-defined contour levels for **contour**. Default 0.
_pzclr	Nx1 or Nx2 vector. This controls the Z level colors. See **surface** for a complete description of how to set this global.

Remarks

A vector of evenly spaced contour levels will be generated automatically from the *z* matrix data. Each contour level will be labeled. For unlabeled contours, use **ztics**.

To specify a vector of your own unequal contour levels, set the vector **_plev** before calling **contour**.

To specify your own evenly spaced contour levels, see **ztics**.

Source

pcontour.src

Globals

rerun, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_pascx**, **_pascy**, **_paxes**, **_paxht**, **_paxnum**, **_pbox**, **_pcartx**, **_pcarty**, **_pcartz**, **_pcrop**, **_pcross**, **_pcwin**, **_pdate**, **_perrbar**, **_pfonts**, **_pgrid**, **_plctrl**, **_plegctl**, **_plegstr**, **_plev**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_pqgtype**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pworld**, **_pwrscal**, **_pxfmt**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pxsci**, **_pyfmt**, **_pylabel**, **_pypmax**, **_pyscale**, **_pysci**, **_pzclr**, **_pzfmt**, **_pzpmax**, **_pzscale**, **_setpage**, **_txtfilt**

See also

surface

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

Globals

rerun, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_pascx**, **_pascy**, **_paxes**, **_paxht**, **_paxnum**, **_pbox**, **_pcartx**, **_pcarty**, **_pcolor**, **_pcrop**, **_pcross**, **_pcsel**, **_pcwin**, **_pdate**, **_perrbar**, **_pfonts**, **_pgrid**, **_plctrl**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_pqgtype**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pworld**, **_pwrscal**, **_pxfmt**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pxsci**, **_pyfmt**, **_pylabel**, **_pypmax**, **_pyscale**, **_pysci**, **_setpage**, **_txtfilt**

See also

window, **makewind**

Purpose

End window manipulation, display graphs with **rerun**.

Library

pgraph

Format

endwind;

Remarks

This function uses **rerun** to display the most recently created `.tkf` file.

Source

pwindow.src

Globals

rerun, **_pageshf**, **_pagesiz**, **_pappend**, **_pfirstw**, **_pscreen**, **_pwindmx**, **_pwindno**

See also

begwind, **window**, **makewind**, **setwind**, **nextwind**, **getwind**

Purpose

Load 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.

Simplex	standard sans serif font.
Simgrma	Simplex greek, math.
Microb	bold and boxy.
Complex	standard font with serif.

The first font specified will be used for the axes numbers.

If *str* is a null string, or **fonts** is not called, Simplex is loaded by default.

Remarks

See Section 16.6 for instructions on how to select fonts within a text string.

Source

pgraph.src

Globals

_pfonts

See also

title, **xlabel**, **ylabel**, **zlabel**

Purpose

Retrieve the current window number.

Library

pgraph

Format

n = **getwind;**

Output

n	scalar, window number of current window.

Remarks

The current window is the window in which the next graph will be drawn.

Source

pwindow.src

Globals

_pwindno

See also

endwind, **begwind**, **window**, **setwind**, **nextwind**

Purpose

Controls automatic printer hardcopy and conversion file output.

Library

pgraph

Format

graphprt(*str***);**

Input

str string, control string.

Remarks

graphprt is used to create hardcopy output automatically without user intervention. The input string *str* can have any of the following items, separated by spaces. If *str* is a null string, the interactive mode is entered. This is the default.

-P print graph.

-PF=*name* set output print file name.

-PO=*c* set print orientation.

- **L** landscape.
- **P** portrait.

-PS=*c* set print page size.

- **Q** quarter page.
- **H** half page.
- **F** full page.

-PM=*l,r,t,b* set print margins to *left,right,top,bottom* in inch units.

-PR=*n* set print resolution.

- **1** low.
- **2** medium.
- **3** high.

-C=*n* convert to another file format.

- **1** Encapsulated PostScript file.
- **2** Lotus .PIC file.
- **3** HPGL Plotter file.
- **4** PCX bitmap format.

-CF=*name* set converted output file name.

-CO=*n* set conversion file orientation.

- **L** landscape.
- **P** portrait.

-CS=*c* set conversion file page size.

- **Q** quarter page.
- **H** half page.
- **F** full page.

-CM=*l,r,t,b* set conversion file margins to *left,right,top,bottom* in inch units.

-W=*n* display graph, wait *n* seconds, then continue.

If you are not using windows, you can call **graphprt** anytime before the call to the graphics routine. If you are using windows, call **graphprt** just before the **endwind** statement.

The print option default values are obtained from the graphics configuration file. Any parameters passed through **graphprt** will override the default values. See Section **??**.

Under DOS this uses a utility called **pqgrun**. **pqgrun** is documented in the DOS supplement.

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 graphics windows. Note **graphprt** is called once just before the **endwind** call.

```
library pgraph;
graphset;
load x,y;
begwind;
window(1,2);      /* create two windows  */

setwind(1);
    xy(x,y);        /* first graphics call */

nextwind;
    xy(x,y);        /* second graphics call */

graphprt("-p");
endwind;          /* print page containing all graphs */
```

The next example shows how to build a string to be used with **graphprt**.

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

/* concatenate options into one string */
cmdstr = "-p" $+ " -pf=" $+ prtnam $+ " -pr=" $+ prtres;
cmdstr = cmdstr $+ " -pm=" $+ prtmarg;

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

The above string *cmdstr* will look like this:

```
"-p -pf=myprint.prn -pr=1 -pm=.5,.5,.5,.5"
```

Source

pgraph.src

Globals

_pcmdlin

See also

_ptek

Purpose

Reset graphics globals to default values.

Library

pgraph

Format

graphset;

Remarks

This procedure is used to reset the defaults between graphs.

graphset may be called between each window 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

Globals

_pageshf, **_pagesiz**, **_parrow**, **_parrow3**, **_pascx**, **_pascy**, **_paxes**, **_paxht**, **_paxmarx**, **_pbartyp**, **_pbarwid**, **_pbox**, **_pbox3d**, **_pboxctl**, **_pcolor**, **_pcrop**, **_pcross**, **_pcsel**, **_pcwin**, **_perrbar**, **_pfonts**, **_pframe**, **_pgrid**, **_plctrl**, **_plegctl**, **_plegstr**, **_plev**, **_pline**, **_pline3d**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_plwidth**, **_pmargin**, **_pmcolor**, **_pmsgctl**, **_pmsgstr**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psurf**, **_psym**, **_psym3d**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pview**, **_pvolume**, **_pwindmx**, **_pworld**, **_pxfmt**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pxsci**, **_pyfmt**, **_pylabel**, **_pypmax**, **_pyscale**, **_pysci**, **_pzclr**, **_pzfmt**, **_pzlabel**, **_pzoom**, **_pzpmax**, **_pzscale**

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	Mx1 vector of data.
v	Nx1 vector, the breakpoints to be used to compute the frequencies,
	or
	scalar, the number of categories.

Output

b	Px1 vector, the breakpoints used for each category.
m	Px1 vector, the midpoints of each category.
freq	Px1 vector of computed frequency counts.

Remarks

If a vector of breakpoints is specified, a final breakpoint equal to the maximum value of x will be added if the maximum breakpoint value is smaller.

If a number of categories is specified, the data will be divided into v evenly spaced categories.

Each time an element falls into one of the categories specified in b, the corresponding element of *freq* will be incremented by one. The categories are interpreted as follows:

$$
\begin{array}{lcllcl}
freq[1] & = & & x & <= & b[1] \\
freq[2] & = & b[1] < & x & <= & b[2] \\
freq[3] & = & b[2] < & x & <= & b[3] \\
\quad . & & & & & \\
\quad . & & & & & \\
\quad . & & & & & \\
freq[P] & = & b[P-1] < & x & <= & b[P]
\end{array}
$$

Example

```
library pgraph;
x = rndn(5000,1);
{ b,m,f } = hist(x,20);
```

Source

phist.src

Globals

bar, **_pascx**, **_pcolor**, **_pcsel**, **_pmcolor**, **_pmsgctl**, **_pmsgstr**, **_pnum**, **_ptitle**, **_pworld**, **_pxlabel**, **_pxpmax**, **_pylabel**

See also

histp, **histf**, **bar**

Purpose

To graph a histogram given a vector of frequency counts.

Library

pgraph

Format

histf(*f*,*c*);

Input

f	Nx1 vector, frequencies to be graphed.
c	Nx1 vector, numeric labels for categories. If this is a scalar 0, a sequence from 1 to **rows**(*f*) will be created.

Remarks

The axes are not automatically labeled. Use **xlabel** for the category axis and **ylabel** for the frequency axis.

Source

phist.src

Globals

bar, **_pcolor**, **_pcsel**, **_pworld**

See also

hist, **bar**, **xlabel**, **ylabel**

Purpose

Computes and graphs a percent frequency histogram of a vector. The percentages in each category are plotted.

Library

`pgraph`

Format

`{` b,m,*freq* `}` = **histp(**x,v**);**

Input

x	Mx1 vector of data.
v	Nx1 vector, the breakpoints to be used to compute the frequencies, or scalar, the number of categories.

Output

b	Px1 vector, the breakpoints used for each category.
m	Px1 vector, the midpoints of each category.
freq	Px1 vector of computed frequency counts. This is the vector of counts, not percentages.

Remarks

If a vector of breakpoints is specified, a final breakpoint equal to the maximum value of x will be added if the maximum breakpoint value is smaller.

If a number of categories is specified, the data will be divided into v evenly spaced categories.

Each time an element falls into one of the categories specified in b, the corresponding element of *freq* will be incremented by one. The categories are interpreted as follows:

$$
\begin{array}{lcccccl}
freq[1] & = & & & x & <= & b[1] \\
freq[2] & = & b[1] & < & x & <= & b[2] \\
freq[3] & = & b[2] & < & x & <= & b[3] \\
\quad . & & & & & & \\
\quad . & & & & & & \\
\quad . & & & & & & \\
freq[P] & = & b[P-1] & < & x & <= & b[P]
\end{array}
$$

Source

`phist.src`

Globals

bar, **_pascx**, **_pcolor**, **_pcsel**, **_pmcolor**, **_pmsgctl**, **_pmsgstr**, **_pnum**, **_ptitle**, **_pworld**, **_pxlabel**, **_pxpmax**, **_pylabel**

See also

hist, histf, bar

Purpose

Load a previously saved window configuration.

Library

pgraph

Format

err = **loadwind(***namestr***);**

Input

namestr	string, name of file to be loaded.

Output

err	scalar, 0 if successful, 1 if window matrix is invalid. Note that the current window configuration will be overwritten in either case.

Source

pwindow.src

Globals

_pwindmx

See also

savewind

Purpose

Graphs X vs. Y using log coordinates.

Library

pgraph

Format

loglog(x**,**y**);**

Input

x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

Source

ploglog.src

Globals

rerun, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_paxes**, **_paxht**, **_pbox**, **_pcartx**, **_pcarty**, **_pcolor**, **_pcrop**, **_pcross**, **_pcsel**, **_pcwin**, **_pdate**, **_perrbar**, **_pfonts**, **_pgrid**, **_plctrl**, **_plegctl**, **_plegstr**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_plwidth**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_pqgtype**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pworld**, **_pxlabel**, **_pxscale**, **_pylabel**, **_pyscale**, **_setpage**, **_txtfilt**

See also

xy, **logy**, **logx**

Purpose

Graphs X vs. Y using log coordinates for the X axis.

Library

pgraph

Format

logx(x,y**);**

Input

x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

Source

plogx.src

Globals

rerun, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_pascy**, **_paxes**, **_paxht**, **_paxnum**, **_pbox**, **_pcartx**, **_pcarty**, **_pcolor**, **_pcrop**, **_pcross**, **_pcsel**, **_pcwin**, **_pdate**, **_perrbar**, **_pfonts**, **_pgrid**, **_plctrl**, **_plegctl**, **_plegstr**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_plwidth**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_pqgtype**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pworld**, **_pwrscal**, **_pxlabel**, **_pxscale**, **_pyfmt**, **_pylabel**, **_pypmax**, **_pyscale**, **_pysci**, **_setpage**, **_txtfilt**

See also

xy, **logy**, **loglog**

Purpose

Graphs X vs. Y using log coordinates for the Y axis.

Library

pgraph

Format

logy(x,y**);**

Input

x	Nx1 or NxM matrix. Each column represents the X values for a particular line.
y	Nx1 or NxM matrix. Each column represents the Y values for a particular line.

Source

plogy.src

Globals

rerun, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_pascx**, **_paxes**, **_paxht**, **_paxnum**, **_pbox**, **_pcartx**, **_pcarty**, **_pcolor**, **_pcrop**, **_pcross**, **_pcsel**, **_pcwin**, **_pdate**, **_perrbar**, **_pfonts**, **_pgrid**, **_plctrl**, **_plegctl**, **_plegstr**, **_pline**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_plwidth**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_pqgtype**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pworld**, **_pwrscal**, **_pxfmt**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pxsci**, **_pylabel**, **_pyscale**, **_setpage**, **_txtfilt**

See also

xy, **logx**, **loglog**

Purpose

Create a window of specific size and position and add it to the list of windows.

Library

`pgraph`

Format

makewind(*xsize*,*ysize*,*xshft*,*yshft*,*typ***);**

Input

xsize	scalar, horizontal size of the window in inches.
ysize	scalar, vertical size of the window in inches.
xshft	scalar, horizontal distance from left edge of screen in inches.
yshft	scalar, vertical distance from bottom edge of screen in inches.
typ	scalar, window attribute type. If this value is 1, the windows will be transparent. If 0, the windows 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 window size to page size ratio remains the same. The result of this implementation automates the rotation and eliminates the required window recalculations by the user.

See the **window** command for creating tiled windows. For more information on using windows see Section 16.4.

Source

`pwindow.src`

Globals

_pagedim, **_pwindmx**

See also

window, **endwind**, **setwind**, **getwind**, **begwind**, **nextwind**

Purpose

Sets the margins for the current graph window.

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 window 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 screen of 9 x 6.855 inches. If this procedure is used with a window, 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

Globals

_pmargin

See also

axmargin

Purpose

Set the current window to the next available window.

Library

pgraph

Format

nextwind;

Remarks

This function selects the next available window to be the current window. This is the window in which the next graph will be drawn.

See the discussion on using graphics windows in Section 16.4.

Source

pwindow.src

Globals

getwind, **setwind**

See also

endwind, **begwind**, **setwind**, **getwind**, **makewind**, **window**

Purpose

Graph data using polar coordinates.

Library

pgraph

Format

polar(*radius*,*theta*);

Input

radius	Nx1 or NxM matrix. Each column contains the magnitude for a particular line.
theta	Nx1 or NxM matrix. Each column represents the angle values for a particular line.

Source

polar.src

Globals

rerun, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_pascx**, **_pascy**, **_paxes**, **_paxht**, **_paxnum**, **_pbox**, **_pcartx**, **_pcarty**, **_pcolor**, **_pcrop**, **_pcsel**, **_pcwin**, **_pdate**, **_perrbar**, **_pfonts**, **_pgrid**, **_plctrl**, **_plegctl**, **_plegstr**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_plwidth**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_pqgtype**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pworld**, **_pwrscal**, **_pxfmt**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pyfmt**, **_pylabel**, **_pypmax**, **_pyscale**, **_setpage**, **_txtfilt**

See also

xy, **logx**, **logy**, **loglog**, **scale**, **xtics**, **ytics**

Purpose

Displays the most recently created graphics file.

Library

pgraph

Format

rerun;

Remarks

rerun is used by the **endwind** function.

Under DOS, **rerun** invokes the graphics utility `pqgrun.exe`. **pqgrun** is documented in the DOS supplement.

Source

pcart.src

Globals

_pcmdlin, **_pnotify**, **_psilent**, **_ptek**, **_pzoom**

Purpose

Save the current window configuration to a file.

Library

pgraph

Format

err = **savewind(***filename***);**

Input

filename	Name of file.

Output

err	scalar, 0 if successful, 1 if window matrix is invalid. Note that the file is written in either case.

Remarks

See the discussion on using graphics windows in Section 16.4.

Source

pwindow.src

Globals

_pwindmx

See also

loadwind

Purpose

Fix 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

Globals

_pworld, **_pxscale**, **_pyscale**

See also

xtics, **ytics**, **ztics**, **scale3d**

Purpose

Fix 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

Globals

_pworld, **_pxscale**, **_pyscale**, **_pzscale**

See also

scale, **xtics**, **ytics**, **ztics**

Purpose

Set the current window to a previously created window number.

Library

`pgraph`

Format

setwind(n**);**

Input

n	scalar, window number.

Remarks

This function selects the specified window to be the current window. This is the window in which the next graph will be drawn.

See the discussion on using graphics windows in Section 16.4.

Source

`pwindow.src`

Globals

_pwindmx, **_pwindno**

See also

begwind, **endwind**, **getwind**, **nextwind**, **makewind**, **window**

Purpose

To graph a 3-D surface.

Library

pgraph

Format

surface(x**,**y**,**z**);**

Input

x	1xK vector, the X axis data.
y	Nx1 vector, the Y axis data.
z	NxK matrix, the matrix of height data to be plotted.

Global Input

_psurf 2x1 vector, controls 3-D surface characteristics.

[1] if 1, show hidden lines. Default 0.

[2] Color for base (default 7). The base is an outline of the X-Y plane with a line connecting each corner to the surface. If 0, no base is drawn.

_pticout scalar, if 0 (default), tick marks point inward, if 1, tick marks point outward.

_pzclr Z level color control.

There are 3 ways to set colors for the Z levels of a surface graph.

1. To specify a single color for the entire surface plot, set the color control variable to a scalar value 1–15. 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 reset the first element to the minimum z value as follows:

```
_pzclr = { 0.0   1,
           0.0  10,
           0.2  15 };
_pzclr[1,1] = minc(minc(z));
```

Source

psurface.src

Globals

rerun, **_cmnfilt**, **_decreas**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_paxes**, **_paxht**, **_paxnum**, **_pbox**, **_pcartx**, **_pcarty**, **_pcartz**, **_pcrop**, **_pcwin**, **_pdate**, **_pfonts**, **_plegctl**, **_plegstr**, **_pline3d**, **_plotshf**, **_plotsiz**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_poptic**, **_pqgtype**, **_prngerr**, **_protate**, **_pscreen**, **_psurf**, **_psym3d**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pview**, **_pvolume**, **_pworld**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pylabel**, **_pypmax**, **_pyscale**, **_pzclr**, **_pzlabel**, **_pzpmax**, **_pzscale**, **_range**, **_setpage**, **_txtfilt**

See also

volume, **view**

Purpose

To set 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. For example,

```
title("First title line\L Second title line\L Third title line");
```

Fonts may be specified in the title string. See Section 16.6 for instructions on using fonts.

Source

pgraph.src

Globals

_ptitle

See also

xlabel, **ylabel**, **fonts**

Purpose

To set 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`

Globals

_pview

See also

volume, **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

Globals

_pview

See also

volume, **view**

Purpose

To set 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

Globals

_pvolume

See also

view

Purpose

Partition the screen into tiled windows of equal size.

Library

pgraph

Format

window(*row*,*col*,*typ***);**

Input

row	scalar, number of rows of windows.
col	scalar, number of columns of windows.
typ	scalar, window attribute type. If 1, the windows will be transparent, if 0, the windows will be nontransparent (blanked).

Remarks

The windows will be numbered from 1 to $(row)\times(col)$ starting from the left topmost window and moving right.

See the **makewind** command for creating windows of a specific size and position. For more information see Section 16.4.

Source

pwindow.src

Globals

makewind, **_pagedim**

See also

endwind, **begwind**, **setwind**, **nextwind**, **getwind**, **makewind**

Purpose

To set a label for the X axis.

Library

pgraph

Format

xlabel(*str***);**

Input

str	string, the label for the X axis.

Source

pgraph.src

Globals

_pxlabel

See also

title, **ylabel**, **zlabel**

Purpose

To set and fix 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

Globals

_pxscale

See also

scale, **ytics**, **ztics**

Purpose

Graphs X vs. Y using Cartesian coordinates.

Library

pgraph

Format

xy(x,y**);**

Input

x	Nx1 or NxM matrix. Each column contains the X values for a particular line.
y	Nx1 or NxM matrix. Each column contains the Y values for a particular line.

Remarks

Missing values are ignored when plotting symbols. If missing values are encountered while plotting a curve, the curve will end and a new curve will begin plotting at the next non-missing value.

Source

pxy.src

Globals

rerun, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_pascx**, **_pascy**, **_paxes**, **_paxht**, **_paxnum**, **_pbox**, **_pcartx**, **_pcarty**, **_pcolor**, **_pcrop**, **_pcross**, **_pcsel**, **_pcwin**, **_pdate**, **_perrbar**, **_pfonts**, **_pgrid**, **_plctrl**, **_plegctl**, **_plegstr**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_plwidth**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_pqgtype**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pworld**, **_pwrscal**, **_pxfmt**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pxsci**, **_pyfmt**, **_pylabel**, **_pypmax**, **_pyscale**, **_pysci**, **_setpage**, **_txtfilt**

See also

xyz, **logx**, **logy**, **loglog**

Purpose

Graphs X vs. Y vs. Z using Cartesian coordinates.

Library

pgraph

Format

xyz(x,y,z**);**

Input

x	Nx1 or NxK matrix. Each column contains the X values for a particular line.
y	Nx1 or NxK matrix. Each column contains the Y values for a particular line.
z	Nx1 or NxK matrix. Each column contains the Z values for a particular line.

Remarks

Missing values are ignored when plotting symbols. If missing values are encountered while plotting a curve, the curve will end and a new curve will begin plotting at the next non-missing value.

Source

pxyz.src

Globals

rerun, **_cmnfilt**, **_fontsiz**, **_linfilt**, **_pageshf**, **_pagesiz**, **_pappend**, **_parrow3**, **_paxes**, **_paxht**, **_paxnum**, **_pbox**, **_pcartx**, **_pcarty**, **_pcartz**, **_pcolor**, **_pcrop**, **_pcsel**, **_pcwin**, **_pdate**, **_pfonts**, **_plctrl**, **_plegctl**, **_plegstr**, **_pline3d**, **_plotshf**, **_plotsiz**, **_plsel**, **_pltype**, **_plwidth**, **_pmargin**, **_pmsgctl**, **_pncwin**, **_pnotify**, **_pnum**, **_pnumht**, **_poptic**, **_pqgtype**, **_prngerr**, **_protate**, **_pscreen**, **_pssel**, **_pstype**, **_psym3d**, **_psymsiz**, **_ptek**, **_pticout**, **_ptitle**, **_ptitlht**, **_pview**, **_pvolume**, **_pworld**, **_pxlabel**, **_pxpmax**, **_pxscale**, **_pylabel**, **_pypmax**, **_pyscale**, **_pzlabel**, **_pzpmax**, **_pzscale**, **_range**, **_setpage**, **_txtfilt**

See also

xy, **surface**, **volume**, **view**

Purpose

To set a label for the Y axis.

Library

pgraph

Format

ylabel(*str***);**

Input

str	string, the label for the Y axis.

Source

pgraph.src

Globals

_pylabel

See also

title, **xlabel**, **zlabel**

Purpose

To set and fix 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

Globals

_pyscale

See also

scale, **xtics**, **ztics**

Purpose

To set a label for the Z axis.

Library

pgraph

Format

zlabel(*str***);**

Input

str	string, the label for the Z axis.

Source

pgraph.src

Globals

_pzlabel

See also

title, **xlabel**, **ylabel**

Purpose

To set and fix 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

Globals

_pzscale

See also

scale3d, **xtics**, **ytics**, **contour**

Chapter 18

Time and Date

GAUSS offers a comprehensive set of time and date functions. These functions afford the user the ability to return the current time and date, to carry out most related calculations and format the results for output. **GAUSS** also allows the user to perform timed iterations.

In the year 1 AD the calendar in general use was the Julian calendar. The Gregorian calendar that we use today was not invented until the late 1500's. This new calendar changed the method of calculating leap years on century marks. With the Julian system simply every fourth year was a leap year. The Gregorian system made every fourth year a leap year with the exception of century marks which are only leap years if divisible by 400. The British adoption of this calendar, which the GAUSS date functions are based on, did not happen until the year 1752. In that year eleven days were removed; September 2, 1752 was followed by September 14, 1752.

dtvnormal and **utctodtv** are accurate back to 1 AD. The rest of the GAUSS date functions assume a normal Gregorian system regardless of year. Thus, they will not account for the days taken out in September of 1752, nor will they account for all century marks being leap years before the adoption of the Gregorian system in 1752.

The time is given by your operating system, daylight savings time is not automatically accounted for by **GAUSS** in calculations.

18.1 Time and Date Formats

The Time and Date formats in **GAUSS** fall into one of two major categories, matrix/vector and string. The matrix/vector formats can be used for either

calculations or if desired for output. The string formats are, however, mostly for use as ouput. Some manipulation of strings is possible with the use of the **stof** function.

A 4×1 vector is returned by both the **date** and **time** functions.

```
d = date;
d;

1997.00    /* Year */
5.00000    /* Month */
29.0000    /* Day */
56.4700    /* Hundredths of a second since midnight */

t = time;
t;

10.00    /* Hours since midnight */
17.00    /* Minutes */
33.00    /* Seconds */
13.81    /* Hundredths of a second */
```

These vectors can be written to a string of the desired form by passing them through the corresponding function.

```
d = { 1997, 5, 29, 56.47 };
datestr(d);

5/29/97

datestrymd(d);

19970529


t = { 10, 17, 33, 13.81 };
timestr(t);

10:17:33
```

A list and brief description of these, and other related functions is provided in the table in section 18.2.

Another major matrix/vector format is the dtv, or date and time vector. The dtv vector is a 1×8 vector used with the **dtvnormal** and **utctodtv** functions. The format for the dtv vector is:

Year	*Month*	*Day*	*Hour*	*Min*	*Sec*	*DoW*	*DiY*
1955	4	21	4	16	0	4	110

Where:

Year	Year, four digit integer.
Month	1-12, Month in year.
Day	1-31, Day of month.
Hour	0-23, Hours since midnight.
Min	0-59, Minutes.
Sec	0-59, Seconds.
DoW	0-6, Day of week, 0=Sunday.
DiY	0-365, Days since Jan 1 of current year.

dtvnormal normalizes a date. The last two elements are ignored for input, as shown in the following example. They are set to the correct values on output. The input can be 1×8 or N×8.

```
dtv = { 1954 3 17 4 16 0 0 0 };
dtv = dtvnormal(dtv);

  1954   3   17   4   16   0   3   75

dtv[3] = dtv[3] + 400;
print dtv;

  1954   3  417   4   16   0   3   75

dtv = dtvnormal(dtv);
print dtv;

  1955   4   21   4   16   0   4  110
```

18.2 Time and Date Functions

Following is a partial listing of the time and date functions available in **GAUSS**.

datestr	Formats a Date vector to a string (mo/dy/yr).
datestrymd	Formats a Date vector to an eight character string of the type yyyymmdd.
dayinyr	Returns day number in the year of a given date.
_daypryr	Returns the number of days in the years given as input.
dtvnormal	Normalizes a 1x8 dtv vector.
etdays	Computes the difference in days between two dates.

ethsec	Computes the difference between two times in hundredths of a second.
etstr	Formats a time difference measured in hundreths of a second to a string.
_isleap	Returns a vector of ones and zeros, 1 if leap year 0 if not.
timestr	Formats a Time vector to a string hr:mn:sc.
timeutc	Universal time coordinate, number of seconds since January 1, 1970 Greenwich Mean Time.
utctodtv	Converts a scalar, number of seconds since, or before, Jan 1 1970 Greenwich mean time, to a dtv vector.

Below is an example of two ways to calculate a time difference.

```
d1 = { 1996, 12, 19, 82 };
d2 = { 1997, 4, 28, 4248879.3 };
dif = ethsec(d1,d2);
ds = etstr(dif);
```

$dif =$ 1.1274488$e+09$

$ds =$ 130$days$ 11$hours$ 48$minutes$ 7.97$seconds$

If only the number of days is needed use **etdays**.

```
d1 = { 1996, 12, 19, 82 };
d2 = { 1997, 4, 28, 4248879.3 };
dif = etdays(d1,d2);
```

$dif =$ 130.00000

The last element of *d1* is optional when used as an input for **etdays**.

_isleap returns a matrix of ones and zeros, ones when the corresponding year is a leap year.

```
x = seqa(1970,1,20);
y = _isleap(x);
delif(x,abs(y-1));

1972.0000  /* Vector containing all leap years
1976.0000     between 1970 - 1989 */
1980.0000
1984.0000
1988.0000
```

To calculate the days of a number of consecutive years:

```
x = seqa(1983,1,3);
y = _daypryr(x);
sumc(y);

1096.0000
```

To add a portion of the following year:

```
g = { 1986, 2, 23, 0 };
dy = dayinyr(g);
sumc(y)+dy;

1150.0000
```

For more information on any of these functions see their respective pages in the command reference.

18.2.1 Timed Iterations

Iterations of a program can be timed with the use of the **hsec** function in the following manner.

```
et = hsec;          /* Start timer */

/* Segment of code to be timed */

et = (hsec-et)/100;  /* Stop timer, convert to seconds */
```

A specific example is located in the tutorial section of your supplement.

In the case of a program running from one day into the next you would need to replace the **hsec** function with the **date** function. The **ethsec** function should be used to compute the time difference; a straight subtraction as in the previous example will not give the desired result.

```
dstart = date;              /* Start timer */

/* Segment of code to be timed */

dend = date;                /* Stop timer */

dif = ethsec(dstart,dend)/100;  /* Convert time difference to seconds */
```

Chapter 19

Utility: atog

atog is a utility program to convert ASCII files into **GAUSS** data files. **atog** can convert delimited and packed ASCII files into **GAUSS** data sets.

From DOS:

atog *cmdfile*

From **GAUSS**:

atog *cmdfile*;

In the examples above, *cmdfile* is the filename of the command file. If no extension is given, `.cmd` will be assumed. If no command file is specified, you will be asked for one. Pressing **Enter** at that point begins the interactive mode. Interactive mode allows the user to create a command file interactively or specify information for the data conversion directly. The Unix version of **atog** does not support interactive mode.

19.1 Command Summary

The following commands are supported with **atog**:

append Append data to an existing file.

complex Treat data as complex variables.

help Display command summary.

input The name of the ASCII input file.

invar Input file variables (column names).

msym Specify missing value character.

nocheck Don't check data type or record length.

nodata Modify the header file only.

output The name of the **GAUSS** data set to be created.

outtyp Output data type.

outvar List of variables to be included in output file.

quit Exit without converting. Valid only in interactive mode.

run Begin execution—cease keyboard input. Valid only in interactive mode.

save Controls saving of command file while in interactive mode.

The principle commands for converting an ASCII file that is delimited with spaces or commas are given in the following simple example. These commands are placed in a command file or typed directly into **atog**'s interactive mode.

```
input agex.asc;
output agex;
invar $ race # age pay $ sex region;
outvar region age sex pay;
outtyp d;
```

In this example a delimited ASCII file `agex.asc` is converted to a double precision **GAUSS** data file `agex.dat`. A header is created which contains the names of the variables and other system information. The input file has 5 variables. The file will be interpreted as having 5 columns:

column	name	data type
1	race	character
2	AGE	numeric
3	PAY	numeric
4	sex	character
5	region	character

The output file will have 4 columns since the first column of the input file (race) is not included in the output variables. The columns of the output file are:

column	name	data type
1	region	character
2	AGE	numeric
3	sex	character
4	PAY	numeric

When using the v89 format (see **create** in the *COMMAND REFERENCE*) the header information is stored in a separate `.dht` file, thus `agex.dht`. The v92 and v96 formats store header information at the top of the `.dat` file.

If the variable is explicitly designated as a character variable, its name will be stored in lowercase. The $ in the **invar** statement specifies that the variables that follow are character type. The # specifies numeric. If $ or # are not used in an **invar** statement, the default is numeric.

Comments in command files must be enclosed between '@' characters.

19.2 Commands

A detailed explanation of each command follows.

- **append**

Instructs **atog** to append the converted data to an existing data file.

```
append;
```

No assumptions are made regarding the format of the existing file. You should make certain that the number, order and type of data converted match the existing file.

- **complex**

Instructs **atog** to convert the ASCII file into a complex **GAUSS** data set.

```
complex;
```

Complex **GAUSS** data sets are stored by rows, with the real and imaginary parts interleaved, element by element. **atog** assumes the same structure for the ASCII input file, and will thus read TWO numbers out for EACH variable specified. **complex** cannot be used with packed ASCII files.

- **help**

Display help screen in interactive mode.

```
help;
```

- **input**

The **input** command specifies the file name of the ASCII file which is to be converted. The full path name can be used in the file specification. For example, the command:

```
input data.raw;
```

will expect an ASCII data file in the current working directory. The command:

```
input /research/data/myfile.asc;
```

specifies a file to be located in the `/research/data` subdirectory.

- **invar**

Soft Delimited ASCII Files

Soft delimited files may have spaces, commas, or cr/lf as delimiters between elements. Two or more consecutive delimiters with no data between them are treated as one delimiter.

```
invar age $ name sex # pay var[1:10] x[005];
```

The **invar** command above specifies the following variables:

column	name	data type
1	AGE	numeric
2	name	character
3	sex	character
4	PAY	numeric
5	VAR01	numeric
6	VAR02	numeric
7	VAR03	numeric
8	VAR04	numeric
9	VAR05	numeric
10	VAR06	numeric
11	VAR07	numeric
12	VAR08	numeric
13	VAR09	numeric
14	VAR10	numeric
15	X001	numeric
16	X002	numeric
17	X003	numeric
18	X004	numeric
19	X005	numeric

As the input file is translated, the first 19 elements will be interpreted as the first row (observation), the next 19 will be interpreted as the second row, and so on. If the number of elements in the file is not evenly divisible by 19, the final incomplete row will be dropped and a warning message will be given.

Hard Delimited ASCII Files

Hard delimited files have a printable character as a delimiter between elements. Two delimiters without intervening data between them will be interpreted as a missing. If **\n** is specified as a delimiter, the file should have one element per line and blank lines will be considered missings. Otherwise, delimiters must be printable characters. The dot '.' is illegal and will always be interpreted as a missing value. To specify the backslash as a delimiter, use \\. If **\r** is specified as a delimiter, the file will be assumed to contain one case or record per line with commas between elements and no comma at the end of the line.

For hard delimited files the **delimit** subcommand is used with the **invar** command. The **delimit** subcommand has two optional parameters. The first parameter is the delimiter. The default is a comma. The second parameter is an 'N'. If the second parameter is present, **atog** will expect N delimiters. If it is not present, **atog** will expect N-1 delimiters.

```
invar delimit(, N) $ name # var[5];
```

would expect a file like this:

```
BILL ,  222.3,  123.2,  456.4,  345.2,  533.2,
STEVE,  624.3,  340.3,       ,  624.3,  639.5,
TOM  ,  244.2,  834.3,  602.3,  333.4,  822.5,
```

```
invar delimit(,) $ name # var[5];
```

or

```
invar delimit $ name # var[5];
```

would expect a file like this:

```
BILL ,  222.3,  123.2,  456.4,  345.2,  533.2,
STEVE,  624.3,  340.3,       ,  624.3,  639.5,
TOM  ,  244.2,  834.3,  602.3,  333.4,  822.5
```

The difference between specifying N or N-1 delimiters can be seen in the following example:

```
456.4,  345.2,  533.2,
     ,  624.3,  639.5,
602.3,  333.4,
```

If the **invar** statement specified 3 variables and N-1 delimiters were specified, this file would be interpreted as having three rows containing a missing in the 2,1 element and the 3,3 element like this:

456.4 345.2 533.2
missing 624.3 639.5
602.3 333.4 *missing*

If N delimiters were specified, this file would be interpreted as having two rows, and a final incomplete row which is dropped:

456.4 345.2 533.2
missing 624.3 639.5

The spaces are shown only for clarity and are not significant in delimited files so:

```
BILL,222.3,123.2,456.4,345.2,533.2,
STEVE,624.3,340.3,,624.3,639.5,
TOM,244.2,834.3,602.3,333.4,822.5
```

would work just as well. Linefeeds are significant only if **\n** is specified as the delimiter or when using **\r**.

invar delimit(\r) $ name # var[5];

would expect a file with no comma after the final element in each row:

```
BILL ,  222.3,  123.2,  456.4,  345.2,  533.2
STEVE,  624.3,  340.3,  245.3,  624.3,  639.5
TOM  ,  244.2,  834.3,  602.3,  333.4,  822.5
```

Packed ASCII Files

Packed ASCII files must have fixed length records. The **record** subcommand is used to specify the record length and variables are specified by giving their type, starting position, length, and the position of an implicit decimal point if necessary.

Note that **outvar** is not used with packed ASCII files. Instead, **invar** is used to specify only those variables to be included in the output file.

For packed ASCII files the syntax of the **invar** command is as follows:

invar record=*reclen (format) variables (format) variables***;**

where,

reclen the total record length in bytes including the final carriage return/line feed if applicable. Records must be fixed length.

format (*start,length.prec*) where:

- **start** starting position of the field in the record, 1 is the first position. The default is 1.
- **length** the length of the field in bytes. The default is 8.
- **prec** optional, a decimal point will be inserted automatically **prec** places in from the RIGHT edge of the field.

If several variables are listed after a format definition, each succeeding field will be assumed to start immediately after the preceding field. If an asterisk is used to specify the starting position, the current logical default will be assumed. An asterisk in the length position will select the current default for both length and prec. This is illegal: (3,8.*).

The type change characters $ and # are used to toggle between character and numeric data type.

Any data in the record that is not defined in a format is ignored.

The examples below assume a 32-byte record with a carriage return/line feed occupying the last 2 bytes of each record. The data below can be interpreted in different ways using different **invar** statements:

```
          ABCDEFGHIJ12345678901234567890<CR><LF>
          |         |         |         |  |   |
position 1        10        20        30 31  32
```

invar record=32 $(1,3) group dept #(11,4.2) x[3] (*,5) y;

variable	value	type
group	ABC	character
dept	DEF	character
X1	12.34	numeric
X2	56.78	numeric
X3	90.12	numeric
Y	34567	numeric

invar record=32 $ dept (*,2) id # (*,5) wage (*,2) area

variable	value	type
dept	ABCDEFGH	character
id	IJ	character
WAGE	12345	numeric
AREA	67	numeric

- **msym**

Specifies the character in the input file that is to be interpreted as a missing value.

```
msym &;
```

The statement above defines the character '&' as the missing value character. The default '.' (dot) will always be interpreted as a missing value unless it is part of a numeric value.

- **nocheck**

Optional, suppresses automatic checking of packed ASCII record length and output data type. The default is to increase the record length by 2 bytes if the second record in a packed file starts with cr/lf, and any files that have explicitly defined character data will be output in double precision regardless of the type specified.

- **nodata**

If specified, the `.dat` file which contains the data will not be modified. **nodata** is used to change the variable names or to reshape the file. Since the data is not modified, the kind of reshaping that is possible is limited. This is usually used to change the names of the variables or to reshape a file that has one variable to have several variables and vice versa.

```
output cvx;
invar x;
outtyp d;
nodata;
```

- **output**

The name of the **GAUSS** data set. Two files will be created with the extensions `.dat` and `.dht` for the data and header file pair. The full DOS file path is supported. For example:

```
output /gauss/dat/test;
```

creates the files `test.dat` and `test.dht` on the `/gauss/dat` directory.

- **outtyp**

Selects the numerical accuracy of the output file. Use of this command should be dictated by the accuracy of the input data and storage space limitations. The format is:

outtyp *fmt*;

where *fmt* is:

D or 8	double precision
F or 4	single precision (default)
I or 2	integer

The ranges of the different formats are:

bytes	data type	significant digits	range
2	integer	4	$-32768 \leq X \leq 32767$
4	single precision	6–7	$8.43\text{x}10^{-37} \leq \|X\| \leq 3.37\text{x}10^{+38}$
8	double precision	15–16	$4.19\text{x}10^{-307} \leq \|X\| \leq 1.67\text{x}10^{+308}$

If the output type is integer, the input numbers will be truncated to integers. If your data has more than 6 or 7 significant digits, you should specify **outtyp** as double.

Character data require **outtyp d**. **atog** automatically selects double precision when character data is specified in the invar statement unless you have specified **nocheck**.

The precision of the storage selected does not affect the accuracy of **GAUSS** calculations using the data. **GAUSS** converts all data to double precision when the file is read.

- **outvar**

To select the variables to be placed in the **GAUSS** data set. The **outvar** command needs only the list of variables to be included in the output data set. **outvar** is not used with packed ASCII files. They can be in any order.

```
invar $name #age pay $sex #var[1:10] x[005];
outvar sex age x001 x003 var[1:8];
```

column	name	data type
1	sex	character
2	AGE	numeric
3	X001	numeric
4	X003	numeric
5	VAR01	numeric
6	VAR02	numeric
7	VAR03	numeric
8	VAR04	numeric
9	VAR05	numeric
10	VAR06	numeric
11	VAR07	numeric
12	VAR08	numeric

outvar is not used with packed ASCII files.

- **preservecase**

Optional, preserves the case of variable names. The default is **nopreservcase** which will force variable names for numeric variables to upper case and character variables to lower case.

- **quit**

To exit **atog** without executing. Used in **atog** interactive mode only.

```
quit;
```

- **run**

Used in **atog** interactive mode only. This command starts execution of a command set that has been specified in interactive mode.

```
run;
```

- **save**

Controls saving of interactive commands in a file.

```
save qdata;
```

This command will open a file `qdata.cmd`. As subsequent commands are typed in, they will be saved in this file.

```
save +qdata;
```

This will open the file `qdata.cmd` and process any commands already in the file. Any subsequent commands that are typed in will be appended to the file.

```
save -;
```

This will close the file, and stop saving commands.

If no file name is specified, the default name is `atog.cmd`.

19.3 Examples

The first example is a soft delimited ASCII file called `agex1.asc`. The file contains seven columns of ASCII data.

```
Jan 167.3 822.4 6.34E06 yes 84.3 100.4
Feb 165.8 987.3 5.63E06 no 22.4 65.6
Mar 165.3 842.3 7.34E06 yes 65.4 78.3
```

The **atog** command file `agex1.cmd`:

```
input /gauss/agex1.asc;
output agex1;
invar $month #temp pres vol $true var[02];
outvar month true temp pres vol;
```

The output data set will contain the following information:

name	month	true	TEMP	PRES	VOL
case 1	Jan	yes	167.3	822.4	6.34e+6
case 2	Feb	no	165.8	987.3	5.63e+6
case 3	Mar	yes	165.3	842.3	7.34e+6
type	char	char	numeric	numeric	numeric

The data set is double precision since character data is explicitly specified.

The second example is a packed ASCII file `xlod.asc` which contains 32-character records.

```
          AEGDRFCSTy023456789606315678900<CR><LF>
          EDJTAJPSTn123958639980648395610<CR><LF>
          GWDNADMSTy198278456597252344510<CR><LF>
          |         |          |          |  |   |
position 1        10         20         30 31  32
```

The **atog** command file `xlod.cmd`:

```
input /gauss/dat/xlod.asc;
output xlod2;
invar record=32 $(1,3) client[2] zone (*,1) reg #(20,5) zip;
```

The output data set will contain the following information:

name	client1	client2	zone	reg	ZIP
case 1	AEG	DRF	CST	y	60631
case 2	EDJ	TAJ	PST	n	98064
case 3	GWD	NAD	MST	y	59725
type	char	char	char	char	numeric

The data set is double precision since character data is explicitly specified.

The third example is a hard delimited ASCII file called `cplx.asc`. The file contains six columns of ASCII data.

```
 456.4,  345.2,  533.2, -345.5,  524.5,  935.3,
-257.6,  624.3,  639.5,  826.5,  331.4,  376.4,
 602.3, -333.4,  342.1,  816.7, -452.6, -690.8
```

The **atog** command file `cplx.cmd`:

```
input /gauss/cplx.asc;
output cplx;
invar delimit #cvar[3];
complex;
```

The output data set will contain the following information:

name	cvar1	cvar2	cvar3
case 1	456.4 + 345.2i	533.2 - 345.5i	524.5 + 935.3i
case 2	-257.6 + 624.3i	639.5 + 826.5i	331.4 + 376.4i
case 3	602.3 - 333.4i	342.1 + 816.7i	-452.6 - 690.8i
type	numeric	numeric	numeric

The data set defaults to single precision since no character data is present, and no **outtyp** command is specified.

19.4 Error Messages

`atog - Can't find input file`

The ASCII input file could not be opened.

1. The file is not there.
2. You are out of file handles. Check the `config.sys` file on the root directory of the disk you boot from to make sure it has a files=25 or greater statement.

`atog - Can't open output file`

The output file could not be opened.

1. The disk or directory is full.
2. You are out of file handles. Check the `config.sys` file on the root directory of the disk you boot from to make sure it has a files=25 or greater statement.

`atog - Can't open temporary file`

Notify Aptech Systems of the circumstances of this error.

`atog - Can't read temporary file`

Notify Aptech Systems of the circumstances of this error.

```
atog - Character data in output file
Setting output file to double precision
```

The output file contains character data. The type was set to double precision automatically.

`atog - Character data longer than 8 bytes were truncated`

The input file contained character elements longer than 8 bytes. The conversion continued and the character elements were truncated to 8 bytes.

`atog - Disk Full`

The output disk is full. The output file is incomplete.

`atog - Found character data in numeric field`

This is a warning that character data was found in a variable that was specified as numeric. The conversion will continue.

`atog - Header file rename unsuccessful`

The temporary header file `atog.$dh` could not be renamed. This could be a disk full problem. `atog.$dh` has been left on the disk and can be renamed manually if you want to recover from the error.

`atog - Illegal command`

An unrecognizable command was found in a command file.

`atog - Internal error`

Notify Aptech Systems of the circumstances of this error.

`atog - Invalid delimiter`

The delimiter following the backslash is not supported.

`atog - Invalid output type`

Output type must be I, F, or D.

`atog - Missing value symbol not found`

No missing value was specified in an **msym** statement.

`atog - No Input file`

No ASCII input file was specified. The **input** command is probably missing.

`atog - No input variables`

No input variable names were specified. The **invar** statement may be missing.

`atog - No output file`

No output file was specified. The **output** command is probably missing.

```
atog - output type d required for character data
Character data in output file will be lost
```

Output file contains character data and is not double precision.

`atog - Open comment`

The command file has a comment that is not closed. Comments must be enclosed in **@**'s.

```
@ comment @
```

`atog - Out of memory`

Notify Aptech Systems of the circumstances of this error.

`atog - read error`

A read error has occurred while converting a packed ASCII file.

`atog - Read error on .dht file`

The temporary header file `atog.$dh` could not be read.

`atog - Record length must be 1-16384 bytes`

The **record** subcommand has an out of range record length.

`atog - Statement too long`

Command file statements must be less than 16384 bytes.

`atog - Syntax error at:`

There is unrecognizable syntax in a command file.

`atog - Too many input variables`

More input variables were specified than available memory permitted.

`atog - Too many output variables`

More output variables were specified than available memory permitted.

`atog - Too many variables`

More variables were specified than available memory permitted.

`atog - Undefined variable`

A variable requested in an **outvar** statement was not listed in an **invar** statement.

`ATOG WARNING: missing ‘)’ at:`

The parentheses in the **delimit** subcommand were not closed.

`ATOG WARNING: some records begin with cr/lf`

A packed ASCII file has some records that begin with a carriage return/linefeed. The record length may be wrong.

`atog - complex illegal for packed ASCII file.`

A **complex** command was encountered following an **invar** command with **record** specified.

`atog - Cannot read packed ASCII. (complex specified)`

An **invar** command with **record** specified was encountered following a **complex** command.

Chapter 20

Utility: liblist

liblist is a library symbol listing utility.

It is a stand-alone program that lists the symbols available to the **GAUSS** autoloading system. **liblist** will also perform `.g` file conversion and listing operations.

Note that `.g` files are specific files once used in older versions of **GAUSS**. Due to compiler efficiency and other reasons, `.g` files are no longer recommended for use. The **liblist** options related to `.g` files are supplied to aid the user in consolidating `.g` files and converting them to the standard `.src` files.

The format for using **liblist** from the operating system is:

liblist -*flags lib1 lib2* **...** *libn*

flags control flags to specify the operation of **liblist**.

G	list all `.g` files in the current directory and along the **src_path**.
D	create `gfile.lst` using all `.g` files in the current directory and along the **src_path**.
C	convert `.g` files to `.src` files and list the files in `srcfile.lst`.
L	list the contents of the specified libraries.
N	list library names.
P	send listing to printer (**lpt1**).
F	use page breaks and form feed characters.
I	assume complex library extensions (`.lcg`).

The search is performed in the following manner:

1. List all symbols available as `.g` files in the current directory and then the **src_path**.

2. List all symbols defined in `.l32` files (`.lcg` files if you included the **l** control flag) in the **lib_path** subdirectory. `gauss.lcg`, if it exists, will be listed last.

20.1 Report Format

The listing produced will go to the standard output. The order the symbols will be listed in is the same order that they will be found by **GAUSS**, except that **liblist** processes the `.lcg` files in the order they appear in the **lib_path** subdirectory, whereas **GAUSS** processes libraries according to the order specified in your **library** statement. **liblist** assumes that all of your libraries are active, i.e., you have listed them all in a **library** statement. `gauss.lcg` will be listed last.

The listing looks like this:

```
       Symbol   Type     File            Library        Path
=================================================================
    1. autoreg  ------   autoreg  .src   auto.lcg       /gauss/src
    2. autoprt  ------   autoreg  .src   auto.lcg       /gauss/src
    3. autoset  ------   autoreg  .src   auto.lcg       /gauss/src
    4. _pticout matrix   pgraph   .dec   pgraph.lcg     /gauss/src
    5. _pzlabel string   pgraph   .dec   pgraph.lcg     /gauss/src
    6. _pzpmax  matrix   pgraph   .dec   pgraph.lcg     /gauss/src
    7. asclabel proc     pgraph   .src   pgraph.lcg     /gauss/src
    8. fonts    proc     pgraph   .src   pgraph.lcg     /gauss/src
    9. graphset proc     pgraph   .src   pgraph.lcg     /gauss/src
   10. _svdtol  matrix   svd      .dec   gauss.lcg      /gauss/src
   12. __maxvec matrix   system   .dec   gauss.lcg      /gauss/src
   13. besselj  proc     bessel   .src   gauss.lcg      /gauss/src
   14. bessely  proc     bessel   .src   gauss.lcg      /gauss/src
   15. xcopy    keyword  dos      .src   gauss.lcg      /gauss/src
   16. atog     keyword  dos      .src   gauss.lcg      /gauss/src
```

Symbol is the symbol name available to the autoloader.

Type is the symbol type. If the library is not strongly typed, this will be a line of dashes.

File is the file the symbol is supposed to be defined in.

Library is the name of the library, if any, the symbol is listed in.

Path is the path the file is located on. If the file cannot be found, the path will be "`*** not found ***`".

20.2 Using liblist

liblist is executed from the operating system or **GAUSS** with:

liblist -*flags lib1 lib2* **...** *libn*

To put the listing in a file called `lib.lst`:

liblist -l > lib.lst

To convert all your `.g` files and list them in `srcfile.lst`:

liblist -c

The numbers are there so you can sort the listing if you want and still tell which symbol would be found first by the autoloader. You may have more than one symbol by the same name in different files. **liblist** can help you keep them organized so you don't inadvertently use the wrong one.

Chapter 21

Categorical Reference

This categorical reference lists those commands that either operate differently under **GAUSS** for UNIX than other platforms, or are supported only under **GAUSS** for UNIX. For a general categorical reference of commands, see chapter **??**.

21.1 Window Control

WinOpenPQG	Open a PQG window.
WinOpenText	Open a Text window.
WinOpenTTY	Open a TTY window.
WinSetActive	Set the active window.
WinGetActive	Get the active window.
WinMove	Move a window.
WinResize	Resize a window.
WinPan	Pan through a window.
WinRefresh	Refresh a window.
WinRefreshArea	Refresh a rectangular area of a Text window.
WinClose	Close a window.
WinCloseAll	Close all windows except Window 1
WinSetTextWrap	Set the text wrap mode for a window.
WinSetRefresh	Set the refresh mode for a window.
WinGetAttributes	Get attributes for a window.

21.2 Console I/O

con	Request input, create matrix.
cons	Request input, create string.
key	Return the next key in the input buffer of the active window. If none are pending, return 0.
keyw	Return the next key in the input buffer of the active window. If none are pending, wait for one.
wait	Wait until a key is pressed in the active window.
waitc	Flush input buffer of the active window, then wait until a key is pressed in it.

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;
    if not key;
        sleep(0.5);
    endif;
endo;
```

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

key and **keyw** have the same return values.

21.3 Output

WinSetCursor	Set the cursor position for a window.
WinGetCursor	Get the cursor position for a window.
WinPrint	Print formatted data to a window.
WinClear	Clear a window.
WinClearArea	Clear a rectangular area of a Text window.
WinClearTTYLog	Clear the output log of a TTY window.
WinPrintPQG	Print PQG graph to file or printer.
WinZoomPQG	Display enlarged area of PQG graph.
WinSetColorCells	Set color cell RGB values for a window.
WinGetColorCells	Get color cell RGB values for a window.
WinSetForeground	Set the foreground color for a window.

WinSetBackground	Set the background color for a window.
WinSetColormap	Select a standard colormap for a window.
FontLoad	Load a system font.
FontUnload	Unload a system font.
FontUnloadAll	Unload all system fonts.
output	Open/close auxiliary output for the active window.
outwidth	Set line width of auxiliary output for the active window.
print	Print formatted data to the active window.
printdos	Print string to the active window (calls C **printf()**), emulate ANSI.SYS commands.
printfm	Print matrix to the active window, use specified format.
print on|off	Turn auto print on/off for the active window.
screen on|off	Direct/suppress **print** statements to the active window.
screen out	Dump snapshot of the active window to its auxiliary output.
plot	Draw text style XY plot in the active window.
plotsym	Set symbol used for **plot** in the active window.
cls	Clear the active window.
font	Set font to be used in the active window.
color	Set foreground, background / colorcells for the active window.
csrcol	Get column position of cursor in the active window.
csrlin	Get line (row) position of cursor in the active window.
csrtype	Sets the cursor shape.
format	Define **print** format for the active window.
locate	Position the cursor in the active window.
tab	Tab the cursor in the active window.
scroll	Scroll a section of the active window.
setvmode	Set the active window to emulate DOS video mode.

The results of all printing may be sent to an output file using **output**. This file can then be printed or ported as an ASCII file to other software. A separate output file can be opened for each window.

locate acts like **tab** in TTY windows. The *row* argument is ignored.

scroll is supported only for Text windows.

21.4 Error Handling and Debugging

errorlog	Send error message to Window 1 and error log file.

21.5 Workspace Management

new	Clear current workspace, close all windows except Window 1.

21.6 Program Control

end	Terminate a program, close all windows except Window 1.
stop	Terminate a program, leave all windows open.
system	Quit **GAUSS** and return to the OS.

Both **stop** and **end** terminate the execution of a program. **end** closes all open windows except for Window 1; **stop** leaves all windows open. Neither **stop** or **end** is required to terminate a **GAUSS** program; if neither is found, an implicit **stop** is executed.

Chapter 22

Command Reference

The command reference is broken into two sections. This first section details commands that operate differently under **GAUSS** for UNIX and **GAUSS** for other operating systems. The second section defines commands that are supported only under **GAUSS** for UNIX.

---

cls

cls clears the active window. For Text windows, this means the window buffer is cleared to the background color. For TTY windows, the current output line is panned to the top of the window, effectively clearing the display. The output log is still intact. To clear the output log of a TTY window, use **WinClearTTYLog**. For PQG windows, the window is cleared to the background color, and the graphics data are discarded.

---

color

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

---

con — **con** gets input from the active window. If you are working in terminal mode, **GAUSS** will not "see" any input until you press **Enter**, so follow each entry with an **Enter**.

---

cons — **cons** gets input from the active window. If you are working in terminal mode, **GAUSS** will not "see" your input until you press **Enter**.

---

csrcol — **csrcol** returns the cursor column for the active window. For Text windows, this value is the cursor column with respect to the text buffer. For TTY windows, this value is the cursor column with respect to the current output line. For PQG windows, this value is meaningless.

---

csrlin — **csrlin** returns the cursor line for the active window. For Text windows, this value is the cursor row with respect to the text buffer. For TTY windows, this value is the current output line number (i.e., the number of lines logged + 1). For PQG windows, this value is meaningless.

---

end — **end** closes all windows except window 1. If you want to leave a program's windows open, use **stop**.

---

errorlog — **errorlog** prints to window 1 and the error log file.

---

format — **format** sets the **print** statement format for the active window. Each window "remembers" its own format, even when it is no longer the active window.

---

key — **key** gets input from the active window. If you are working in terminal mode, **key** doesn't "see" any keystrokes until **Enter** is pressed.

keyw	**keyw** gets the next key from the input buffer of the active window. If the input buffer is empty, **keyw** waits for a keystroke. If you are working in UNIX terminal mode, **GAUSS** will not see any input until you press the **Enter** key.
locate	**locate** locates the cursor in the active window. The row argument is ignored for window 1 and TTY windows, making **locate** act like **tab** in those windows.
new	**new** closes all open windows except window 1.
output	**output** commands affect the active window. Each window "remembers" its own settings, even when it is no longer the active window. Each window can have its own auxiliary output file, or several windows can write to the same output file.
outwidth	**outwidth** affects the active window. Each window "remembers" its own setting, even when it is no longer the active window.
plot	**plot** writes to the active window. Supported only for Text windows.
print	**print** writes to the active window. The refresh setting of the window determines when output is displayed. To force immediate display, use the **/flush** flag: **print /flush** *expr1 expr2* **...;;**

print on|off	**print on|off** affects the active window. Each window "remembers" its own setting, even when it is no longer the active window.
printdos	**printdos** writes to the active window. The refresh setting of the window determines when output is displayed. To force immediate display, use the **WinRefresh** command.
printfm	**printfm** writes to the active window. The refresh setting of the window determines when output is displayed. To force immediate display, use the **WinRefresh** command.
screen on|off	**screen on|off** affects the active window. Each window "remembers" its own setting, even when it is no longer the active window.
screen out	**screen out** affects the active window. Not supported for PQG windows.
scroll	**scroll** scrolls a region of the active window. Supported only for Text windows.
setvmode	**setvmode** affects the active window. This command forces the active window into a DOS emulation mode, and is supported for backwards compatibility only. Window size, resolution, font and/or colormap may be changed to accomodate the requested video mode. Not supported for window 1 and TTY windows.
stop	**stop** does not close any windows. If you want to close a

program's windows when it is done, use **end**.

---

tab — **tab** tabs the cursor in the active window. **tab** is not supported in terminal mode.

---

wait — **wait** waits for a keystroke in the active window. If you are working in terminal mode, **wait** doesn't "see" any keystrokes until **Enter** is pressed.

---

waitc — **waitc** flushes the input buffer of the active window, then waits for a new keystroke in the active window. If you are working in terminal mode, **waitc** doesn't "see" any keystrokes until **Enter** is pressed.

The rest of the command reference defines the commands that are supported only under **GAUSS** for UNIX.

Purpose

Set the font for the active window.

Format

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

Input

fonthandle scalar, font handle or −1.

Output

oldfonth scalar, handle of previous font.

Remarks

font sets the font for the active window to *fonthandle* and returns the handle of the window's previous font in *oldfonth*. If $fonthandle = -1$, **font** returns the current font handle without changing it.

font is not supported for PQG windows.

Purpose

Load a font.

Format

fonthandle = **FontLoad(***fontspec***);**

Input

fontspec string, font specification.

Output

fonthandle scalar, font handle.

Remarks

FontLoad loads the X11 font specified in *fontspec*, where *fontspec* is a standard X11 font name, and may contain the wildcard characters '∗' and '?'.

If the font is successfully loaded, **FontLoad** returns a **GAUSS** font handle which is used in other routines (e.g., **font**, **WinGetAttributes**) to refer to the font.

If the font requested cannot be loaded, **FontLoad** returns a −1.

Font handles 1 through 4 are reserved by **GAUSS**; font handles returned by **FontLoad** are ≥ 5.

Purpose

Unload a font.

Format

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

Input

fonthandle scalar, font handle.

Output

ret scalar, 0 if call is successful, −1 if it fails.

Remarks

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

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

```
fonthandle = FontUnload(fonthandle);
```

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

Purpose

Unload all currently loaded fonts.

Format

FontUnloadAll;

Remarks

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

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

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

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

Purpose

Clear a window.

Format

WinClear(*whandle***);**

Input

whandle scalar, window handle.

Remarks

WinClear clears the window to the background color. The effect of **WinClear** on window data depends upon the window type, as follows:

If *whandle* is a PQG window, the current graphics data are discarded. The image can only be restored by repeating the sequence of commands that originally generated it.

If *whandle* is a Text window, its text buffer is cleared, destroying all data for the window. The previous contents of the window can only be restored by repeating the sequence of commands that originally generated it.

If *whandle* is a TTY window, the current output line is panned to the top of the window. Window data is not affected.

See also

WinClearTTYLog

Purpose

Clear a rectangular region of a Text window.

Format

WinClearArea(*whandle*,*row*,*col*,*nrows*,*ncols***);**

Input

whandle	scalar, window handle.
row	scalar, the topmost row to clear.
col	scalar, the leftmost column to clear.
nrows	scalar, the number of rows to clear.
ncols	scalar, the number of columns to clear.

Remarks

WinClearArea clears a rectangular region of a Text window to the background color. The region has its top-left corner at (*row*,*col*), and is *nrows*×*ncols* in size.

The corresponding region of the text buffer is cleared, destroying all data for the region. The previous contents of the region can only be restored by repeating the sequence of commands that originally generated it.

WinClearArea is supported only for Text windows.

Purpose

Clear the output log of a TTY window.

Format

WinClearTTYLog(*ttywhandle***);**

Input

ttywhandle scalar, window handle of a TTY window.

Remarks

WinClearTTYLog deletes the output log of a TTY window, destroying the window data. The window is automatically cleared and refreshed.

See also

WinClear

Purpose

Close a window.

Format

ret = **WinClose(***whandle***);**

Input

whandle	scalar, window handle.

Output

ret	scalar 0.

Remarks

WinClose closes (destroys) the window specified by *whandle*. When a window is closed, **GAUSS** destroys all of the internal information required to maintain the window. Any further commands which reference *whandle* will result in an error.

When a window is closed, the window handle should be set to to zero, to let your program know that it no longer refers to a valid window. The easiest way to do this is:

```
wh = WinClose(wh);
```

If you close the active window, you will have no active window. To set a new one, use **WinSetActive**.

Window 1 cannot be closed.

Purpose

Close all windows except window 1.

Format

WinCloseAll;

Remarks

WinCloseAll closes all open windows except window 1.

You should set the affected window handles to zero after a **WinCloseAll**, to let your program know that they no longer refer to valid windows. The easiest way to do this is:

```
clear wh1, wh2 [,wh3...];
```

If window 1 is not the active window, you will have no active window. To set a new one, use **WinSetActive**.

The **end** command also closes all open windows (except window 1). If you want to terminate a program but leave its windows open, use **stop**.

Purpose

Get the active window.

Format

whandle = **WinGetActive;**

Output

whandle scalar, the active window handle.

Remarks

WinGetActive returns the window handle of the active window. If there is no active window, **WinGetActive** returns a −1.

Purpose

Get window attributes.

Format

wattr = **WinGetAttributes(***whandle***);**

Input

whandle scalar, window handle.

Output

wattr 25×1 vector, window attributes, or scalar -1 if error.

PQG Window

[1] X window position in pixels
[2] Y window position in pixels
[3] X window size in pixels
[4] Y window size in pixels
[5] unused
[6] unused
[7] unused
[8] colormap
[9] foreground color
[10] background color
[11] aspect ratio attribute
[12] unused
[13] resizing attribute
[14] x pixel at left edge
[15] y pixel at top edge
[16] **GAUSS** window handle
[17] system window handle
[18] active window flag
[19] window type
[20] unused
[21] unused
[22] unused

[23] bits per pixel
[24] unused
[25] unused

Text Window

[1] X window position in pixels
[2] Y window position in pixels
[3] window rows
[4] window columns
[5] buffer rows
[6] buffer columns
[7] font (handle)
[8] colormap
[9] foreground color
[10] background color
[11] text wrapping mode
[12] automatic refresh mode
[13] resizing attribute
[14] buffer row at top of window
[15] buffer column at left side of window
[16] **GAUSS** window handle
[17] system window handle
[18] active window flag
[19] window type
[20] cursor text row
[21] cursor text column
[22] cursor type
[23] bits per pixel
[24] character height in pixels
[25] character width in pixels

TTY Window

[1] X window position in pixels
[2] Y window position in pixels
[3] window rows

[4] window columns
[5] number of lines to log
[6] number of lines logged
[7] font (handle)
[8] colormap
[9] foreground color
[10] background color
[11] text wrapping mode
[12] automatic refresh mode
[13] resizing attribute
[14] buffer line at top of window
[15] buffer column at left of window
[16] **GAUSS** window handle
[17] system window handle
[18] active window flag
[19] window type
[20] cursor text row (number of lines logged+1)
[21] cursor text column
[22] cursor type
[23] bits per pixel
[24] character height in pixels
[25] character width in pixels

Remarks

WinGetAttributes returns the attributes of the window specified by *whandle*. If *whandle* is 1, the attributes of the **GAUSS** command window are returned. If *whandle* is 2, the attributes of the **GAUSS** help window are returned.

wattr[1] and *wattr*[2] indicate the position on the display of the upper left corner of the window border (i.e., the border the window manager supplies).

wattr[3] and *wattr*[4] indicate the size of the window. For PQG windows, these values are given in pixels, for Text and TTY windows, these values are given in rows and columns.

wattr[8] is the colormap number as specified in the **WinSetColormap** command.

For a PQG window, *wattr*[11] indicates whether the aspect ratio is fixed or not. For Text and TTY windows it indicates how text is wrapped. *wattr*[12] indicates how the window is refreshed. *wattr*[13] indicates how the window can be resized.

wattr[14] and *wattr*[15] indicate the buffer coordinates at the upper left corner of Text windows. For TTY windows, *wattr*[14] indicates the line of the output log displayed at the top of the window.

wattr[16], the **GAUSS** window handle, is the handle ID assigned by **GAUSS**. *wattr*[17], the system window handle, is the handle ID assigned by the window manager. All **GAUSS** functions that take a window handle expect a **GAUSS** window handle.

wattr[18] indicates whether this window is the active window (1 = *true*, 0 = *false*).

wattr[22] is the cursor type number as specified in the **csrtype** command.

See **WinOpenPQG**, **WinOpenText**, and **WinOpenTTY** for details on window attributes.

Purpose

Get normalized RGB values from a window colormap.

Format

rgb = **WinGetColorCells(***whandle*,*cellidx*);

Input

whandle	scalar, window handle.
cellidx	N×1 vector, color cell indices.

Output

rgb	N×3 matrix, RGB settings of color cells.

Remarks

Each row of *rgb* contains the RGB values for the color cell specified in the corresponding row of *cellidx*. The RGB values are normalized to range from 0.0 (black) to 1.0 (full intensity). The columns are laid out as follows:

[.,1]	red values
[.,2]	green values
[.,3]	blue values

If any row of *cellidx* contains an invalid color cell index, the corresponding row of *rgb* will contain −1 in each column.

Purpose

Get the cursor position for a Text or TTY window.

Format

{ *r*,*c* } = **WinGetCursor(***whandle***);**

Input

whandle	scalar, window handle.

Output

r	scalar, cursor row position.
c	scalar, cursor column position.

Remarks

If *whandle* is a Text window, (r,c) specifies the location of the cursor in the window buffer. If *whandle* is a TTY window, r specifies the current output line (i.e., the number of lines logged + 1), c specifies the cursor location on that line. If *whandle* is a PQG window, r and c are meaningless.

For a Text window, the upper left corner of the buffer is (1,1). For a TTY window, the first column of the line is column 1.

Purpose

Move a window.

Format

WinMove(*whandle*,*x*,*y*);

Input

whandle	scalar, window handle.
x	scalar, X window position in pixels.
y	scalar, Y window position in pixels.

Remarks

WinMove moves the window specified by *whandle* to the position (x,y). (x,y) specifies the position on the display of the upper left corner of the window border (i.e., the border supplied by the window manager). The upper left corner of the display is (0,0). (x,y) can specify an off-screen location.

To use the current value of x or y, set that argument to a missing value.

Purpose

Open a PQG graphics window.

Format

whandle = **WinOpenPQG(***wattr*,*wtitle*,*ititle***);**

Input

wattr 13×1 vector, window attributes.

[1]	X window position in pixels.
[2]	Y window position in pixels.
[3]	X window size in pixels.
[4]	Y window size in pixels.
[5]	unused.
[6]	unused.
[7]	unused.
[8]	colormap.
[9]	foreground color.
[10]	background color.
[11]	aspect ratio attribute.
[12]	unused.
[13]	resizing attribute.

wtitle string, window title.

ititle string, icon title.

Output

whandle scalar, window handle.

Remarks

wattr[1] and *wattr*[2] specify the position on the display of the upper left corner of the window border (i.e., the border supplied by the window manager). The upper left corner of the display is (0,0).

wattr[8] specifies the colormap for the window. See **WinSetColormap** for the list of valid colormap values.

wattr[11], the aspect ratio attribute, determines whether the aspect ratio of the window is maintained during resize and zoom actions:

0	aspect ratio can change
1	aspect ratio is fixed

The aspect ratio attribute cannot be changed.

wattr[13], the resizing attribute, controls the resizing of the window as follows:

0	the window cannot be resized
1	the window can be resized only through a **WinResize** call
2	the window can be resized through a **WinResize** call or external event (e.g., resizing with the mouse)

The resizing attribute cannot be changed.

Default values are defined for the elements of *wattr* as follows:

[1]	0	window position: $(0, 0)$
[2]	0	
[3]	640	window size: 640×480
[4]	480	
[5]	0	unused
[6]	0	unused
[7]	1	unused
[8]	6	colormap: 16 color
[9]	15	foreground color: white
[10]	0	background color: black
[11]	1	fixed aspect ratio
[12]	0	unused
[13]	2	resize by function or mouse

To use the default value for any attribute, set that element to a missing value.

WinOpenPQG returns a -1 if it fails to open the window.

Note that opening a window does not make it the active window.

Purpose

Open a Text window.

Format

whandle = **WinOpenText(***wattr*,*wtitle*,*ititle***);**

Input

wattr 13×1 vector, window attributes.

[1]	X window position in pixels.
[2]	Y window position in pixels.
[3]	window rows.
[4]	window columns.
[5]	buffer rows.
[6]	buffer columns.
[7]	font (handle).
[8]	colormap.
[9]	foreground color.
[10]	background color.
[11]	text wrapping mode.
[12]	automatic refresh mode.
[13]	resizing attribute.

wtitle string, window title.

ititle string, icon title.

Output

whandle scalar, window handle.

Remarks

wattr[1] and *wattr*[2] specify the position on the display of the upper left corner of the window border (i.e., the border supplied by the window manager). The upper left corner of the display is (0,0).

wattr[8] specifies the colormap for the window. See **WinSetColormap** for the list of valid colormap values.

wattr[11], the text wrapping mode, controls how text is wrapped in the window buffer as follows:

0	text is word-wrapped
1	text is character-wrapped
2	text is clipped

The text wrapping mode can be changed with **WinSetTextWrap**.

wattr[12], the automatic refresh mode, controls how the window display is refreshed as follows:

0	the display is refreshed whenever a *newline* (\n) is output
1	the display is refreshed whenever anything is output
2	the display is never automatically refreshed; you must explicitly refresh it

The refresh mode can be changed with **WinSetRefresh**.

wattr[13], the resizing attribute, controls the resizing of the window as follows:

0	the window cannot be resized
1	the window can be resized only through a **WinResize** call
2	the window can be resized through a **WinResize** call or external event (e.g., resizing with the mouse)

The resizing attribute cannot be changed.

Default values are defined for the elements of *wattr* as follows:

[1]	0	window position: $(0, 0)$
[2]	0	
[3]	25	window size: 25×80
[4]	80	
[5]	25	buffer size: 25×80
[6]	80	
[7]	1	system default font
[8]	0	system default colormap
[9]	1	system foreground color
[10]	0	system background color
[11]	1	character wrap
[12]	0	refresh on \n
[13]	2	resize by function or mouse

To use the default value for any attribute, set that element to a missing value.

WinOpenText returns a -1 if it fails to open the window.

Note that opening a window does not make it the active window.

Purpose

Open a TTY window.

Format

whandle = **WinOpenTTY(***wattr*,*wtitle*,*ititle***);**

Input

wattr 13×1 vector, window attributes.

[1]	X window position in pixels.
[2]	Y window position in pixels.
[3]	window rows.
[4]	window columns.
[5]	number of lines to log.
[6]	unused.
[7]	font (handle).
[8]	colormap.
[9]	foreground color.
[10]	background color.
[11]	text wrapping mode.
[12]	automatic refresh mode.
[13]	resizing attribute.

wtitle string, window title.

ititle string, icon title.

Output

whandle scalar, window handle.

Remarks

wattr[1] and *wattr*[2] specify the position on the display of the upper left corner of the window border. The upper left corner of the display is (0,0).

wattr[8] specifies the colormap for the window. See **WinSetColormap** for the list of valid colormap values.

wattr[11], the text wrapping mode, controls how text is wrapped in the window as follows:

0	text is word-wrapped
1	text is character-wrapped
2	text is clipped

The text wrapping mode can be changed with **WinSetTextWrap**.

wattr[12], the automatic refresh mode, controls how the window display is refreshed as follows:

0	the display is refreshed whenever a *newline* (\n) is output
1	the display is refreshed whenever anything is output
2	the display is never automatically refreshed; you must explicitly refresh it

The refresh mode can be changed with **WinSetRefresh**.

wattr[13], the resizing attribute, controls the resizing of the window as follows:

0	the window cannot be resized
1	the window can be resized only through a **WinResize** call
2	the window can be resized through a **WinResize** call or external event (e.g. resizing with the mouse)

The resizing attribute cannot be changed.

Default values are defined for the elements of *wattr* as follows:

[1]	0	window position: $(0, 0)$
[2]	0	
[3]	25	window size: 25×80
[4]	80	
[5]	500	log 500 lines
[6]	0	unused
[7]	1	system default font
[8]	0	system default colormap
[9]	1	system foreground color
[10]	0	system background color
[11]	1	character wrap
[12]	0	refresh on \n
[13]	2	resize by function or mouse

To use the default value for any attribute, set that element to a missing value.

WinOpenTTY returns a -1 if it fails to open the window.

Note that opening a window does not make it the active window.

Purpose

Pan through a Text or TTY window.

Format

WinPan(*whandle*,*r*,*c*);

Input

whandle	scalar, window handle.
r	scalar, number of rows to pan.
c	scalar, number of columns to pan.

Remarks

WinPan pans through the contents of the window specified by *whandle*. To pan up and left, use negative *r* and *c*. To pan down and right, use positive *r* and *c*.

WinPan is not supported for PQG windows.

Purpose

Print to a specified window.

Format

WinPrint ⟦**/flush**⟧ ⟦/*typ*⟧ ⟦/*fmted*⟧ ⟦/*mf*⟧ ⟦/*jnt*⟧ *whandle* ⟦*expr1 expr2*...⟧**;**⟦**;**⟧

Input

whandle scalar, window handle.

Remarks

WinPrint and the **print** statement have exactly the same syntax and behavior, except that **WinPrint** prints to a specified window, and **print** prints to the active window.

WinPrint is not supported for PQG windows.

Purpose

Prints a PQG graph to a file or the printer.

Format

WinPrintPQG(*whandle*,*prtopts***);**

Input

whandle scalar, window handle.

prtopts string, print options.

WinPrintPQG prints the graphics data for *whandle* to the system printer, or to a file, if one is specified in *prtopts*.

prtopts consists of a series of option flags and their settings. Separate items in *prtopts* with spaces.

The following flags and settings are defined:

-f output file name, any valid filename. File must be writeable.

-o print orientation.

L	landscape
P	portrait

-t printer/file type.

PS	PostScript
EPS	Encapsulated PostScript
HPLJET3	HP LaserJet III, IIID
HPGL	Hewlett-Packard Graphics Language
PIC	Lotus PIC

The file `prndev.tbl` in the $GAUSSHOME directory lists additional printer/file types that are available but not tested. You can try any of the types listed there except the type 4 entries (i.e., those with a 4 in the second field), which are not supported.

-r print resolution.

1	low resolution
2	medium resolution
3	high resolution
4	maximum resolution

-m print margins, specified in inches.

left,*right*,*top*,*bottom*

This is a list of four decimal values. Separate margin values with commas. Do not put any white space between values.

Example:

-m .5,.5,.5,.75

-c print color mode.

BW	black/white
GRAY	gray scale
RGB	16 color

-a print aspect ratio.

F	fixed
V	variable

-g print placement.

TL	top left
T	top
TR	top right
L	left
C	center
R	right
BL	bottom left
B	bottom
BR	bottom right

Flags and settings are case-sensitive.

Print options not specified in *prtopts* are set to their values in `.gaussrc`. *prtopts* can be a null string.

WinPrintPQG is supported only for PQG windows.

Example

```
WinPrintPQG(wh, -f /tmp/graph.out -o P -t EPS);

WinPrintPQG(wh, -m 1,1,1,1.5 -a V -r 1);

WinPrintPQG(wh, -a F -g TL -m 1,1,1,4);
```

Purpose

Refresh a window.

Format

WinRefresh(*whandle***);**

Input

whandle scalar, window handle.

Remarks

WinRefresh redraws the window specified by *whandle* using the current contents of the window's graphics data, text buffer or output log, depending upon the window type.

This is the only way to update windows with an automatic refresh mode = [$no\ refresh$].

Purpose

Refresh a rectangular region of a Text window.

Format

WinRefreshArea(*whandle*,*row*,*col*,*nrows*,*ncols*);

Input

whandle	scalar, window handle.
row	scalar, top row of area in buffer to refresh.
col	scalar, left column of area in buffer to refresh.
nrows	scalar, number of rows to refresh.
ncols	scalar, number of cols to refresh.

Remarks

WinRefreshArea redraws a rectangular region of the Text window using the current contents of the text buffer. The region has its top-left corner at (*row*,*col*), and is *nrows*×*ncols* in size.

Purpose

Resize a window.

Format

WinResize(*whandle*,*height*,*width*);**

Input

whandle	scalar, window handle.
height	scalar, window height.
width	scalar, window width.

Remarks

WinResize changes the size of the window specified by *whandle*. The position of the upper left corner of the window remains fixed.

For PQG windows, *height* and *width* are specified in pixels. The final size of the window may differ from the requested size. If the window was opened with fixed aspect ratio, the *width* will be used to calculate the *height* required to maintain the correct ratio.

For Text and TTY windows, *height* and *width* are specified in characters. (Thus, the actual requested size of the window is dependent upon the window font.) For Text windows, the size of the window cannot be larger than the text buffer.

height and *width* can be nonintegral, but will be truncated to integers..

To use the current value of *height* or *width*, set that argument to a missing value.

Purpose

Set the active window.

Format

whout = **WinSetActive(***whin***);**

Input

whin	scalar, window handle or −1.

Output

whout	scalar, window handle.

Remarks

WinSetActive sets the active window to *whin* and returns the handle of the previous active window in *whout*. If *whin* is −1, the active window is deactivated.

If there was no previous active window, **WinSetActive** returns a −1.

Purpose

Set the background color for the specified window.

Format

oldback = **WinSetBackground(***whandle*,*color***);**

Input

whandle	scalar, window handle.
color	scalar, color index or −1.

Output

oldback	scalar, previous background color index.

Remarks

WinSetBackground sets the background color index for *whandle* to *color* and returns the previous background color index. If *color* is −1, the current background color index is returned.

Purpose

Set one or more cells in a window's colormap.

Format

ret = **WinSetColorCells(***whandle*,*cellidx*,*colorspec*);

Input

whandle	scalar, window handle.
cellidx	N×1 vector, color cell indices.
colorspec	N×3 matrix, color specifications.

Output

ret	scalar, 0 if call is successful, −1 if one or more colors can't be found.

Remarks

The columns of *colorspec* are interpreted as follows:

[.,1]	red values
[.,2]	green values
[.,3]	blue values

Color values can range from 0.0 (black) to 1.0 (full intensity). Values < 0.0 will be treated as 0.0, values > 1.0 will be treated as 1.0.

Due to the X11 color handling protocol, **WinSetColorCells** has to regard the contents of *colorspec* as color "suggestions" rather than color commands, but it will always give you colors as close to the settings of *colorspec* as it can. See Chapter 3 for details.

WinSetColorCells attempts to change the RGB values for the specified color cells. If it succeeds in setting all of them, it returns a 0. If it fails to set all of them — specifically, if *cellidx* or *colorspec* contain any invalid values, or the system fails to reset the color cell — **WinSetColorCells** returns a −1.

Purpose

Set the colormap for a window.

Format

mapinfo = **WinSetColormap(***whandle*,*cmap*);

Input

whandle	scalar, window handle.
cmap	scalar, colormap number or −1.

The supported colormaps are:

0	System Color
1	System BW
2	True BW
3	4 Gray
4	4 Color
5	16 Gray
6	16 Color
7	64 Gray
8	256 Color

Output

mapinfo 3×1 vector, colormap information.

mapinfo[1]	number of colors in colormap
mapinfo[2]	number of bits per pixel
mapinfo[3]	current colormap number

Remarks

WinSetColormap sets the colormap for the window *whandle* to *cmap*, and returns information about the selected colormap. If *cmap* = −1, no changes are made, but the information for the current colormap is returned.

On error, scalar 0 is returned.

Purpose

Set the cursor position for a Text or TTY window.

Format

WinSetCursor(*whandle*,*r*,*c*);

Input

whandle	scalar, window handle.
r	scalar, cursor row.
c	scalar, cursor column.

Remarks

If *whandle* is a Text window, (*r*,*c*) specifies where to locate the cursor in the window buffer. If *whandle* is 1 or a TTY window, *r* is ignored, and *c* specifies where to locate the cursor on the current line.

For a Text window, the upper left corner of the buffer is (1,1). For a TTY window, the first column of a line is column 1.

(*r*,*c*) can specify a location that is outside the buffer. You can think of the buffer as covering a small region in "text coordinate space"; text written within the buffer bounds is remembered, text written outside it is lost.

Note that there is no right-hand boundary in a TTY window. So, if you locate to column 254328 and begin printing, **GAUSS** will attempt to allocate enough memory to store a 254K line of output.

r and *c* are truncated if they are not integers.

WinSetCursor is not supported for PQG windows.

Purpose

Set the foreground color for the specified window.

Format

oldfore = **WinSetForeground(***whandle*,*color*);

Input

whandle	scalar, window handle.
color	scalar, color index or −1.

Output

oldfore	scalar, previous foreground color index.

Remarks

WinSetForeground sets the foreground color index for *whandle* to *color* and returns the previous foreground color index. If *color* is −1, the current foreground color index is returned.

Purpose

Set the refresh mode for a window.

Format

oldref = **WinSetRefresh(***whandle*,*refresh*);

Input

whandle	scalar, window handle.
refresh	scalar, refresh mode.

Output

oldref	scalar, previous refresh mode.

Remarks

refresh can be one of the following:

0	refresh on *newline* (\n); this is the system default
1	refresh on any output
2	refresh only on **WinRefresh** call

To get the current refresh mode, call **WinGetAttributes**.

WinSetRefresh is not supported for window 1 or PQG windows.

Purpose

Set the text wrapping mode for a window.

Format

oldtwrap = **WinSetTextWrap(***whandle*,*textwrap*);

Input

whandle	scalar, window handle.
textwrap	scalar, text wrapping mode.

Output

oldtwrap	scalar, previous text wrapping mode.

Remarks

textwrap can be one of the following:

0	text is word-wrapped
1	text is character-wrapped
2	text is clipped

For Text windows, the text wrapping mode specifies how lines are stored in the window buffer. For TTY windows, the text wrapping mode specifies how lines are displayed in the window. Thus, for Text windows, clipped output is actually lost, as is output that scrolls off the top of the buffer; for TTY windows, clipped output is merely displayed differently than wrapped output.

To get the current text wrap mode, call **WinGetAttributes**.

WinSetTextWrap is not supported for PQG windows.

Purpose

Display a zoomed image in a PQG window.

Format

WinZoomPQG(*whandle*,*x*,*y*,*xsize*,*ysize*);

Input

whandle	scalar, window handle.
x	scalar, x position of zoom rectangle in pixels.
y	scalar, y position of zoom rectangle in pixels.
xsize	scalar, x size of zoom rectangle in pixels.
ysize	scalar, y size of zoom rectangle in pixels.

Remarks

WinZoomPQG displays a zoomed image of the area specified. If the PQG window has the aspect ratio attribute set to **fixed**, the y size of the rectangle will be adjusted as needed to maintain the aspect ratio.

To zoom out (that is, to redisplay the entire graph in the PQG window), use:

```
WinZoomPQG(whandle,0,0,0,0);
```

WinZoomPQG provides the same capabilities as the "Zoom" pop-up menu available in the PQG windows.

For more details, consult the PQG graphics section of the manual.

WinZoomPQG is supported only for PQG windows.

Chapter 23

Command Reference Introduction

The *COMMAND REFERENCE* describes each of the commands, procedures and functions available in the **GAUSS** programming language. These functions can be roughly divided into three 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.

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 which 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 which create, save, open and read from and write to **GAUSS** data sets. A variety of sorting routines which will operate on both numeric and character data are also available.

Programming statements are all of the commands which 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.

23.1 Using This Manual

Users who are new to **GAUSS** should make sure they have familiarized themselves with Chapter 7 in Volume I before proceeding here. That chapter contains the basics of **GAUSS** programming.

In all, there are over 400 routines described in this manual. We suggest that new **GAUSS** users skim through Chapter 24, and then browse through the main part of this manual, the *COMMAND REFERENCE* section. Here, the user can familiarize him or herself with the kinds of tasks that **GAUSS** can handle easily.

Chapter 24 gives a categorical listing of all functions in the *COMMAND REFERENCE* section, 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 the *COMMAND REFERENCE* section.

If a function is an *extrinsic* (i.e., 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 the *COMMAND REFERENCE* under the discussion of that function.

23.2 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. See Section 23.2.1 for information on how to change default values.

23.2.1 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 2 to 0, for example, edit the file `gauss.dec` and change the statement

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

so it reads:

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

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

```
__output = 2;
```

similarly.

23.2.2 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.

Chapter 24

Commands by Category

24.1 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.
cos	Computes cosine.
cosh	Computes hyperbolic cosine.
exp	Computes the exponential function of x.
gamma	Computes gamma function value.
ln	Computes the natural log of each element.
lnfact	Computes natural log of factorial function.
log	Computes the $\log_{10}$ of each element.
pi	Returns π.
sin	Computes sine.
sinh	Computes the hyperbolic sine.
sqrt	Computes the square root of each element.
tan	Computes tangent.
tanh	Computes hyperbolic tangent.

All trigonometric functions take or return values in radian units.

Differentiation and Integration

gradp	Computes first derivative of a function.
hessp	Computes second derivative of a function.
intgrat2	Integrate a 2-dimensional function over a user-defined region.
intgrat3	Integrate a 3-dimensional function over a user-defined region.
intquad1	Integrate a 1-dimensional function.
intquad2	Integrate a 2-dimensional function over a user-defined rectangular region.
intquad3	Integrate a 3-dimensional function over a user-defined rectangular region.
intsimp	Integrate 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 one-dimensional integration.

Linear Algebra

balance	Balances a matrix.
cond	Computes condition number of a matrix.
chol	Computes Cholesky decomposition, $X = Y'Y$.
choldn	Performs Cholesky downdate on an upper triangular matrix.
cholsol	Solves a system of equations given the Cholesky factorization of a matrix.
cholup	Performs Cholesky update on an upper triangular matrix.
crout	Computes Crout decomposition, $X = LU$ (real matrices only).
croutp	Computes Crout decomposition with row pivoting (real matrices only).
det	Computes determinant of square matrix.
detl	Computes determinant of decomposed matrix.
hess	Computes upper Hessenberg form of a matrix (real matrices only).
inv	Inverts a matrix.

invpd	Inverts a positive definite matrix.
invswp	Generalized sweep inverse.
lu	Computes LU decomposition with row pivoting (real and complex matrices).
null	Computes orthonormal basis for right null space.
null1	Computes orthonormal basis for right null space.
orth	Computes orthonormal basis for column space x.
pinv	Generalized pseudo-inverse: Moore-Penrose.
qqr	QR decomposition: returns Q_1 and R.
qqre	QR decomposition: returns Q_1, R and a permutation vector, E.
qqrep	QR decomposition with pivot control: returns Q_1, R and E.
qr	QR decomposition: returns R.
qre	QR decomposition: returns R and E.
qrep	QR decomposition with pivot control: returns R and E.
qtyr	QR decomposition: returns $Q'Y$ and R.
qtyre	QR decomposition: returns $Q'Y$,R and E.
qtyrep	QR decomposition with pivot control: returns $Q'Y$,R and E.
qyr	QR decomposition: returns QY and R.
qyre	QR decomposition: returns QY, R and E.
qyrep	QR decomposition with pivot control: returns QY, R and E.
qrsol	Solves a system of equations $Rx = b$ given an upper triangular matrix, typically the R matrix from a QR decomposition.
qrtsol	Solves a system of equations $R'x = b$ given an upper triangular matrix, typically the R matrix from a QR decomposition.
rank	Computes rank of a matrix.
rcondl	Returns reciprocal of the condition number of last decomposed matrix.
rref	Computes reduced row echelon form of a matrix.
schur	Computes Schur decomposition of a matrix (real matrices only).
solpd	Solves a system of positive definite linear equations.
svd	Computes the singular values of a matrix.
svd1	Computes singular value decomposition, $X = USV'$.
svd2	Computes **svd1** with compact U.

The decomposition routines are **chol** for Cholesky decomposition, **crout** and **croutp** for Crout decomposition, **qqr**–**qyrep** for QR decomposition, and **svd**, **svd1**, and **svd2** for singular value decomposition.

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

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

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

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

Eigenvalues

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

There are four eigenvalue-eigenvector routines. Two calculate eigenvalues only, and two calculate eigenvalues and eigenvectors. The 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.
fftn	Computes 1- or 2-D FFT using prime factor algorithm.
ffti	Computes inverse 1- or 2-D FFT.
fftm	Computes multi-dimensional FFT.
fftmi	Computes inverse multi-dimensional FFT.
rfft	Computes real 1- or 2-D FFT.

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

Random Numbers

rndbeta	Computes random numbers with beta distribution.
rndgam	Computes random numbers with gamma distribution.
rndn	Computes random numbers with Normal distribution.
rndns	Computes random numbers with Normal distribution using specified seed.
rndnb	Computes random numbers with negative binomial distribution.
rndp	Computes random numbers with Poisson distribution.
rndu	Computes random numbers with uniform distribution.
rndus	Computes random numbers with uniform distribution using specified seed.
rndcon	Changes constant of random number generator.
rndmod	Changes modulus of random number generator.
rndmult	Changes multiplier of random number generator.
rndseed	Changes seed of random number generator.

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

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

seed = 44435667;
x = rndus(1,1,seed);
```

Fuzzy Conditional Functions

feq	Fuzzy ==
fge	Fuzzy >=
fgt	Fuzzy >

fle	Fuzzy <=
flt	Fuzzy <
fne	Fuzzy / =
dotfeq	Fuzzy . ==
dotfge	Fuzzy . >=
dotfgt	Fuzzy . >
dotfle	Fuzzy . <=
dotflt	Fuzzy . <
dotfne	Fuzzy ./ =

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

Statistical Functions

dstat	Computes descriptive statistics of a data set or matrix.
conv	Computes convolution of two vectors.
corrm	Computes correlation matrix of a moment matrix.
corrvc	Computes correlation matrix from a variance-covariance matrix.
corrx	Computes correlation matrix.
crossprd	Computes cross product.
design	Creates a design matrix of 0's and 1's.
meanc	Computes mean value of each column of a matrix.
median	Computes medians of the columns of a matrix.
moment	Computes moment matrix ($x'x$) with special handling of missing values.
momentd	Computes moment matrix from a data set.
ols	Computes least squares regression of data set or matrix.
olsqr	Computes OLS coefficients using QR decomposition.
olsqr2	Computes OLS coefficients, residuals, and predicted values using QR decomposition.
stdc	Computes standard deviation of the columns of a matrix.
toeplitz	Computes Toeplitz matrix from column vector.
vcm	Computes a variance-covariance matrix from a moment matrix.
vcx	Computes a variance-covariance matrix from a data matrix.

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

Statistical Distributions

cdfbeta	Computes integral of beta function.
cdfbvn	Computes lower tail of bivariate Normal cdf.
cdfchic	Computes complement of cdf of χ^2.
cdfchii	Computes χ^2 abscissae values given probability and degrees of freedom.
cdfchinc	Computes integral of noncentral χ^2.
cdffc	Computes complement of cdf of F.
cdffnc	Computes integral of noncentral F.
cdfgam	Computes integral of incomplete Γ function.
cdfmvn	Computes multivariate Normal cdf.
cdfn	Computes integral of Normal distribution: lower tail, or cdf.
cdfn2	Computes interval of Normal cdf.
cdfnc	Computes complement of cdf of Normal distribution (upper tail).
cdftc	Computes complement of cdf of t-distribution.
cdftnc	Computes integral of noncentral t-distribution.
cdftvn	Computes lower tail of trivariate Normal cdf.
erf	Computes Gaussian error function.
erfc	Computes complement of Gaussian error function.
lncdfbvn	Computes natural log of bivariate Normal cdf.
lncdfmvn	Computes natural log of multivariate Normal cdf.
lncdfn	Computes natural log of Normal cdf.
lncdfn2	Computes natural log of interval of Normal cdf.
lncdfnc	Computes natural log of complement of Normal cdf.
lnpdfn	Computes Normal log-probabilities.
lnpdfmvn	Computes multivariate Normal log-probabilities.
pdfn	Computes standard Normal probability density function.

Series and Sequence Functions

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

Precision Control

base10	Convert number to *x.xxx* and a power of 10.
ceil	Round up towards $+\infty$.
floor	Round down towards $-\infty$.
prcsn	Set computational precision for matrix operations.
round	Round to the nearest integer.
trunc	Truncate toward 0.

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

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

Each matrix element in memory requires 8 bytes of workspace. See the function **coreleft** to determine the amount of workspace available.

24.2 Matrix Manipulation

Creating Vectors and Matrices

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

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

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

Matrices may also be loaded from an ASCII file, from a **GAUSS** matrix file, or from a **GAUSS** data set. For more information see the *File I/O* chapter in Volume I of the manual.

Loading and Storing Matrices

loadd	Load matrix from data set.
loadm	Load matrix from ASCII or matrix file.
save	Save matrix to matrix file.
saved	Save matrix to data set.

Size, Ranking, and Range

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

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

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

Sparse Matrix Functions

sparseFD	Converts dense matrix to sparse matrix.
sparseFP	Converts packed matrix to sparse matrix.
sparseOnes	Generates sparse matrix of ones and zeros.
sparseEye	Creates sparse identity matrix.
sparseTD	Multiplies sparse matrix by dense matrix.
sparseTrTD	Multiplies sparse matrix transposed by dense matrix.
sparseSolve	Solves $Ax = B$ for x where A is a sparse matrix.
sparseHConcat	Horizontally oncatenates sparse matrices.
sparseVConcat	Vertically concatenates sparse matrices.
sparseSubmat	Returns sparse submatrix of sparse matrix.
denseSubmat	Returns dense submatrix of sparse matrix.
sparseRows	Returns number of rows in sparse matrix.
sparseCols	Returns number of columns in sparse matrix.
sparseNZE	Returns the number of nonzero elements in sparse matrix.
isSparse	Tests whether a matrix is a sparse matrix.
sparseSet	Resets sparse library globals.

Miscellaneous Matrix Manipulation

rev	Reverses the order of rows of a matrix.
rotater	Rotates the rows of a matrix, wrapping elements as necessary.
shiftr	Shifts rows of a matrix, filling in holes with a specified value.
reshape	Reshapes a matrix to new dimensions.
vec	Stacks columns of a matrix to form a single column.
vech	Reshapes the lower triangular portion of a symmetric matrix into a column vector.
vecr	Stacks rows of a matrix to form a single column.
xpnd	Expands a column vector into a symmetric matrix.
delif	Deletes rows from a matrix using a logical expression.
diag	Extracts the diagonal of a matrix.
diagrv	Puts a column vector into the diagonal of a matrix.
exctsmpl	Creates a random subsample of data set, with replacement.
lowmat	Returns the main diagonal and lower triangle.
lowmat1	Returns a main diagonal of 1's and the lower triangle.
upmat	Returns the main diagonal and upper triangle.
upmat1	Returns a main diagonal of 1's and the upper triangle.
selif	Selects rows from a matrix using a logical expression.

submat	Extracts a submatrix from a matrix.
trimr	Trims rows from top or bottom of a matrix.
union	Returns the union of two vectors.
intrsect	Returns the intersection of two vectors.
setdif	Returns elements of one vector that are not in another.
complex	Creates a complex matrix from two real matrices.
imag	Returns the imaginary part of a complex matrix.
real	Returns the real part of a complex matrix.

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

delif and **selif** are complementary functions. **delif** deletes rows of a matrix based on a logical comparison, and **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 the function **packr**.

24.3 Data Handling

Data Sets

close	Closes an open data set (`.dat` file).
closeall	Closes all open data sets.
create	Creates and opens a data set.
eof	Tests for end of file.
loadd	Loads a small data set.
open	Opens an existing data set.
readr	Reads rows from open data set.
saved	Creates small data sets.
seekr	Moves pointer to specified location in open data set.
iscplxf	Returns whether a data set is real or complex.
typef	Returns the element size (2, 4 or 8 bytes) of data in open data set.
writer	Writes matrix to an open data set.

These functions all operate on **GAUSS** data sets (`.dat` files). See the *File I/O* chapter in Volume I for more discussion.

To create a **GAUSS** data set from a matrix in memory, use the function **saved**. To create a data set from an existing one, use **create**. To create a data set from a large ASCII file, use the utility **atog** (see Chapter 19 in Volume I).

Data sets can be opened, read from, and written to using the functions **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 typed as character or numeric. See the *File I/O* chapter in Volume I, **create**, and **vartypef**.

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

Data Set Variable Names

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

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

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

Data Coding

code	Code the data in a vector by applying a logical set of rules to assign each data value to a category.
dummy	Creates a dummy matrix, expanding values in vector

	to rows with ones in columns corresponding to true categories and zeros elsewhere.
dummybr	Similar to **dummy**.
dummydn	Similar to **dummy**.
ismiss	Returns 1 if matrix has any missing values, 0 otherwise.
miss	Change specified values to missing value code.
missex	Change elements to missing value using logical expression.
missrv	Change missing value codes to specified values.
msym	Set symbol to be interpreted as missing value.
packr	Delete rows with missing values.
recode	Similar to **code**, but leaves the original data in place if no condition is met.
scalmiss	Test whether a scalar is the missing value code.
substute	Similar to **recode**, but operates on matrices.
subscat	Simpler version of **recode**, but uses ascending bins instead of logical conditions.

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

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

Sorting and Merging

sortc	Quick-sort rows of matrix based on numeric key.
sortcc	Quick-sort rows of matrix based on character key.
sortd	Sort data set on a key column.
sorthc	Heap-sort rows of matrix based on numeric key.
sorthcc	Heap-sort 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	Sort rows of matrix on the basis of multiple columns.
uniqindx	Returns a sorted unique index of a vector.
unique	Remove duplicate elements of a vector.
intrleav	Produces one large sorted data file from two smaller sorted files having the same keys.
mergeby	Produces one large sorted data file from two smaller sorted files having a single key column in common.

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

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.

24.4 Compiler Control

#define	Define a case-insensitive text-replacement or flag variable.
#definecs	Define a case-sensitive text-replacement or flag variable.
#undef	Undefine a text-replacement or flag variable.
#ifdef	Compile code block if a variable has been **#define**'d.
#ifndef	Compile code block if a variable has not been **#define**'d.
#iflight	Compile code block if running **GAUSS Light**.
#ifdos	Compile code block if running DOS.
#ifos2win	Compile code block if running OS/2 or Windows.
#ifunix	Compile code block if running Unix.
#else	Alternate clause for **#if**-**#else**-**#endif** code block.
#endif	End of **#if**-**#else**-**#endif** code block.
#include	Include code from another file in program.
#lineson	Compile program with line number and file name records.
#linesoff	Compile program without line number and file name records.
#srcfile	Insert source file name record at this point (currently used when doing data loop translation).
#srcline	Insert source file line number record at this point (currently used when doing data loop translation).

These commands are compiler directives. That is, they do not generate **GAUSS** program instructions; rather, they are instructions that tell **GAUSS** how to process a program during compilation. They determine what the final compiled form of a program will be. They are not executable statements and have no effect at run-time. See Section 7.4 for a discussion and examples of how to use compiler directives.

24.5 Program Control

Execution Control

end	Terminate a program and close all files.

pause	Pause for the specified time.
run	Run a program in a text file.
sleep	Sleep for the specified time.
stop	Stop a program and leave files open.
system	Quit and return 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** or **end** is required in a **GAUSS** program.

Branching

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

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

Looping

break	Jump out the bottom of a **do** or **for** loop.
continue	Jump to the top of a **do** or **for** loop.
do while.. endo	Loop if *TRUE*.
do until.. endo	Loop if *FALSE*.
for.. endfor	Loop with integer counter.

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

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

Subroutines

gosub	Branch to subroutine.
pop	Retrieve **gosub** arguments.
return	Return from subroutine.

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

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

Procedures

local	Declare variables local to a procedure.
proc	Begin definition of multi-line procedure.
retp	Return from a procedure.
endp	Terminate a procedure definition.

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

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

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

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

See Chapters 9 and 10 for more details.

Libraries

call	Call function and discard return values.
declare	Initialize variables at compile time.
external	External symbol definitions.
library	Set up list of active libraries.
lib	Build or update a **GAUSS** library.

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

Compiling

compile	Compiles and saves a program to a `.gcg` file.
saveall	Saves the contents of the current workspace to a file.
use	Loads previously compiled code.
save	Saves the compiled image of a procedure to disk.
loadp	Loads compiled procedure.

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

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

To save the compiled images of procedures *that do not make any global references*, use the **save** command. 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. This is here to maintain backward compatibility with previous versions.

24.6 OS Functions

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

24.7 Workspace Management

clear	Set matrices equal to 0.
clearg	Set global symbols to 0.
coreleft	Returns amount of workspace memory left.
maxvec	Returns maximum allowed vector size.
delete	Delete specified global symbols.
new	Clear current workspace.
show	Display global symbol table.
iscplx	Returns whether a matrix is real or complex.
hasimag	Examines matrix for nonzero imaginary part.
type	Returns type of argument (matrix or string).
typecv	Returns type of symbol (argument contains the name of the symbol to be checked).

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

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

24.8 Error Handling and Debugging

debug	Execute a program under the source level debugger.
disable	Disable invalid operation interrupt of coprocessor.
error	Create user-defined error code.
errorlog	Send error message to screen and log file.
enable	Enable invalid operation interrupt of coprocessor.
#lineson	Include line number and file name records in program.
#linesoff	Omit line number and file name records from program.
ndpchk	Examine status word of coprocessor.
ndpclex	Clear coprocessor exception flags.
ndpcntrl	Set and get coprocessor control word.
scalerr	Test for a scalar error code.
trace	Trace program execution for debugging.
trap	Control trapping of program errors.
trapchk	Examine the trap flag.

To trap coprocessor overflows, invalid operations, divide by zero, and other exceptions, use **ndpchk**. To clear these exceptions, use **ndpclex**. To set exceptions to be checked, use **ndpcntrl**.

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

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

24.9 String Handling

chrs	Convert ASCII values to a string.
ftocv	Convert an NxK matrix to a character matrix.
ftos	Convert a floating point scalar to string.
getf	Load ASCII or binary file into string.
loads	Loads a string file (`.fst` file).
lower	Convert a string to lowercase.
putf	Writes a string to disk file.
stof	Convert a string to floating point scalar.
strindx	Find starting location of one string in another string.
strlen	Returns length of a string.
strrindx	Find starting location of one string in another string, searching from the end to the start of the string.
strsect	Extract a substring of a string.
upper	Changes a string to uppercase.
vals	Convert a string to ASCII values.
varget	Access the global variable named by a string.
varput	Assign a global variable named by a string.
vargetl	Access the local variable named by a string.
varputl	Assign 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**.

24.10 Time and Date Functions

date	Returns current system date.
datestr	Formats date as "mm/dd/yy".
datestring	Formats date as "mm/dd/yyyy".
datestrymd	Formats date as "yyyymmdd".
dayinyr	Returns day number of a date.
etdays	Difference between two times in days.
ethsec	Difference between two times in 100ths of a second.
etstr	Convert elapsed time to string.
hsec	Returns elapsed time since midnight in 100ths of a second.
time	Returns current system time.
timestr	Format time as "hh:mm:ss".

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

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

will time the **GAUSS** multiplication operator.

24.11 Console I/O

con	Request console input, create matrix.
cons	Request console input, create string.
key	Gets the next key from the keyboard buffer. If buffer is empty, returns a 0.
keyw	Gets the next key from the keyboard buffer. If buffer is empty, waits for a key.
wait	Wait for a keystroke.
waitc	Flush buffer, then wait 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** in the *COMMAND REFERENCE*.

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

24.12 Output Functions

Text Output

comlog	Control interactive command logging.
output	Redirect **print** statements to auxiliary output.
outwidth	Set line width of auxiliary output.
print	Print to screen.
print [[**on**|**off**]]	Turn auto screen print on and off.
printfm	Print matrices using a different format for each column.
screen [[**on**|**off**]]	Direct/suppress **print** statements to screen.
screen out	Dump snapshot of screen to auxiliary output.
lpos	Returns print head position in printer buffer.
lprint	Print expression to the printer.
lprint [[**on**|**off**]]	Switch auto printer mode on and off.
lpwidth	Specifies printer width.
lshow	Print global symbol table on the printer.
plot	Plot elements of two matrices in text mode.
plotsym	Controls data symbol used by **plot**.
cls	Clear the screen.
color	Set pixel, text, background colors.
csrcol	Get column position of cursor on screen.
csrlin	Get row position of cursor on screen.
edit	Edits a file with the **GAUSS** editor.
ed	Access an alternate editor.
format	Defines format of matrix printing.
locate	Position the cursor on the screen.
printdos	Print a string for special handling by the OS.
scroll	Scroll a section of the screen.
tab	Position the cursor on the current line.

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

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

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

Screen Graphics

graph	Set pixels.
setvmode	Set video mode.
color	Set color.
line	Draw lines.

The function **graph** allows the user to plot individual pixels.

Chapter 25

Command Reference

Purpose

Returns the absolute value or complex modulus of x.

Format

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

Input

x	N×K matrix.

Output

y	N×K matrix containing absolute values of x.

Example

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

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

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

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

Purpose

Computes the inverse cosine.

Format

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

Input

x	N×K matrix.

Output

y	N×K matrix containing the angle in radians whose cosine is x.

Remarks

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

Example

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

$$x = \begin{matrix} -1.000000 \\ -0.500000 \\ 0.000000 \\ 0.500000 \\ 1.000000 \end{matrix}$$

$$y = \begin{matrix} 3.141593 \\ 2.094395 \\ 1.570796 \\ 1.047198 \\ 0.000000 \end{matrix}$$

Source

trig.src

Globals

None

Purpose

Computes the inverse sine.

Format

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

Input

x	N×K matrix.

Output

y	N×K matrix, the angle in radians whose sine is x.

Remarks

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

Example

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

$$x = \begin{matrix} -1.000000 \\ -0.500000 \\ 0.000000 \\ 0.500000 \\ 1.000000 \end{matrix}$$

$$y = \begin{matrix} -1.570796 \\ -0.523599 \\ 0.000000 \\ 0.523599 \\ 1.570796 \end{matrix}$$

Source

trig.src

Globals

None

Purpose

Returns the arctangent of its argument.

Format

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

Input

x	N×K matrix.

Output

y	N×K matrix containing the arctangent of x in radians.

Remarks

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

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

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

Example

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

$$y = \begin{matrix} 0.785398 \\ 1.107149 \\ 1.249046 \\ 1.325818 \end{matrix}$$

See also

atan2, **sin**, **cos**, **pi**, **tan**

Purpose

Compute an angle from an x,y coordinate.

Format

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

Input

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

Output

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

Remarks

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

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

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

Example

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

$$z = \begin{matrix} 0.785398 \\ 1.107149 \\ 1.249046 \\ 1.325818 \end{matrix}$$

See also

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

Purpose

Balances a square matrix.

Format

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

Input

x	K×K matrix.

Output

b	K×K matrix, balanced matrix.
z	K×K matrix, diagonal scale matrix.

Remarks

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

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

In particular,

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

balance uses the BALANC function from EISPACK.

Example

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

$$b = \begin{matrix} 100.0 & 100.0 & 37.5 \\ 80.0 & 50.0 & 15.0 \\ 56.0 & 32.0 & 9.0 \end{matrix}$$

$$z = \begin{matrix} 4.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 0.5 \end{matrix}$$

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 (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);
```

$$bx = \begin{matrix} 0.0000000 & 1.0000000 \\ 2.0000000 & 8.0000000 \\ 1.0000000 & 5.0000000 \\ 2.0000000 & 3.0000000 \end{matrix}$$

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);
```

$$bx = \begin{matrix} 0.0000000 & 1.0000000 \\ 2.0000000 & 8.0000000 \\ 1.0000000 & 5.0000000 \\ 2.0000000 & 3.0000000 \end{matrix}$$

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

$$cx = \begin{matrix} 0.0000000 & 1.0000000 \\ 2.0000000 & 2.0000000 \\ 0.50000000 & 2.1794495 \\ 0.91766294 & 1.4689774 \end{matrix}$$

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*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);
```

$$bx = \begin{matrix} 0.0000000 & 1.0000000 \\ 2.0000000 & 8.0000000 \\ 1.0000000 & 5.0000000 \\ 2.0000000 & 3.0000000 \end{matrix}$$

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

$$cx = \begin{matrix} 0.0000000 & 1.0000000 \\ 2.0000000 & 2.0000000 \\ 0.50000000 & 2.1794495 \\ 0.91766294 & 1.4689774 \end{matrix}$$

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

$$xi = \begin{matrix} 2.0731707 & -0.53658537 & 0.14634146 & -0.097560976 \\ -0.53658537 & 0.26829268 & -0.073170732 & 0.048780488 \\ 0.14634146 & -0.073170732 & 0.29268293 & -0.19512195 \\ -0.097560976 & 0.048780488 & -0.19512195 & 0.46341463 \end{matrix}$$

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

$$bx = \begin{matrix} 0.0000000 & 1.0000000 \\ 2.0000000 & 8.0000000 \\ 1.0000000 & 5.0000000 \\ 2.0000000 & 3.0000000 \end{matrix}$$

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

$$cx = \begin{matrix} 0.0000000 & 1.0000000 \\ 2.0000000 & 2.0000000 \\ 0.50000000 & 2.1794495 \\ 0.91766294 & 1.4689774 \end{matrix}$$

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

$$xci = \begin{matrix} 1.0000000 & 0.0000000 & 0.0000000 & 0.0000000 \\ -1.0000000 & 0.50000000 & 0.0000000 & 0.0000000 \\ 0.22941573 & -0.11470787 & 0.45883147 & 0.0000000 \\ -0.14331487 & 0.071657436 & -0.28662975 & 0.68074565 \end{matrix}$$

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

$$bx = \begin{matrix} 0.0000000 & 1.0000000 \\ 2.0000000 & 8.0000000 \\ 1.0000000 & 5.0000000 \\ 2.0000000 & 3.0000000 \end{matrix}$$

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

$$x = \begin{matrix} 1.0000000 & 2.0000000 & 0.0000000 & 0.0000000 \\ 2.0000000 & 8.0000000 & 1.0000000 & 0.0000000 \\ 0.0000000 & 1.0000000 & 5.0000000 & 2.0000000 \\ 0.0000000 & 0.0000000 & 2.0000000 & 3.0000000 \end{matrix}$$

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]
```

Purpose

Break 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

Globals

None

Purpose

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

Format

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

Input

n	N×K matrix, the order of the Bessel function. Nonintegers will be truncated to an integer.
x	L×M matrix, E×E conformable with n.

Output

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

Example

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

$$y = \begin{matrix} 0.99750156 & 0.67113274 \\ 0.53987253 & 0.17922585 \end{matrix}$$

See also

bessely, **mbesseli**

Purpose

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

Format

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

Input

n	N×K matrix, the order of the Bessel function. Nonintegers will be truncated to an integer.
x	L×M matrix, E×E conformable with n.

Output

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

Example

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

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

See also

besselj, **mbesseli**

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**

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

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:

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

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);
```

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

See also

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

Technical Notes

cdfbeta has the following approximate accuracy:

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

References:

1. "ALGORITHM 179 INCOMPLETE BETA RATIO", O.G. Ludwig, Comm. ACM, Vol. 6, No. 6, pp. 314, June 1963

2. "REMARK ON ALGORITHM 179 INCOMPLETE BETA RATIO", M.C. Pike and I.D. Hill, Comm. ACM, Vol. 10, No. 6, pp. 375-6, June. 1967

3. "REMARK ON ALGORITHM 179 INCOMPLETE BETA RATIO", N. E. Bosten and E.L. Battiste, Comm. ACM, Vol. 17, No. 3, pp. 156-7, March 1974

4. "ASYMPTOTICALLY PEARSON'S TRANSFORMATIONS", L.N. Bol'shev, Teor. Veroyat. Primen. (Theory of Probability and its Applications), Vol. 8, No. 2, pp. 129-155, 1963

5. "A NORMAL APPROXIMATION FOR BINOMIAL, F, BETA, AND OTHER COMMON, RELATED TAIL PROBABILITIES, I", D.B. Peizer and J.W. Pratt, Journal of American Statistical Association Journal, Vol. 63, pp. 1416-1456, Dec. 1968.

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

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:

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

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

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

See also

cdfn, **cdftvn**

Technical Notes

The absolute error for **cdfbvn** is approximately $\pm 5.0e{-}9$ for the entire range of arguments.

References:

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

2. "Computation of Bi- and Tri-variate Normal Integral", D.J. Daley, Appl. Statist., Vol. 23, No. 3, pp. 435-438, 1974

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 $1 - F(x, n)$, where F is the chi-square cdf with n degrees of freedom. Thus, to get the chi-square cdf, subtract **cdfchic(**x,n**)** from 1. The complement of the cdf is computed because this is what is most commonly needed in statistical applications, and because it can be computed with fewer problems of roundoff error.

Example

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

$$y = \begin{matrix} 0.991837 \\ 0.977589 \\ 0.960028 \\ 0.940242 \end{matrix}$$

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

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

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

References:

1. "ALGORITHM AS 32, THE INCOMPLETE GAMMA INTEGRAL", G.P. Bhattacharjee, Applied Statistics, Vol. 19, pp. 285-287, 1970.

2. "A NORMAL APPROXIMATION FOR BINOMIAL, *F*, BETA, AND OTHER COMMON, RELATED TAIL PROBABILITIES, I", D.B. Peizer and J.W. Pratt, Journal of American Statistical Association Journal, Vol. 63, pp. 1416-1456, Dec. 1968.

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

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-square 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));
```

$$x = \begin{matrix} 2.1096456 & 1.9354989 & 1.7549182 \\ 4.4971008 & 9.2643386 & 4.3639694 \\ 4.5737473 & 1.3706243 & 2.5653688 \end{matrix}$$

Globals

gammaii

Source

cdfchii.src

Purpose

The integral under noncentral chi-square distribution, from 0 to x. It can return a vector of values, but the degrees of freedom and noncentrality parameter must be the same for all values of x.

Format

y = **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, 1959, app. IV.

Output

y	N×1 vector, integrals from 0 to x of noncentral chi-square.

Technical Notes

- 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, p. 942, formula 26.4.25.

Example

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

```
  0.0042086234
   0.016608592
    0.30954232
    0.99441140
```

Source

cdfnonc.src

Globals

None

See also

cdffnc, cdftnc

Purpose

Computes the complement of the cdf 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.

Allowable ranges for the arguments are:

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

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

This equals $1 - G(x, n1, n2)$, where G is the F cdf with *n1* and *n2* degrees of freedom. Thus, to get the F cdf, subtract **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.

Example

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

$$y = \begin{matrix} 0.751772 \\ 0.708152 \\ 0.680365 \\ 0.659816 \end{matrix}$$

See also

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

Technical Notes

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

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

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

References:

1. "ALGORITHM 179 INCOMPLETE BETA RATIO", O.G. Ludwig, Comm. ACM, Vol. 6, No. 6, pp. 314, June 1963

2. "REMARK ON ALGORITHM 179 INCOMPLETE BETA RATIO", M.C. Pike and I.D. Hill, Comm. ACM, Vol. 10, No. 6, pp. 375-6, June. 1967

3. "REMARK ON ALGORITHM 179 INCOMPLETE BETA RATIO", N. E. Bosten and E.L. Battiste, Comm. ACM, Vol. 17, No. 3, pp. 156-7, March 1974

4. "ASYMPTOTICALLY PEARSON'S TRANSFORMATIONS", L.N. Bol'shev, Teor. Veroyat. Primen. (Theory of Probability and its Applications), Vol. 8, No. 2, pp. 129-155, 1963

5. "A NORMAL APPROXIMATION FOR BINOMIAL, F, BETA, AND OTHER COMMON, RELATED TAIL PROBABILITIES, I", D.B. Peizer and J.W. Pratt, Journal of American Statistical Association Journal, Vol. 63, pp. 1416-1456, Dec. 1968.

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

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

Purpose

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

Format

y = **cdffnc(**x,*v1*,*v2*,d**);**

Input

x	N×1 vector, values of upper limits of integrals, $x > 0$.
v1	scalar, degrees of freedom of numerator, $v1 > 0$.
v2	scalar, degrees of freedom of denominator, $v2 > 0$.
d	scalar, noncentrality parameter, $d > 0$. This is the square root of the noncentrality parameter that sometimes goes under the symbol lambda. See Scheffe, THE ANALYSIS OF VARIANCE, 1959, app. IV.

Output

y	N×1 vector of integrals from 0 to x of noncentral F.

Technical Notes

- Relation to cdffc:

  cdffc(x,v1,v2) = 1 - cdffnc(x,v1,v2,0);

- The formula used is taken from Abramowitz and Stegun, HANDBOOK OF MATHEMATICAL FUNCTIONS, 1970, p. 947, formula 26.6.20.

Source

`cdfnonc.src`

Globals

None

See also

cdftnc, **cdfchinc**

Purpose

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

$$intlim = \begin{matrix} 0.000000 \\ 0.200000 \\ 0.400000 \\ 0.600000 \\ 0.800000 \\ 1.000000 \end{matrix}$$

$$g = \begin{array}{cccc} 0.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.472911 & 0.181269 & 0.00114848 & 2.35307E-014 \\ 0.628907 & 0.329680 & 0.00792633 & 2.00981E-011 \\ 0.726678 & 0.451188 & 0.0231153 & 9.66972E-010 \\ 0.794097 & 0.550671 & 0.0474226 & 1.43310E-008 \\ 0.842701 & 0.632120 & 0.0803014 & 1.11425E-007 \end{array}$$

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. "ALGORITHM AS 32, THE INCOMPLETE GAMMA INTEGRAL", G.P. Bhattacharjee, Applied Statistics, Vol. 19, pp. 285-287, 1970.

2. "A NORMAL APPROXIMATION FOR BINOMIAL, F, BETA, AND OTHER COMMON, RELATED TAIL PROBABILITIES, I", D.B. Peizer and J.W. Pratt, Journal of American Statistical Association Journal, Vol. 63, pp. 1416-1456, Dec. 1968.

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

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**, **cdftvn**, **lncdfmvn**

Source

`lncdfn.src`

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. "ALGORITHM 304 NORMAL CURVE INTEGRAL", I.D. Hill and S.A. Joyce, Comm. ACM, Vol. 10, No. 6, pp. 374-5, June 1967

2. "REMARK ON ALGORITHM 304 NORMAL CURVE INTEGRAL", A.G. Adams, Comm. ACM, Vol. 12, No. 10, pp. 565-6, Oct. 1969

3. "REMARK ON ALGORITHM 304 NORMAL CURVE INTEGRAL", B. Holmgren, Comm. ACM, Vol. 13, No. 10, Oct. 1970

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

Purpose

Computes interval of Normal cumulative distribution function.

Format

y = **cdfn2(**x,dx**);**

Input

x	M×N matrix, abscissae.
dx	K×L matrix, ExE conformable to x, intervals.

Output

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

Remarks

The relative error is:

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

A relative error of $\pm 1e-14$ implies that the answer is accurate to better than ± 1 in the 14th digit.

Example

```
print cdfn2(1,0.5);

 9.1848052662599017e-02

print cdfn2(20,0.5);

 2.7535164718736454e-89

print cdfn2(20,1e-2);

 5.0038115018684521e-90

print cdfn2(-5,2);

 1.3496113800582164e-03

print cdfn2(-5,0.15);

 3.3065580013000255e-07
```

See also

lncdfn2

Source

`lncdfn.src`

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);
```

$$y = \begin{matrix} 0.473165 \\ 0.447100 \\ 0.422428 \\ 0.399555 \end{matrix}$$

See also

cdfbeta, **cdfchic**, **cdffc**, **cdfn**, **cdfnc**, **gamma**

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

2. "ALGORITHM 395 STUDENT'S *t*-DISTRIBUTION", G.W. Hill, Comm. ACM. Vol. 13, No. 10, Oct. 1970

3. *Handbook of Mathematical Functions*, Eds. M. Abramowitz and I. A. Stegun, Dover, New York, 7th Ed, 1970, ISBN 0-486-61272-4

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, 1959, app. IV.

Output

y	N×1 vector, integrals from $-\infty$ to x of noncentral t.

Technical Notes

- Relation to cdftc:

  cdftc(x,v) = 1 - cdftnc(x,v,0);

- The formula used is based on the formula in SUGI Supplemental Library User's Guide, 1983, SAS Institute, page 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.

  We invite any feedback from those using this function.

Source

`cdfnonc.src`

Globals

None

See also

cdffnc, **cdfchinc**

Purpose

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

Format

c = **cdftvn(***x1*,*x2*,*x3*,*rho12*,*rho23*,*rho31***);**

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.
rho12	scalar or N×1 vector of correlation coefficients between the two variables *x1* and *x2*.
rho23	scalar or N×1 vector of correlation coefficients between the two variables *x2* and *x3*.
rho31	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:

f(x1,x2,x3,rho12,rho23,rho31)

Remarks

Allowable ranges for the arguments are:

$$\begin{array}{rcl} -\infty & < x1 & < +\infty \\ -\infty & < x2 & < +\infty \\ -\infty & < x3 & < +\infty \\ -1 & < rho12 & < 1 \\ -1 & < rho23 & < 1 \\ -1 & < rho31 & < 1 \end{array}$$

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 (*rho12*,*rho23*,*rho31*) 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);
```

See also

cdfn, **cdfbvn**

Technical Notes

The absolute error for **cdftvn** is approximately ±2.5e−8 for the entire range of arguments.

References:

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

2. "Computation of Bi- and Tri-variate Normal Integral", D.J. Daley, Appl. Statist., Vol. 23, No. 3, pp. 435-438, 1974

Purpose

Returns the current directory.

Format

y = **cdir(**s**);**

Input

s	string, if the first character is 'A'-'Z' and the second character is a colon ':' then that drive will be used. If not, the current default drive will be used.

Output

y	string containing the drive and full path name of the current directory on the specified drive.

Remarks

If the current directory is the root directory, the returned string will end with a backslash, otherwise it will not.

A null string or scalar zero can be passed in as an argument to obtain the current drive and path name.

Example

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

C:\GAUSS
D:\
```

See also

files

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);
```

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

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

See also

floor, **trunc**

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

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 the Unix version.

Purpose

Computes the Cholesky decomposition of a symmetric, positive definite 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;
```

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

$$y = \begin{matrix} 9.526099 & -0.678892 & -0.202338 & -1.647690 \\ 0.000000 & 9.345890 & 0.661433 & -0.380334 \\ 0.000000 & 0.000000 & 10.074465 & 0.424211 \\ 0.000000 & 0.000000 & 0.000000 & 9.798130 \end{matrix}$$

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

See also

crout, **solpd**

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);
```

$$r = \begin{matrix} 18.87055964 & 15.32294435 & 8.04947012 \\ 0.00000000 & 9.30682813 & -2.12009339 \\ 0.00000000 & 0.00000000 & 7.62878355 \end{matrix}$$

See also

cholup

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);
```

$$x = \begin{matrix} -1.94396905 \\ -1.52686768 \\ 3.21579513 \end{matrix}$$

$$A0 = \begin{matrix} 3.00506436 & 2.65577048 & 3.08742844 \\ 2.65577048 & 3.55545737 & 3.42362593 \\ 3.08742844 & 3.42362593 & 4.02095978 \end{matrix}$$

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);
```

$$r = \begin{matrix} 20.16210005 & 16.50544413 & 9.86676135 \\ 0.00000000 & 11.16601462 & 2.97761666 \\ 0.00000000 & 0.00000000 & 11.65496052 \end{matrix}$$

See also

choldn

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 imbedding 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**

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**

Purpose

This command clears global symbols by setting them equal to scalar zero.

Format

clearg a,b,c;

Output

a,b,c	scalar global matrices containing 0.

Remarks

clearg x; is equivalent to **x = 0;**, where **x** is understood to be a global symbol. **clearg** can be used to initialize symbols not previously referenced. This command can be used inside 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**

Purpose

Close a **GAUSS** file.

Format

y = **close(***handle***);**

Input

handle	scalar, the file handle given to the file when it was opened with the **open**, **create**, or **fopen** command.

Output

y	scalar, 0 if successful, -1 if unsuccessful.

Remarks

handle is the scalar file handle created when the file was opened. It will contain an integer which can be used to refer to the file.

close will close the file specified by handle, and will return a 0 if successful and a -1 if not successful. The handle itself is not affected by **close** unless the return value of **close** is assigned to it.

If *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" 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 COMMAND 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 COMMAND 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

Purpose

Close all currently open **GAUSS** files.

Format

closeall;

closeall *list_of_handles*;

Remarks

list_of_handles is a comma-delimited list of file handles.

closeall with no specified list of handles will close all files. The file handles will not be affected. The main advantage of using **closeall** is ease of use; the file handles do not have to be specified, and one statement will close all files.

When a list of handles follows **closeall**, all files are closed and the file handles listed are set to scalar 0. This is safer than **closeall** without a list of handles because the handles are cleared.

It is important to set unused file handles to zero because both **open** and **create** check the value that is in a file handle before they proceed with the process of opening a file. During **open** or **create**, if the value that is in the file handle matches that of an already open file, the process will be aborted and a "File already open" message will be given. This gives you some protection against opening a second file with the same handle as a currently open file. If this happened, you would no longer be able to access the first file.

Files are not automatically closed when a program terminates. This allows users to run a program that opens files, and then access the files from COMMAND 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 COMMAND 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**

Purpose

Clear the screen.

Format

cls;

Portability

Unix, **OS/2**, **Windows**

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

Remarks

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

See also

locate

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)x1 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);
```

$$x1[5,1] = \begin{matrix} 0 \\ 5 \\ 10 \\ 15 \\ 20 \end{matrix}$$ (column vector of original values)

$$v[3,1] = \begin{matrix} 1 & 2 & 3 \end{matrix}$$ (Note: v is a column vector)

$$e[5,2] = \begin{matrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{matrix}$$

$$y[5,1] = \begin{matrix} 3 \\ 1 \\ 2 \\ 2 \\ 2 \end{matrix}$$

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`

Globals

None

See also

recode, **substute**

Purpose

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

Format

y = **color(**cv**)**

Input

cv	scalar, 2×1 or 3×1 vector of color values or N×4 matrix of palette color values. See **Portability** for platform specifics. If the input vector is smaller than 3×1 or the corresponding element in the input vector is -1, the corresponding color will be left unchanged. If the input is an N×4 matrix, it will initialize the VGA palette with user-defined RGB colors interpreted as follows: [N,1] palette register index 0-255 [N,2] red value 0-63 [N,3] green value 0-63 [N,4] blue value 0-63

Output

y	vector, or N×4 matrix the same size as the input which contains the original color values or palette values.

Portability

Dos

[1] pixel color
[2] text color
[3] ignored

Unix

color affects the active window. X supports foreground and background colors. The **color** command makes no distinction between text and pixel colors; both affect the foreground color of the active window. If both a pixel color and text color are specified,

the pixel color will be ignored, and the text color will be used to set the foreground color. Thus:

[1] foreground

or

[1] ignored
[2] foreground

or

[1] ignored
[2] foreground
[3] background

OS/2, **Windows**

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

Remarks

This changes the screen colors for your program's output. The editor and COMMAND mode will not be affected.

The color values 0-15 may be obtained through the help system by requesting help on '@PQG'. You will have to page down several pages to the page listing the color values.

Under DOS, the VGA color palette registers may be set only if the display adapter has been already been initialized to VGA graphics mode 19 (320×200, 256 colors) with the **setvmode** command. The registers will retain the new values until the adapter is reset to text mode, which resets the palette to the default VGA colors.

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

See also

graph, **line**, **setvmode**

Purpose

cols returns the number of columns in a matrix.

colsf returns the number of columns in a **GAUSS** data (`.dat`) file or **GAUSS** matrix (`.fmt`) file.

Format

y = **cols(**x**);**

yf = **colsf(**fh**);**

Input

x	any valid expression that returns a matrix.
fh	file handle of an open file.

Output

y	number of columns in x.
yf	number of columns in the file that has the handle fh.

Remarks

If x is an empty matrix, **rows(**x**)** and **cols(**x**)** return 0.

For **colsf**, the file must be open.

Example

```
x = rndn(100,3);
y = cols(x);
```

$y =$ 3.000000

```
create fp = myfile with x,10,4;
b = colsf(fp);
```

$b =$ 10.000000

See also

rows, **rowsf**, **show**, **lshow**

Purpose

Controls logging of COMMAND 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 COMMAND mode statements to. This can be any legal file name.

If the name of the file is to be taken from a string variable, the name of the string must be preceded by the ^ (caret) operator.

There is no default file name.

on, off, reset literal, mode command

- **on** turns on command logging to the current log file. If the file already exists, subsequent commands will be appended.
- **off** closes the log file and turns off command logging.
- **reset** similar to the **on** subcommand, except that it resets the log file by deleting any previous commands.

Portability

This function is supported only under DOS.

Remarks

COMMAND 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 on the screen.

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

Purpose

Compiles a source file to a compiled code file. See also Chapter 11.

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 DOS text file containing a **GAUSS** program. There can be references to global symbols, Run-Time Library references, etc.

If there are **library** statements in *source*, they will be used during the compilation to locate various procedures and symbols used in the program. Since all of these library references are resolved at compile time, the **library** statements are not transferred to the compiled file. The compiled file can be run without activating any libraries.

If you do not want extraneous stuff saved in the compiled image, put a **new** at the top of the *source* file or execute a **new** from COMMAND level before compiling.

The program saved in the compiled file can be run with the **run** command. If no extension is given, the **run** command will look for a file with the correct extension for the version of **GAUSS**. The **src_path** will be used to locate the file if the full path name is not given and it is not located on the current directory.

When the compiled file is **run**, all previous symbols and procedures are deleted before the program is loaded. It is therefore unnecessary to execute a **new** before **run**'ning a compiled file.

If you want line number records in the compiled file you can put a **#lineson** statement in the *source* file or turn line tracking on from the **Options** menu.

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**

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);
```

$$t = \begin{matrix} 4.0000000 + 3.0000000i & 6.0000000 + 5.0000000i \\ 9.0000000 + 1.0000000i & 8.0000000 + 7.0000000i \end{matrix}$$

See also

imag, **real**

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.

Portability

Unix, **OS/2**, **Windows**

con gets input from the active window. If you are working in terminal mode **GAUSS** will not "see" any input until you press **Enter**, so follow each entry with an **Enter**. Cursor keys are not supported in terminal mode.

Remarks

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

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

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

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

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

The matrix editor will show you what is in each element, and will prompt you for a new number. If you do not want to change anything, you can move to a different element by hitting the appropriate cursor key. If you start to type a number and then hit a cursor key, the number will not be saved. The **BACKSPACE** key will delete characters to the left of the cursor.

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

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

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

If using DOS, π and e can be entered with **Alt-P** and **Alt-E**.

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

Example

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

? 2
```

```
  −0.148030     0.861562
   1.791516   −0.663392
```

In this example, the **con** function is used to obtain the size of a square matrix of Normal random variables which is to be printed out.

See also

cons, **editm**, **let**, **load**, **medit**

Purpose

This procedure will compute 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

Globals

None

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);
```

$$x = \begin{matrix} 1.0000000 + 9.0000000i & 2.0000000 \\ 4.0000000 + 4.0000000i & 0.0000000 + 5.0000000i \\ 0.0000000 + 7.0000000i & 8.0000000 - 2.0000000i \end{matrix}$$

$$y = \begin{matrix} 1.0000000 - 9.0000000i & 2.0000000 \\ 4.0000000 - 4.0000000i & 0.0000000 - 5.0000000i \\ 0.0000000 - 7.0000000i & 8.0000000 + 2.0000000i \end{matrix}$$

Purpose

Retrieves a character string from the keyboard.

Format

x = **cons;**

Output

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

Portability

Unix

cons gets input from the active window.

Remarks

x is assigned the value of a character string typed in at the keyboard. The program will pause to accept keyboard input. The maximum length of the string that can be entered is 254 characters. The program will resume execution when the **Enter** key is pressed. The standard **DOS** editing keys will be in effect.

Example

```
x = cons;
```

At the cursor enter:

```
probability
```

x = "probability"

See also

con

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;
```

$$x = \begin{matrix} -1.032195 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & -1.033763 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.061205 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & -0.225936 \end{matrix}$$

Remarks

This command works just like in C.

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);
```

$$z1 = \begin{matrix} 5 \\ 16 \\ 34 \\ 60 \\ 61 \\ 52 \\ 32 \end{matrix}$$

$$z2 = \begin{matrix} 16 \\ 34 \\ 60 \\ 61 \end{matrix}$$

See also

polymult

Purpose

Returns the amount, in bytes, of free workspace memory.

Format

y = **coreleft;**

Output

y	scalar, number of bytes free.

Remarks

The amount of free memory is dynamic and can change rapidly as expressions and procedures are being executed. **coreleft** returns the amount of workspace memory free at the time it is called. Workspace memory is used for storing matrices, strings, procedures, and for manipulating matrices and strings.

This function can be used to write programs that automatically adjust their use of memory so they do not crash with the "Insufficient memory" error if they are used on machines with less free memory than the one used for development or if the size of the data used becomes larger. A common use is to adjust the number of rows that are read per iteration of a read loop in programs that access data from a disk.

Example

```
open fp = myfile;
k = colsf(fp);    /* columns in file */
fac = 4;
/* check amount of memory available */
nr = coreleft/(fac*k*8);
```

In this example, **nr**, the number of rows to read, is computed by taking the number of bytes free (**coreleft**) divided by **fac*k*8**. **fac** is a guesstimate of the number of copies of the data read each iteration that the algorithm we are using will require plus a little slop. **k*8** is the number of columns times the number of bytes per element.

See also

dfree, **new**

Purpose

Computes a 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**, $P = K$.

Source

`corr.src`

Globals

None

See also

momentd

Purpose

Returns the cosine of its argument.

Format

y = **cos(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix.

Remarks

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

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

Example

```
x = { 0, .5, 1, 1.5 };
y = cos(x);
```

$$y = \begin{matrix} 1.00000000 \\ 0.87758256 \\ 0.54030231 \\ 0.07073720 \end{matrix}$$

See also

atan, **atan2**, **pi**

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);
```

$$x = \begin{matrix} -1.570796 \\ -0.785398 \\ 0.000000 \\ 0.785398 \\ 1.570796 \\ 3.141593 \end{matrix}$$

$$y = \begin{matrix} 2.509178 \\ 1.324609 \\ 1.000000 \\ 1.324609 \\ 2.509178 \\ 11.591953 \end{matrix}$$

Source

trig.src

Globals

None

Purpose

Count 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:

$$\begin{aligned} x &\leq v[1], \\ v[1] < x &\leq v[2], \\ &\vdots \\ v[p-1] < x &\leq v[p]. \end{aligned}$$

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);
```

$$c = \begin{matrix} 0.0000000 \\ 2.0000000 \\ 1.0000000 \\ 2.0000000 \\ 1.0000000 \end{matrix}$$

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:

$$\begin{array}{rcccl} & & x & \leq & v[1] \\ v[1] & < & x & \leq & v[2] \\ & & \cdots & & \\ v[p-1] & < & x & \leq & v[p] \end{array}$$

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);
```

$$c = \begin{matrix} 0.000000 \\ 0.500000 \\ 0.333000 \\ 2.00000 \\ 0.100000 \end{matrix}$$

Purpose

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

Format

create ⟦*vflag*⟧ ⟦**complex**⟧ *fh* = *filename* **with** *vnames*,*col*,*dtyp*,*vtyp*;

create ⟦*vflag*⟧ ⟦**complex**⟧ *fh* = *filename* **using** *comfile*;

Input

vflag version flag.

-v89 supported on DOS, OS/2, Windows NT
-v92 supported on Unix
-v96 supported on all platforms

See the *File I/O* chapter in Volume I of the manual 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**.

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	$8.43x10^{-37}$	$\leq \|X\| \leq$	$3.37x10^{+38}$
double	15-16	$4.19x10^{-307}$	$\leq \|X\| \leq$	$1.67x10^{+308}$

If the integer type is specified, numbers will be rounded to the nearest integer as they are written to the data set. If the data to be written to the file contains character data, the precision must be 8 or the character information will be lost.

vtyp matrix, types of variables.

The types of the variables in the data set. If **rows(***vtyp***)*cols(***vtyp***)** $<$ *col* only the first element is used. Otherwise nonzero elements indicate a numeric variable and zero elements indicate character variables. *vtyp* is ignored for v89 files.

create... using...

comfile literal or ^string.

comfile is the name of a command file that contains the information needed to create the file. The default extension for the command file is `.gcf`, which can be overridden.

There are three possible commands in this file:

NUMVAR *n str***;**
OUTVAR *varlist***;**
OUTTYP *dtyp***;**

NUMVAR and **OUTVAR** are alternate ways of specifying the number and names of the variables in the data set to be created.

When **NUMVAR** is used, *n* is a constant which specifies the number of variables (columns) in the data file and *str* is a string literal specifying the prefix to be given to all the variables. Thus:

```
numvar 10 xx;
```

says that there are 10 variables and that they are to be named **xx01** through **xx10**. The numeric part of the names will be padded on the left with zeros as necessary so the names will sort correctly:

xx1,	...	xx9	1–9 names
xx01,	...	xx10	10–99 names
xx001,	...	xx100	100–999 names
xx0001,	...	xx1000	1000–8100 names

If *str* is omitted, the variable prefix will be "X".

When **OUTVAR** is used, *varlist* is a list of variable names, separated by spaces or commas. For instance:

```
outvar x1, x2, zed;
```

specifies that there are to be 3 variables per row of the data set, and that they are to be named **X1**, **X2**, **ZED**, in that order.

OUTTYP specifies the precision. It can be a constant: 2, 4, or 8, or it can be a literal: I, F, or D. See *dtyp* above in the discussion of **create... with...** for an explanation of the available data types.

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.

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;
```

The example above 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 above program 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, then 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**

For the next example we will assume a command file called `comd.gcf` containing the following lines has been 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

open, **readr**, **writer**, **eof**, **close**, **output**, **iscplxf**

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);
```

$$z = \begin{matrix} 7.0000000 & -39.000000 \\ -28.000000 & 107.00000 \\ 17.000000 & -95.000000 \end{matrix}$$

Source

crossprd.src

Globals

None

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**

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)xN matrix containing the lower (L) and upper (U) matrices of the Crout decomposition of a permuted x. The N+1 row of the matrix y gives the row order of the y matrix. The matrix must be reordered prior to extracting the L and U matrices. Use **lowmat** and **upmat1** to extract the L and U matrices from the reordered y matrix.

Example

This example illustrates a procedure for extracting L and U of the permuted x matrix. It continues by sorting the result of LU to compare with the original matrix x.

```
X = { 1  2 -1,
      2  3 -2,
      1 -2  1 };

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

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

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

$r = 4$

$$indx = \begin{matrix} 2 \\ 3 \\ 1 \end{matrix}$$

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

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

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

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

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

See also

crout, **chol**, **lowmat**, **lowmat1**, **lu**, **upmat**, **upmat1**

Purpose

Returns the position of the cursor.

Format

y = **csrcol;**

y = **csrlin;**

Portability

Unix

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

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

OS/2, Windows

csrcol returns the cursor column with respect to the current output line, i.e., it will return the same value whether the text is wrapped or not. **csrlin** returns the cursor line with respect to the top line in the window.

Remarks

y will contain the current column or row position of the cursor on the screen. The upper left corner is (1,1). Under DOS, columns are usually numbered 1–80, rows are usually numbered 1–25. **setvmode** will return the current screen dimensions.

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

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

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**, **lpos**, **setvmode**

Purpose

To set the cursor shape.

Format

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

Portability

Unix

This function is not supported in terminal mode.

OS/2, **Windows**

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

Input

mode scalar, cursor type to set.

DOS

0	cursor off
1	normal cursor
2	large cursor

Unix

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

Output

old scalar, original cursor type.

Remarks

Under DOS, this function will set the same shape as **GAUSS** is already using for its three modes. See the *Configuration* chapter in the DOS supplement for details.

Example

```
x = csrtype(2);
```

See also

csrcol, **csrlin**

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);
```

$$y = \begin{matrix} 1.00 & -3.00 \\ 2.00 & -6.00 \\ 6.00 & 6.00 \end{matrix}$$

Source

`cumprodc.src`

Globals

None

See also

cumsumc, **recsercp**, **recserar**

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`

Globals

None

See also

cumprodc, **recsercp**, **recserar**

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

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**

Purpose

List selected variables from a data set.

Format

datalist *dataset* [[*var1* [[*var2*...]]];

Input

dataset	literal, name of the dataset.
var#	literal, the names of the variables to list.
__range	global scalar, the range of rows to list. The default is all rows.
__miss	global scalar, controls handling of missing values.

0 display rows with missing values.

1 do not display rows with missing values.

The default is 0.

__prec	global scalar, the number of digits to the right of the decimal point to display. The default is 3.

Remarks

The variables are listed in an interactive mode. As many rows and columns as will fit 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`

Purpose

Returns the current date in a 4-element column vector, in the order: year, month, day, and hundredths of a second since midnight.

Format

y = **date;**

Remarks

The hundredths of a second since midnight can be accessed using **hsec**.

Example

```
print date;

  1998.0000
  6.0000000
  15.000000
  4011252.7
```

See also

time, **timestr**, **ethsec**, **hsec**, **etstr**

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 = { 1998, 6, 15, 0 };
y = datestr(d);
print y;
```

```
  6/15/98
```

Source

`time.src`

Globals

None

See also

date, **datestring**, **datestrymd**, **time**, **timestr**, **ethsec**

Purpose

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

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;
```

```
  6/15/1996
```

Source

`time.src`

Globals

None

See also

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

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 = { 1998, 6, 15, 0 };
y = datestrymd(d);
print y;
```

```
  19980615
```

Source

`time.src`

Globals

None

See also

date, **datestr**, **datestring**, **time**, **timestr**, **ethsec**

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 = { 1998, 6, 15, 0 };
y = dayinyr(x);
print y;
```

```
  166.00000
```

Source

time.src

Globals

_isleap

Purpose

Runs a program under the source level debugger.

Format

debug *filename*;

Input

filename The name of the program to debug.

Purpose

To initialize matrices and strings at compile time.

Format

declare [[*type*]] *symbol* [[*aop clist*]]**;**

Input

type optional literal, specifying the type of the symbol.

matrix
string

if *type* is not specified, **matrix** is assumed.

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.

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 };
```

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

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

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

```
declare matrix x[3,3] = 1;
```

$$x = \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

```
declare matrix x[3,3];
```

$$x = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

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

$$x = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{matrix}$$

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

$$x = \begin{matrix} \text{DOG} \\ \text{CAT} \end{matrix}$$

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

$$x = \begin{matrix} \text{dog} \\ \text{cat} \end{matrix}$$

Remarks

The **declare** syntax is similar to the **let** statement.

declare generates no executable code. This is strictly for compile time initialization. The data on the right-hand side of the equal sign must be constants. No expressions or variables are allowed.

declare statements are intended for initialization of global matrices and strings that are used by procedures in a library system.

It is best to place **declare** statements in a separate file from procedure definitions. This will prevent redefinition errors when rerunning the same program without clearing your workspace.

Complex numbers can be entered by joining the real and imaginary parts with a sign (+ or -); there can be no spaces between the numbers and the sign. Numbers with no real part can be entered by appending an 'i' to the number.

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.

Ctrl-W controls the **declare** statement warning level. If **declare** warnings are on, you will be warned whenever a **declare** statement encounters a symbol that is already initialized. 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.

See also

let, **external**

Purpose

Deletes global symbols from the symbol table.

Format

delete -*flags symbol1,symbol2,symbol3*;

Remarks

This completely and irrevocably deletes a symbol from **GAUSS**'s memory and workspace. It will no longer show up when the **show** command is used.

The following flags are defined:

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

Flags must be preceded by a slash (e.g. **-pfk**) and are defined in the same way as the **show** command. If symbol names are ended with an asterisk, any names matching those first letters will be deleted. If the **n** (no pause) flag is used, you will not be asked for confirmation for each symbol.

This command is supported only from COMMAND level. Since the interpreter executes a compiled pseudo-code, this command would invalidate a previously compiled code image and therefore would destroy any program it was a part of. If any symbols are deleted, all procedures, keywords and functions with global references will be deleted as well.

Example

```
print x;

  96.000000
  6.0000000
  14.000000
  3502965.9

delete -m x;
```

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

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);
```

Here is the resulting matrix **y**:

```
  0   10   20
```

All rows for which the elements in column 1 are greater than 0 and the elements in column 3 are less than 100 are deleted.

Source

datatran.src

Globals

None

See also

selif

Purpose

Returns dense submatrix of sparse matrix.

Format

c = **denseSubmat(**x,r,c**);**

Input

x	M×N sparse matrix.
r	K×1 vector, row indices.
c	L×1 vector, column indices.

Output

e	K×L dense matrix.

Remarks

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

Source

`sparse.src`

See also

sparseFd, **sparseFp**

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);
```

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

Source

design.src

Globals

None

See also

cumprodc, **cumsumc**, **recserrc**

Purpose

Returns the determinant of a square matrix.

Format

y = **det(**x**);**

Input

x	N×N square matrix.

Output

y	determinant of x.

Remarks

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

det computes a LU decomposition.

detl can be much faster in many applications.

Example

```
x = { 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

Purpose

Returns the determinant of the last matrix that was passed to one of the intrinsic matrix decomposition routines.

Format

y = **detl;**

Remarks

Whenever one of the following functions is executed, the determinant of the matrix is also computed and stored in a system variable. This function will return the value of that determinant and, because the value has been computed in a previous instruction, this will require no computation.

The following functions will set the system variable used by **detl**:

chol(x**)**	
crout(x**)**	
croutp(x**)**	
det(x**)**	
inv(x**)**	
invpd(x**)**	
solpd(y**,**x**)**	determinant of x
y/x	determinant of x when neither argument is a scalar or determinant of $x'x$ if x is not square

Example

If both the inverse and the determinant of the matrix are needed, the following two commands will return both with the minimum amount of computation:

```
xi = inv(x);
xd = detl;
```

The function **det(**x**)** returns the determinant of a matrix using the Crout decomposition. If you only want the determinant of a positive definite matrix, the following code will be the fastest for matrices larger than 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

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`

Globals

None

See also

dffti, **fft**, **ffti**

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`

Globals

None

See also

fft, **dffti**, **ffti**

Purpose

Returns the amount of room left on a diskette or hard disk.

Format

y = **dfree(***drive***);**

Input

drive	scalar, valid disk drive number.

Output

y	number of bytes free.

Portability

Unix

The **dfree** function is not supported in Unix.

Remarks

Valid disk drive numbers are 0 = default, 1 = A, 2 = B, etc. If an error is encountered, **dfree** will return -1.

See also

coreleft

Purpose

Creates a column vector from the diagonal of a matrix.

Format

y = **diag(**x**);**

Input

x	N×K matrix.

Output

y	min(N,K)x1 vector.

Remarks

The matrix need not be square.

diagrv reverses the procedure and puts a vector into the diagonal of a matrix.

Example

```
x = rndu(3,3);
y = diag(x);
```

$$x = \begin{matrix} 0.660818 & 0.367424 & 0.302208 \\ 0.204800 & 0.077357 & 0.145755 \\ 0.712284 & 0.353760 & 0.642567 \end{matrix}$$

$$y = \begin{matrix} 0.660818 \\ 0.077357 \\ 0.642567 \end{matrix}$$

See also

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) 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);
```

$$x = \begin{matrix} 0.660818 & 0.367424 & 0.302208 \\ 0.204800 & 0.077357 & 0.145755 \\ 0.712284 & 0.353760 & 0.642567 \end{matrix}$$

$$v = \begin{matrix} 1.000000 \\ 1.000000 \\ 1.000000 \end{matrix}$$

$$y = \begin{matrix} 1.000000 & 0.367424 & 0.302208 \\ 0.204800 & 1.000000 & 0.145755 \\ 0.712284 & 0.353760 & 1.000000 \end{matrix}$$

See also

diag

Purpose

Disables the invalid operation interrupt of the numeric processor. This affects the way missing values are handled in most calculations.

Format

disable;

Portability

Unix, **OS/2**, **Windows**

This function is not used by these platforms. The invalid operation is always disabled

Remarks

When **disable** is in effect, missing values will be allowed in most calculations. A missing value is a special floating point encoding which the numeric processor considers a NaN (*Not A Number*). **disable** allows missing values to pass through most calculations unchanged, i.e., a number plus a missing value is a missing value.

The default when **GAUSS** is started is to have the program crash when missing values are encountered or when any operation sets the numeric processor's invalid operation exception. See the *Debugger* or *Error Handling and Debugging* chapter in your supplement.

If **disable** is on, these operations will return a NaN, and the program will continue. This can complicate debugging for programs that do not need to handle missing values, because the program may proceed far beyond the point that NaN's are created before it actually crashes.

The opposite of **disable** is **enable**, which is the default. If **enable** is on, the program will terminate with an "Invalid floating point operation" error message.

The following operators are specially designed to handle missing values and are not affected by the **disable/enable** commands: b/a (matrix division when a is not square and neither a nor b is scalar), **counts**, **ismiss**, **maxc**, **maxindc**, **minc**, **minindc**, **miss**, **missex**, **missrv**, **moment**, **packr**, **scalmiss**, **sortc**.

ndpcntrl can be used to get and reset the numeric processor control word, so it is more flexible than **enable/disable.**

See also

enable, **miss**, **missrv**, **ndpchk**, **ndpcntrl**

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.

```
format /rd 1,0;
i = 1;
do while i <= 10;
    print i;;
    i = i+1;
endo;
i = 1;
do until i > 10;
    print i;;
    i = i+1;
endo;
```

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

See also

continue, **break**

Purpose

Provides access to the operating system from within **GAUSS**.

Format

dos [[*s*]];

Portability

Unix

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

This function may be used in terminal mode.

OS/2, Windows

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

Running programs in the background is allowed in all three of the aforementioned platforms.

Input

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

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 **DOS** command has been executed.

All matrices are retained in memory when the OS is accessed in this way. This command allows the use of word processing, communications, and other programs from within **GAUSS**.

Do not execute programs that terminate and remain resident. This is because they will be left resident inside of **GAUSS**'s workspace. Some examples are programs that create ramdisks or print spoolers.

If the command is to be taken from a string variable, the ^ (caret) must precede the string.

The shorthand ">" can be used in place of the word "DOS".

Example

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

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

```
>dir *.prg;
```

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

```
dos;
```

This will cause a second level OS shell to be entered. The OS prompt will appear and OS commands or other programs can be executed. To return to **GAUSS**, type **exit**.

See also

exit, **exec**

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.
_fcmptol	global scalar, comparison tolerance. The default value is $1.0e-15$.

Output

y	max(N,L) by max(K,M) matrix of 1's and 0's.

Remarks

The return value is 1 if true and 0 if false.

The statement:

```
y = dotfeq(a,b);
```

is equivalent to:

```
y = a .eq b;
```

The calling program can reset **_fcmptol** before calling these procedures.

```
_fcmptol = 1e-12;
```

Example

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

$$x = \begin{matrix} 0.85115559 & 0.98914218 \\ 0.12703276 & 0.43365175 \end{matrix}$$

$$y = \begin{matrix} 0.41907226 & 0.49648058 \\ 0.58039125 & 0.98200340 \end{matrix}$$

$$t = \begin{matrix} 1.0000000 & 1.0000000 \\ 0.0000000 & 0.0000000 \end{matrix}$$

Source

fcompare.src

Globals

_fcmptol

See also

feq

Purpose

Compute descriptive statistics.

Format

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

Input

dataset string, name of data set.

If dataset 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.
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.

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

N×K matrix the data on which to compute the descriptive statistics.

Defaults are provided for the following global input variables, so they can be ignored unless you need control over the other options provided by this procedure.

__altnam global matrix, default 0.

This can be a K×1 character vector of alternate variable names for the output.

__miss global scalar, default 0.

0	there are no missing values (fastest).
1	listwise deletion, drop a row if any missings occur in it.
2	pairwise deletion.

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

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

__output global 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`

Globals

__output, **_dstatd**, **_dstatx**

Purpose

Normalizes a date and time 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 = { 1996 14 21 6 21 37 0 0 };
d = dtvnormal(x);
```

$d =$ 1997.00 2.00 21.00 6.00 21.00 37.00 2.00 51.00

Globals

None.

See also

date, **ethsec**, **etstr**, **time**, **timestr**, **timeutc**, **utctodtv**

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:

$$\begin{array}{rcrcl} & & x & \leq & 1 \\ 1 & < & x & \leq & 5 \\ 5 & < & x & \leq & 7 \\ 7 & < & x & & \end{array}$$

Source

`datatran.src`

Globals

None

See also

dummybr, **dummydn**

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:

$$\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}$$

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

$$\begin{array}{ccccc} & & x & \leq & 1 \\ 1 & < & x & \leq & 5 \\ 5 & < & x & \leq & 7 \end{array}$$

Source

`datatran.src`

Globals

None

See also

dummydn, **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, 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:

$$\begin{array}{ccccc} & & x & \leq & 1 \\ 1 & < & x & \leq & 5 \\ 5 & < & x & \leq & 7 \\ 7 & < & x & & \end{array}$$

Since $p = 2$, the second dummy is dropped.

Source

`datatran.src`

Globals

None

See also

dummy, **dummybr**

Purpose

To access an alternate editor.

Format

ed *filename*;

Input

filename The name of the file to be edited.

Remarks

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

ed = *editor_name flags*;

or

ed = "*editor_name flags*";

The flags are any command line flags you may want between the name of the editor and the filename when your editor is invoked. The quoted version will prevent the flags, if any, from being forced to uppercase.

This command can be placed in the `startup` file so it will be set for you automatically when you start **GAUSS**.

Purpose

To edit a disk file.

Format

edit *filename*;

Remarks

Example

```
edit b:test1;
```

See also

run, **medit**, **editm**

Purpose

To edit a matrix. See **medit** for a full-screen matrix editor.

Format

y = **editm(**x**);**

Portability

Unix

editm is not supported.

DOS

π and e can be entered with **Alt-P** and **Alt-E**.

Input

x	any legal expression that returns a matrix.

Output

y	edited matrix.

Remarks

A matrix editor is invoked when the **editm** function is called. This editor allows you to move around the matrix you are editing and make changes. (This editor is also used by the **con** function.)

When **editm** appears in a program, the following will appear on the screen:

```
[1,1] = 1.2361434675434E+002 ?
```

The number after the equal sign is the [1,1] element of the matrix being edited.

There are two general ways to move around the matrix. First, you can simply type numbers separated by commas, spaces or carriage returns. The editor will automatically move you left to right and down through the matrix. That is, you will first go across the first row left to right, then across the second row, and so on. When you reach the last element in the matrix, you will automatically cycle back to the first element. When this occurs, the editor's prompt will again be displayed on the screen.

To get out of the editor, type a semicolon. If a semicolon is typed after you have entered a number, that number will be saved.

The second general way to move around the matrix is to use the cursor keys. The left and right cursor keys move you back and forth along rows. The up and down cursor keys move you up and down in a column. When you come to the end of a row or column, movement is left and up, or right and down. If, for instance, you are moving left to right along a row using the right cursor key, you will move down to the beginning of the next row when you come to the end of the row you are in. **GAUSS** will beep at you if you try to move outside the matrix.

The question mark you see on the screen is the matrix editor's prompt. If you want to change the number that is in the [1,1] position of the matrix, just type in the number you want. If you do not want to change anything, you can move to a different element by hitting the appropriate cursor key. If you start to type a number and then hit a cursor key, the number will not be saved. The **BACKSPACE** key will delete characters to the left of the cursor.

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

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

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

An **editm** function can appear anywhere in an expression that any other function can appear. Thus, for instance, the following statements are legal:

```
y = x*editm(z);

y = sqrt(editm(x));
```

Example

```
y = editm(ones(2,2));

[1,1] =  1.000000000000E+000  ? 2 3 4 5[space]

[1,1] =  2.000000000000E+000  ? 6 7 8 9[;]
```

$$y = \begin{matrix} 6.000000 & 7.000000 \\ 8.000000 & 9.000000 \end{matrix}$$

In this example, a 2×2 matrix of 1's is edited. Spaces are used to separate the numbers as they are entered. Note that after 4 numbers have been entered, the editor has cycled

back to the [1,1] position in the matrix again. When this occurs, the prompt appears on the screen again. This time, after 4 more numbers have been entered, a semicolon is entered (as indicated in the brackets). This causes the editor to stop and the edited matrix to be returned.

See also

con, **medit**

Purpose

Computes the eigenvalues of a general matrix.

Format

va = **eig(***x***);**

Input

x	N×N matrix.

Output

va	N×1 vector, the eigenvalues of *x*.

Remarks

If the eigenvalues cannot all be determined, *va*[1] is set to an error code. Passing *va*[1] to the **scalerr** function will return the index of the eigenvalue that failed. The eigenvalues for indices **scalerr**(*va*[1])+1 to N should be correct.

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

trap 0	set *va*[1] and terminate with message
trap 1	set *va*[1] and continue execution

The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.

Example

```
x = { 4 8 1,
      9 4 2,
      5 5 7 };
va = eig(x);
```

$$va = \begin{matrix} -4.4979246 \\ 14.475702 \\ 5.0222223 \end{matrix}$$

See also

eigh, **eighv**, **eigv**

Purpose

Computes the eigenvalues of a complex, general matrix. (Included for backwards compatibility—please use **eig** instead.)

Format

{ *var*,*vai* } = **eigcg(***xr*,*xi***);**

Input

xr	N×N matrix, real part.
xi	N×N matrix, imaginary part.

Output

var	N×1 vector, real part of eigenvalues.
vai	N×1 vector, imaginary part of eigenvalues.
_eigerr	global scalar, if all the eigenvalues can be determined **_eigerr** = 0, otherwise **_eigerr** is set to the index of the eigenvalue that failed. The eigenvalues for indices **_eigerr**+1 to N should be correct.

Remarks

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

trap 0	set **_eigerr** and terminate with message
trap 1	set **_eigerr** and continue execution

The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.

Source

eigcg.src

Globals

_eigerr

See also

eigcg2, **eigch**, **eigrg**, **eigrs**

Purpose

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

Format

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

Input

xr	N×N matrix, real part.
xi	N×N matrix, imaginary part.

Output

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

Remarks

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

trap 0	set **_eigerr** and terminate with message
trap 1	set **_eigerr** and continue execution

The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first. The columns of *ver* and *vei* contain the real and imaginary eigenvectors of x in the same order as the eigenvalues. The eigenvectors are not normalized.

Source

eigcg.src

Globals

_eigerr

See also

eigcg, **eigch**, **eigrg**, **eigrs**

Purpose

Computes the eigenvalues of a complex, hermitian matrix. (Included for backwards compatibility—please use **eigh** instead.)

Format

va = **eigch(***xr***,***xi***);**

Input

xr	N×N matrix, real part.
xi	N×N matrix, imaginary part.

Output

va	N×1 vector, real part of eigenvalues.
_eigerr	global scalar, if all the eigenvalues can be determined **_eigerr** = 0, otherwise **_eigerr** is set to the index of the eigenvalue that failed. The eigenvalues for indices 1 to **_eigerr**−1 should be correct.

Remarks

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

trap 0	set **_eigerr** and terminate with message
trap 1	set **_eigerr** and continue execution

The eigenvalues are in ascending order. The eigenvalues for a complex hermitian matrix are always real so this procedure returns only one vector.

Source

`eigch.src`

Globals

_eigerr

See also

eigch2, **eigcg**, **eigrg**, **eigrs**

Purpose

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

Format

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

Input

xr	N×N matrix, real part.
xi	N×N matrix, imaginary part.

Output

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

Remarks

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

trap 0	set **_eigerr** and terminate with message
trap 1	set **_eigerr** and continue execution

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

Source

`eigch.src`

Globals

_eigerr

See also

eigch, **eigcg**, **eigrg**, **eigrs**

Purpose

Computes the eigenvalues of a complex hermitian or real symmetric matrix.

Format

va = **eigh(***x***);**

Input

x	N×N matrix.

Output

va	N×1 vector, the eigenvalues of *x*.

Remarks

If the eigenvalues cannot all be determined, *va*[1] is set to an error code. Passing *va*[1] to the **scalerr** function will return the index of the eigenvalue that failed. The eigenvalues for indices 1 to **scalerr**(*va*[1])-1 should be correct.

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

trap 0	set *va*[1] and terminate with message
trap 1	set *va*[1] and continue execution

The eigenvalues are in ascending order.

The eigenvalues of a complex hermitian or real symmetric matrix are always real.

See also

eig, **eighv**, **eigv**

Purpose

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

Format

{ *va*,*ve* } = **eighv**(*x*);

Input

x	N×N matrix.

Output

va	N×1 vector, the eigenvalues of *x*.
ve	N×N matrix, the eigenvectors of *x*.

Remarks

If the eigenvalues cannot all be determined, *va*[1] is set to an error code. Passing *va*[1] to the **scalerr** function will return the index of the eigenvalue that failed. The eigenvalues for indices 1 to **scalerr**(*va*[1])-1 should be correct. The eigenvectors are not computed.

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

trap 0	set *va*[1] and terminate with message
trap 1	set *va*[1] and continue execution

The eigenvalues are in ascending order. The columns of *ve* contain the eigenvectors of *x* in the same order as the eigenvalues. The eigenvectors are orthonormal.

The eigenvalues of a complex hermitian or real symmetric matrix are always real.

See also

eig, **eigh**, **eigv**

Purpose

Computes the eigenvalues of a real, general matrix. (Included for backwards compatibility—please use **eig** instead.)

Format

{ *var*,*vai* } = **eigrg**(*x*);

Input

x	N×N matrix.

Output

var	N×1 vector, real part of eigenvalues.
vai	N×1 vector, imaginary part of eigenvalues.
_eigerr	global scalar, if all the eigenvalues can be determined **_eigerr** = 0, otherwise **_eigerr** is set to the index of the eigenvalue that failed. The eigenvalues for indices **_eigerr**+1 to N should be correct.

Remarks

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

trap 0	set **_eigerr** and terminate with message
trap 1	set **_eigerr** and continue execution

The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first.

Example

```
x = { 1 2i 3,
      4i 5+3i 6,
      7 8 9i };
{y,n} = eigrg(x);
```

$$y = \begin{matrix} -6.3836054 \\ 2.0816489 \\ 10.301956 \end{matrix}$$

$$n = \begin{matrix} 7.2292503 \\ -1.4598755 \\ 6.2306252 \end{matrix}$$

Source

`eigrg.src`

Globals

_eigerr

See also

eigrg2, **eigcg**, **eigch**, **eigrs**

Purpose

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

Format

{ *var*,*vai*,*ver*,*vei* } = **eigrg2**(*x*);

Input

x	N×N matrix.

Output

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

Remarks

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

trap 0	set **_eigerr** and terminate with message
trap 1	set **_eigerr** and continue execution

The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first. The columns of *ver* and *vei* contain the real and imaginary eigenvectors of *x* in the same order as the eigenvalues. The eigenvectors are not normalized.

Source

`eigrg.src`

Globals

_eigerr

See also

eigrg, **eigcg**, **eigch**, **eigrs**

Purpose

Computes the eigenvalues of a real, symmetric matrix. (Included for backwards compatibility—please use **eigh** instead.)

Format

va = **eigrs(***x***);**

Input

x	N×N matrix.

Output

va	N×1 vector, eigenvalues of *x*.
_eigerr	global scalar, if all the eigenvalues can be determined **_eigerr** = 0, otherwise **_eigerr** is set to the index of the eigenvalue that failed. The eigenvalues for indices 1 to **_eigerr**−1 should be correct.

Remarks

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

trap 0	set **_eigerr** and terminate with message
trap 1	set **_eigerr** and continue execution

The eigenvalues are in ascending order. The eigenvalues for a real symmetric matrix are always real so this procedure returns only one vector.

Source

`eigrs.src`

Globals

_eigerr

See also

eigrs2, **eigcg**, **eigch**, **eigrg**

Purpose

Computes eigenvalues and eigenvectors of a real, symmetric matrix. (Included for backwards compatibility—please use **eighv** instead.)

Format

{ *va*,*ve* } = **eigrs2**(*x*);

Input

x	N×N matrix.

Output

va	N×1 vector, eigenvalues of *x*.
ve	N×N matrix, eigenvectors of *x*.
_eigerr	global scalar, if all the eigenvalues can be determined **_eigerr** = 0, otherwise **_eigerr** is set to the index of the eigenvalue that failed. The eigenvalues and eigenvectors for indices 1 to **_eigerr**−1 should be correct.

Remarks

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

trap 0	set **_eigerr** and terminate with message
trap 1	set **_eigerr** and continue execution

The eigenvalues are in ascending order. The columns of *ve* contain the eigenvectors of *x* in the same order as the eigenvalues. The eigenvectors are orthonormal.

The eigenvalues and eigenvectors for a real symmetric matrix are always real so this procedure returns only the real parts.

Source

eigrs.src

Globals

_eigerr

See also

eigrs, **eigcg**, **eigch**, **eigrg**

Purpose

Computes the eigenvalues and eigenvectors of a general matrix.

Format

{ *va*,*ve* } = **eigv**(*x*);

Input

x	N×N matrix.

Output

va	N×1 vector, the eigenvalues of *x*.
ve	N×N matrix, the eigenvectors of *x*.

Remarks

If the eigenvalues cannot all be determined, *va*[1] is set to an error code. Passing *va*[1] to the **scalerr** function will return the index of the eigenvalue that failed. The eigenvalues for indices **scalerr**(*va*[1])+1 to N should be correct. The eigenvectors are not computed.

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

trap 0	set *va*[1] and terminate with message
trap 1	set *va*[1] and continue execution

The eigenvalues are unordered except that complex conjugate pairs of eigenvalues will appear consecutively with the eigenvalue having the positive imaginary part first. The columns of *ve* contain the eigenvectors of *x* in the same order as the eigenvalues. The eigenvectors are not normalized.

Example

```
x = { 4 8 1,
      9 4 2,
      5 5 7 };
{y,n} = eigv(x);
```

$$y = \begin{matrix} -4.4979246 \\ 14.475702 \\ 5.0222223 \end{matrix}$$

$$n = \begin{matrix} -0.66930459 & -0.64076622 & -0.40145623 \\ 0.71335708 & -0.72488533 & -0.26047487 \\ -0.019156716 & -0.91339349 & 1.6734214 \end{matrix}$$

See also

eig, **eigh**, **eighv**

Purpose

Enables the invalid operation interrupt of the numeric processor. This affects the way missing values are handled in most calculations.

Format

enable;

Portability

Unix, **OS/2**, **Windows**

Invalid operation is always disabled. **enable** is not supported by these platforms.

Remarks

When **enable** is in effect, missing values will not be allowed in most calculations. A missing value is a special floating point encoding which the numeric processor considers a **NaN** (*Not A Number*).

The default, when **GAUSS** is started, is to have the program crash when missing values are encountered or when any operation sets the numeric processor's invalid operation exception. See the *Debugger* or *Error Handling and Debugging* chapter in your supplement.

If **enable** is on, these operations will cause the program to terminate with an "Invalid floating point operation" error message.

The opposite of **enable** is **disable**. If **disable** is on these operations will return a NaN, and the program will continue. This can complicate debugging for programs that do not need to handle missing values, because the program may proceed far beyond the point that NaN's are created before it actually crashes.

The following operators are specially designed to handle missing values and are not affected by the **disable/enable** commands: b/a (matrix division when a is not square), **counts**, **ismiss**, **maxc**, **maxindc**, **minc**, **minindc**, **miss**, **missex**, **missrv**, **moment**, **packr**, **scalmiss**, **sortc**.

ndpcntrl can be used to get and reset the numeric processor control word, and is more flexible than **enable/disable.**

See also

ndpcntrl, **disable**, **miss**, **missrv**, **ndpchk**

Purpose

Terminate a program.

Format

end;

Portability

Unix

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

OS/2, Windows

end closes all user-created windows.

DOS

COMMAND mode requires the screen to be in text mode, so it will be reset if it is in graphics mode. The screen will revert to text mode whenever it drops to COMMAND mode.

Remarks

end causes **GAUSS** to revert to COMMAND mode, and closes all open files. **end** also closes the auxiliary output file and turns the screen 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 screen 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 screen 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 screen will be turned back on.

See also

new, **stop**, **system**

Purpose

Close a procedure or keyword definition.

Format

endp;

Remarks

endp marks the end of a procedure definition that began with a **proc** or **keyword** statement. See Chapter 9 for details on writing and using procedures.

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);
```

$$x = \begin{matrix} 1.00000000 & 3.00000000 & 2.00000000 \\ 7.00000000 & 4.00000000 & 9.00000000 \\ 1.00000000 & 1.00000000 & 6.00000000 \\ 3.00000000 & 3.00000000 & 2.00000000 \end{matrix}$$

$$y = \begin{matrix} 3.00000000 \\ 5.00000000 \\ 2.00000000 \\ 7.00000000 \end{matrix}$$

$$b = \begin{matrix} 0.15456890 \\ 1.50276345 \\ -0.12840825 \end{matrix}$$

See also

proc, **keyword**, **retp**

Purpose

Searches the environment for a defined name.

Format

y = **envget(**s**);**

Input

s	string, the name to be searched for.

Output

y	string, the string that corresponds to that name in the environment or a null string if it is not found.

Example

```
proc dopen(file);
    local fname,fp;
    fname = envget("DPATH");
    if fname $== "";
        fname = file;
    else;
        if strsect(fname,strlen(fname),1) $== "\\";
            fname = fname $+ file;
        else;
            fname = fname $+ "\\" $+ file;
        endif;
    endif;
    open fp = ^fname;
    retp(fp);
endp;
```

This is an example of a procedure which will open a data file using a path stored in an environment string called DPATH. The procedure returns the file handle and is called as follows:

fp = dopen("myfile");

See also

cdir

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 (**x**′**x**) matrix of the data. On each iteration of the loop, 100 additional rows of data are read in and the moment matrix for this set of rows is computed, and added to the matrix **xx**. When all the data have been read, **xx** will contain the entire moment matrix for the data set.

GAUSS will keep reading until **eof(f1)** returns the value 1, which it will when the end of the data set has been reached. On the last iteration of the loop, all remaining observations are read in if there are 100 or fewer left.

See also

open, **readr**, **seekr**

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.

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.

Globals

_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;

__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.

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);
```

```
================================================================================
 EqSolve Version 3.2.22                                  2/24/97   9:54 am
================================================================================

||F(X)|| at final solution:                                        0.93699762
--------------------------------------------------------------------------------
Termination Code = 1:

Norm of the scaled function value is less than __Tol;
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
VARIABLE            START                 ROOTS              F(ROOTS)
--------------------------------------------------------------------------------
X1                -1.00000              0.54144351        4.4175402e-06
X2                12.00000               1.4085912       -6.6263102e-06
X3                -1.00000               1.1111111        4.4175402e-06
--------------------------------------------------------------------------------
```

Source

`eqsolve.src`

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:

$$x \quad \geq 0$$

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

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

Example

```
x = { .5 .4 .3,
      .6 .8 .3 };
y = erf(x);
```

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

```
x = { .5 .4 .3,
      .6 .8 .3 };
y = erfc(x);
```

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

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.

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**

Purpose

Prints an error message to the screen and error log file.

Format

errorlog *str*;

Input

str string, the error message to print.

Portability

Unix

errorlog prints to window 1 and the error log file.

OS/2, Windows

errorlog prints to the main **GAUSS** window and the error log file.

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 400.

Example

```
let date1 = 1986 1 2;
let date2 = 1987 10 25;
d = etdays(date1,date2);
```

$d = 661$

Source

time.src

Globals

_daypryr, **dayinyr**

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 400.

Example

```
let date1 = 1986 1 2 0;
let date2 = 1987 10 25 0;
t = ethsec(date1,date2);
```

$t = 5711040000$

Source

time.src

Globals

_daypryr, **dayinyr**

Purpose

Formats an elapsed time measured in hundredths of a second to a string.

Format

str = **etstr(***tothsecs***);**

Input

tothsecs	scalar, an elapsed time measured in hundredths of a second, as given, for instance, by the **ethsec** function.

Output

str	string containing the elapsed time in the form: # days # hours # minutes #.## seconds

Example

```
d1 = { 86, 1, 2, 0 };
d2 = { 86, 2, 5, 815642 };
t = ethsec(d1,d2);
str = etstr(t);
```

$t = 2.9457564e+08$

str = 34 *days* 2 *hours* 15 *minutes* 56.42 *seconds*

Source

time.src

Globals

None

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("freq.dat","rout",30);
```

$n =$ 30

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

Source

`exctsmpl.src`

Globals

None

Purpose

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

Format

y = **exec(***program*,*comline*);

Portability

Unix

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

Windows

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

OS/2

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

DOS

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

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

Input

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

Output

y the exit code returned by *program*.

If **exec** 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

Remarks

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

Example

```
y = exec("atog","comd1.cmd");
if y;
    errorlog "atog failed";
    end;
endif;
```

In this example the **atog** ASCII conversion utility is executed under the **exec** function. The name of the command file to be used, **comd1.cmd**, is passed to **atog** on its command line. The exit code **y** returned by **exec** is tested to see if **atog** was successful; if not, the program will be terminated after printing an error message.

Purpose

Calculates the exponential function.

Format

y = **exp(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix containing e, the base of natural logs, raised to the powers given by the elements of x.

Example

```
x = eye(3);
y = exp(x);
```

$$x = \begin{matrix} 1.000000 & 0.000000 & 0.000000 \\ 0.000000 & 1.000000 & 0.000000 \\ 0.000000 & 0.000000 & 1.000000 \end{matrix}$$

$$y = \begin{matrix} 2.718282 & 1.000000 & 1.000000 \\ 1.000000 & 2.718282 & 1.000000 \\ 1.000000 & 1.000000 & 2.718282 \end{matrix}$$

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

Purpose

To let the compiler know about symbols that are referenced above or in a separate file from their definitions.

Format

external proc *dog*,*cat*;

external keyword *dog*;

external fn *dog*;

external matrix *x*,*y*,*z*;

external string *mstr*,*cstr*;

Remarks

See Chapter 9.

You may have several procedures in different files that reference the same global variable. By placing an **external** statement at the top of each file, you can let the compiler know if the symbol is a matrix, string or procedure. If the symbol is listed and strongly typed in an active library, no **external** statement is needed.

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

Example

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

```
external matrix _eigerr;

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

Without the **external** statement, the compiler would assume that **_eigerr** was a procedure and incorrectly compile this program. The file containing the **eigrg** procedure also contains an **external** statement that defines **_eigerr** as a matrix, but this would not be encountered until the **if** statement containing the reference to **_eigerr** in the main program file had already been incorrectly compiled.

See also

declare

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);
```

$$x = \begin{matrix} 1.000000 & 0.000000 & 0.000000 \\ 0.000000 & 1.000000 & 0.000000 \\ 0.000000 & 0.000000 & 1.000000 \end{matrix}$$

See also

zeros, **ones**

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.

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**.

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.
_fcmptol	global scalar, comparison tolerance. The default value is $1.0e-15$.

Output

y	scalar, 1 (true) or 0 (false).

Remarks

The return value is true if **every** comparison is true.

The statement:

```
y = feq(a,b);
```

is equivalent to:

```
y = a eq b;
```

The calling program can reset **_fcmptol** before calling these procedures.

```
_fcmptol = 1e-12;
```

Example

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

$$x = \begin{matrix} 0.038289504 & 0.072535275 \\ 0.014713947 & 0.96863611 \end{matrix}$$

$$y = \begin{matrix} 0.25622293 & 0.70636474 \\ 0.0036191244 & 0.35913385 \end{matrix}$$

$$t = \begin{matrix} 0.0000000 \end{matrix}$$

Source

fcompare.src

Globals

_fcmptol

See also

dotfeq

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.

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);
```

$$y = \begin{matrix} 23.500000 & -1.0000000 \\ -0.5000000 & 0.00000000 \end{matrix}$$

See also

ffti, **rfft**, **rffti**

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);
```

$$y = \begin{matrix} 23.500000 & -1.0000000 \\ -0.5000000 & 0.00000000 \end{matrix}$$

```
fi = ffti(y);
```

$$fi = \begin{matrix} 22.000000 & 24.000000 \\ 23.000000 & 25.000000 \end{matrix}$$

See also

fft, **rfft**, **rffti**

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 2x2x2x2 hypercubic x would consist of two cubes in sequence, each cube containing two matrices in sequence, each matrix containing two rows in sequence, and each row containing two columns in sequence. Visually, x would look something like this:

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

Or, in an extended **GAUSS** notation, x would be:

```
Xhyper = x[1,.,.,.] | x[2,.,.,.];
Xcube1 = x[1,1,.,.] | x[1,2,.,.];
Xmat1  = x[1,1,1,.] | x[1,1,2,.];
Xrow1  = x[1,1,1,1] | x[1,1,1,2];
```

To be explicit, x would be laid out like this:

```
x[1,1,1,1]  x[1,1,1,2]  x[1,1,2,1]  x[1,1,2,2]
x[1,2,1,1]  x[1,2,1,2]  x[1,2,2,1]  x[1,2,2,2]
x[2,1,1,1]  x[2,1,1,2]  x[2,1,2,1]  x[2,1,2,2]
x[2,2,1,1]  x[2,2,1,2]  x[2,2,2,1]  x[2,2,2,2]
```

Constructing multi-dimensional matrices in vector format is actually quite simple. If you look at the last diagram for the layout of x you'll notice that each line actually constitutes the elements of an ordinary matrix in normal row-major order. This is easy to achieve with **vecr**. Further, each pair of lines or "matrices" constitutes one of the desired cubes, again with all the elements in the correct order. And finally, the two cubes combine to form the hypercube. So, the process of construction is simply a sequence of concatenations of column vectors, with a **vecr** step if necessary to get started.

Here's an example, this time working with a 2x3x2x3 hypercube.

```
let dim = 2 3 2 3;

let x1[2,3] = 1 2 3 4 5 6;
let x2[2,3] = 6 5 4 3 2 1;
let x3[2,3] = 1 2 3 5 7 11;
xc1 = vecr(x1)|vecr(x2)|vecr(x3);    /* cube 1 */

let x1 = 1 1 2 3 5 8;
let x2 = 1 2 6 24 120 720;
let x3 = 13 17 19 23 29 31;
xc2 = x1|x2|x3;                      /* cube 2 */

xh = xc1|xc2;                        /* hypercube */
xhfft = fftm(xh,dim);

let dimi = 2 4 2 4;
xhffti = fftmi(xhfft,dimi);
```

I 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 2x3x3 three-dimensional array, i.e., two 3x3 matrices of data. Suppose that *x1* is the first 3x3 matrix and *x2* the second 3x3 matrix, then $x =$ **vecr(***x1***) | vecr(***x2***)**.

The size of *dim* tells you how many dimensions x has.

The arrays have to be padded in each dimension to the nearest power of two. Thus the output array can be larger than the input array. In the 2x3x2x3 hypercube example, x would be padded from 2x3x2x3 out to 2x4x2x4. The input vector would contain 36 elements, while the output vector would contain 64 elements. You may have noticed that we used a *dimi* with padded values at the end of the example to check our answer.

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 2x2x2x2 hypercubic x would consist of two cubes in sequence, each cube containing two matrices in sequence, each matrix containing two rows in sequence, and each row containing two columns in sequence. Visually, x would look something like this:

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

Or, in an extended **GAUSS** notation, x would be:

```
Xhyper = x[1,.,.,.] | x[2,.,.,.];
Xcube1 = x[1,1,.,.] | x[1,2,.,.];
Xmat1  = x[1,1,1,.] | x[1,1,2,.];
Xrow1  = x[1,1,1,1] | x[1,1,1,2];
```

To be explicit, x would be laid out like this:

```
x[1,1,1,1]  x[1,1,1,2]  x[1,1,2,1]  x[1,1,2,2]
x[1,2,1,1]  x[1,2,1,2]  x[1,2,2,1]  x[1,2,2,2]
x[2,1,1,1]  x[2,1,1,2]  x[2,1,2,1]  x[2,1,2,2]
x[2,2,1,1]  x[2,2,1,2]  x[2,2,2,1]  x[2,2,2,2]
```

Constructing multi-dimensional matrices in vector format is actually quite simple. If you look at the last diagram for the layout of x you'll notice that each line actually constitutes the elements of an ordinary matrix in normal row-major order. This is easy to achieve with **vecr**. Further, each pair of lines or "matrices" constitutes one of the desired cubes, again with all the elements in the correct order. And finally, the two cubes combine to form the hypercube. So, the process of construction is simply a sequence of concatenations of column vectors, with a **vecr** step if necessary to get started.

Here's an example, this time working with a 2x3x2x3 hypercube.

```
let dim = 2 3 2 3;

let x1[2,3] = 1 2 3 4 5 6;
let x2[2,3] = 6 5 4 3 2 1;
let x3[2,3] = 1 2 3 5 7 11;
xc1 = vecr(x1)|vecr(x2)|vecr(x3);    /* cube 1 */

let x1 = 1 1 2 3 5 8;
let x2 = 1 2 6 24 120 720;
let x3 = 13 17 19 23 29 31;
xc2 = x1|x2|x3;                      /* cube 2 */

xh = xc1|xc2;                        /* hypercube */
xhffti = fftmi(xh,dim);
```

I 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 2x3x3 three-dimensional array, i.e., two 3x3 matrices of data. Suppose that *x1* is the first 3x3 matrix and *x2* the second 3x3 matrix, then $x =$ **vecr(***x1***) | vecr(***x2***)**.

The size of *dim* tells you how many dimensions x has.

The arrays have to be padded in each dimension to the nearest power of two. Thus the output array can be larger than the input array. In the 2x3x2x3 hypercube example, x would be padded from 2x3x2x3 out to 2x4x2x4. The input vector would contain 36 elements, while the output vector would contain 64 elements.

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

$$2^p \times 3^q \times 5^r \times 7^s, \qquad \begin{array}{l} p, q, r \text{ nonnegative integers} \\ s = 0 \text{ or } 1 \end{array}$$

If a dimension of x does not meet this requirement, it will be padded with zeros to the next allowable size before the FFT is computed.

fftn pads matrices to the next allowable dimensions; however, it generally runs faster for matrices whose dimensions are highly composite numbers, i.e., products of several factors (to various powers), rather than powers of a single factor. For example, even though it is bigger, a 33600x1 vector can compute as much as 20% faster than a 32768x1 vector, because 33600 is a highly composite number, $2^6 \times 3 \times 5^2 \times 7$, whereas 32768 is a simple power of 2, 2^{15}. For this reason, you may want to hand-pad matrices to optimum dimensions before passing them to **fftn**. The Run-Time Library includes a routine, **optn**, for determining optimum dimensions.

The Run-Time Library also includes the **nextn** routine, for determining allowable dimensions for a matrix. (You can use this to see the dimensions to which **fftn** would pad a matrix.)

fftn scales the computed FFT by 1/(L*M).

See also

fft, **ffti**, **fftm**, **fftmi**, **rfft**, **rffti**, **rfftip**, **rfftn**, **rfftnp**, **rfftp**

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.

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.

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.

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.

Purpose

Returns names and information for files that match a specification.

Format

{ *fnames*,*finfo* } = **fileinfo(***fspec***);**

Input

fspec string, file specification. Can include path. Wildcards are allowed in the filename.

Output

fnames N×1 string array of all filenames that match, null string if none are found.

finfo N×13 matrix, information about matching files.

Unix

[N, 1]	filesystem ID
[N, 2]	inode number
[N, 3]	mode bit mask
[N, 4]	number of links
[N, 5]	user ID
[N, 6]	group ID
[N, 7]	device ID (char/block special files only)
[N, 8]	size in bytes
[N, 9]	last access time
[N,10]	last data modification time
[N,11]	last file status change time
[N,12]	preferred I/O block size
[N,13]	number of 512-byte blocks allocated

OS/2, Windows

[N, 1]	drive number (A = 0, B = 1, etc.)
[N, 2]	n/a, 0
[N, 3]	mode bit mask
[N, 4]	number of links, always 1
[N, 5]	n/a, 0
[N, 6]	n/a, 0
[N, 7]	n/a, 0
[N, 8]	size in bytes
[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

DOS

[N, 1]	drive number (A = 0, B = 1, etc.)
[N, 2]	n/a, 0
[N, 3]	mode bit mask
[N, 4]	number of links, always 1
[N, 5]	n/a, 0
[N, 6]	n/a, 0
[N, 7]	n/a, 0
[N, 8]	size in bytes
[N, 9]	n/a, 0
[N,10]	last data modification time
[N,11]	n/a, 0
[N,12]	n/a, 0
[N,13]	n/a, 0

finfo will be a scalar zero if no matches are found.

Remarks

fnames will contain file *names* only; any path information that was passed is dropped.

The time stamp fields ($finfo[N, 9]$–$[N, 11]$) are expressed as the number of seconds since midnight, Jan. 1, 1970, Coordinated Universal Time (UTC).

See also

files, **filesa**

Purpose

Returns a matrix of matching file names.

Format

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

Input

n	string containing any combination of drive, path, and filename to search for. Wildcards are allowed in the filename.
a	scalar, the attribute to use in the search.

Output

y	N×2 matrix of all filenames that match, or scalar 0 if none are found.

Portability

Unix, **OS/2**, **Windows**

files was written to work under DOS, with its 8x3 filenames; use **fileinfo** or **filesa** instead.

Remarks

The file names will be in the first column of the returned matrix. The drive and path that was passed is dropped, and the extensions, including the period '.', will be in the second column. For files with no extension the second column entry will be null. Volume labels look like filenames and have a period before the 9^{th} character.

The attribute corresponds to the file attribute for each entry in the disk directory. For normal files an attribute of 0 is used. This will return the names of all matching "normal" files. The bits in the attribute byte have the following meaning:

2	Hidden files
4	System files
8	Volume Label (DOS only)
16	Subdirectory

The values above are added together to specify different types of files to be searched for. Therefore 6 would specify hidden+system.

If the attribute is set for hidden, system, or subdirectory entries then it will be an inclusive search. All normal entries plus all entries matching the specified attributes will be returned.

If the attribute is set for the volume label, it will be an exclusive search and only the volume label will be returned.

Example

```
y = files("ch*.*",0);
```

In this example all normal files listed in the current directory that begin with "ch" will be returned.

```
y = files("ch*.",0);
```

In this example all normal files listed in the current directory that begin with "ch" but do not have an extension will be returned.

```
y = files("ch*.*",2+4+16);
```

In this example all normal, hidden, and system files and all subdirectories listed in the current directory that begin with "ch" will be returned.

```
proc exist(filename);
    retp(not files(filename,0) $== 0);
endp;
```

This procedure will return 1 if the file exists or 0 if not.

See also

dos, **fileinfo**, **filesa**

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 the filename.

Output

y	N×1 string array of all filenames 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

dos, **fileinfo**, **files**

Purpose

Round down toward $-\infty$.

Format

y = **floor(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix containing the elements of x rounded down.

Remarks

This rounds every element in the matrix x "down" to the nearest integer.

Example

```
x = 100*rndn(2,2);
```

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

```
f = floor(x);
```

$$f = \begin{matrix} 77.00 & -15.00 \\ 4.00 & -159.00 \end{matrix}$$

See also

ceil, **round**, **trunc**

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, in Chapter 8.

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

Purpose

Allows user to create one-line functions.

Format

fn *fn_name*(*args*) = *code_for_function*;

Example

```
fn area(r) = pi*r*r;

a = area(4);
```

Remarks

Functions can be called in the same way as other procedures.

Purpose

Opens a file.

Format

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

Input

filename	string, name of file to open.
omode	string, file I/O mode.

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**.

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. It is created with an implicit **#define**. 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.

When the loop terminates, the value of the counter is *stop* if the loop terminated naturally. 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 below.

Example

```
x = zeros(10, 5);
for i (1, rows(x), 1);
    for j (1, cols(x), 1);
        x[i,j] = i*j;
    endfor;
```

```
endfor;


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;


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;
```

Purpose

Controls the format of matrices and numbers printed out with **print** or **lprint** 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 (**/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 1x1 string array.

/*mf* literal, matrix row format flag.

/m0 no delimiters before or after rows when printing out matrices.

/m1 or /mb1 print 1 carriage return/line feed pair **before** each row of a matrix **with more than 1 row**.

/m2 or /mb2 print 2 carriage return/line feed pairs **before** each row of a matrix **with more than 1 row**.

/m3 or /mb3 print "Row 1", "Row 2"... **before** each row of a matrix **with more than one row**.

/ma1 print 1 carriage return/line feed pair **after** each row of a matrix **with more than 1 row**.

/ma2 print 2 carriage return/line feed pairs **after** each row of a matrix **with more than 1 row**.

/a1 print 1 carriage return/line feed pair **after** each row of a matrix.

/a2 print 2 carriage return/line feed pairs **after** each row of a matrix.

/b1 print 1 carriage return/line feed pair **before** each row of a matrix.

/b2 print 2 carriage return/line feed pairs **before** each row of a matrix.

/b3 print "Row 1", "Row 2"... **before** each row of a matrix.

/jnt literal, matrix element format flag – controls justification, notation and trailing character.

Right-Justified

/rd Signed decimal number in the form ⟦-⟧`####.####`, where `####` is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed.

/re Signed number in the form ⟦-⟧`#.##E±###`, where `#` is one decimal digit, `##` is one or more decimal digits depending on the precision, and `###` is three decimal digits. If precision is 0, the form will be ⟦-⟧`#E±###` with no decimal point printed.

/ro This will give a format like **/rd** or **/re** depending on which is most compact for the number being printed. A format like **/re** will be used only if the exponent value is less than -4 or greater than the precision. If a **/re** format is used, a decimal point will always appear. The precision signifies the number of significant digits displayed.

/rz This will give a format like **/rd** or **/re** depending on which is most compact for the number being printed. A format like **/re** will be used only if the exponent value is less than -4 or greater than the precision. If a **/re** format is used, trailing zeros will be supressed and a decimal point will appear only if one or more digits follow it. The precision signifies the number of significant digits displayed.

Left-Justified

/ld Signed decimal number in the form ⟦-⟧`####.####`, where `####` is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed. If the number is positive, a space character will replace the leading minus sign.

/le Signed number in the form ⟦-⟧`#.##E±###`, where `#` is one decimal digit, `##` is one or more decimal digits depending on the precision, and `###` is three decimal digits. If precision is 0, the form will be ⟦-⟧`#E±###` with no decimal point printed. If the number is positive, a space character will replace the leading minus sign.

/lo This will give a format like **/ld** or **/le** depending on which is most compact for the number being printed. A format like **/le** will be

used only if the exponent value is less than -4 or greater than the precision. If a **/le** format is used, a decimal point will always appear. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.

/lz This will give a format like **/ld** or **/le** depending on which is most compact for the number being printed. A format like **/le** will be used only if the exponent value is less than -4 or greater than the precision. If a **/le** format is used, trailing zeros will be supressed and a decimal point will appear only if one or more digits follow it. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.

Trailing Character

The following characters can be added to the */jnt* parameters above to control the trailing character if any:

```
format /rdn 1,3;
```

s The number will be followed immediately by a space character. This is the default.

c The number will be followed immediately with a comma.

t The number will be followed immediately with a tab character.

n No trailing character.

f scalar expression, controls the field width.

p scalar expression, controls the precision.

Portability

Unix, **OS/2**, **Windows**

format sets the **print** statement format for the active window. Each window "remembers" its own format, even when it is no longer the active window.

Remarks

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 have to 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

```
x = rndn(3,3);

format /m1 /rd 16,8;
print x;

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

format /m1 /rzs 1,10;
print x;

-1.635334648 1.613507002 -1.062951787
0.2617128159 0.2797229414 -1.389372421
0.5889111366 0.4681220206 1.088059602

format /m3 /rdn 16,4;
print x;

Row 1
         -1.6353          1.6135         -1.0630
Row 2
```

```
          0.2617          0.2797         -1.3894
Row 3
          0.5889          0.4681          1.0881

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

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

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

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

See also

formatcv, **formatnv**, **print**, **lprint**, **output**

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

```
x = { A 1, B 2, C 3 };
oldfmt = formatcv("*.*s" ~ 3 ~ 3);
call printfmt(x,0~1);
call formatcv(oldfmt);
```

Source

gauss.src

Globals

__fmtcv

See also

formatnv, **printfm**, **printfmt**

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

```
x = { A 1, B 2, C 3 };
oldfmt = formatnv("*.*lf" ~ 8 ~ 4);
call printfmt(x,0~1);
call formatnv(oldfmt);
```

Source

gauss.src

Globals

__fmtnv

See also

formatcv, **printfm**, **printfmt**

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.

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.

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 is 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.

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.

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.

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

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**

Purpose

Returns the value of the gamma function.

Format

y = **gamma(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix.

Remarks

For each element of x this function returns the integral

$$\int_0^\infty t^{(x-1)}e^{-t}dt$$

All elements of x must be positive and less than or equal to 169. Values of x greater than 169 will cause an overflow.

The natural log of **gamma** is often what is required and it can be computed without the overflow problems of **gamma**. **lnfact** can be used to compute log gamma.

Example

```
y = gamma(2.5);
```

$y = 1.32934$

See also

cdfchic, **cdfbeta**, **cdffc**, **cdfn**, **cdfnc**, **cdftc**, **erf**, **erfc**

Purpose

Compute 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**

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**

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, up to 65519 bytes long.

Remarks

If the file is loaded in ASCII mode, it will be tested to see if it contains any end of file characters. These are ^Z (ASCII 26). The file will be truncated before the first ^Z and there will be no ^Z's in the string. This is the correct way to load most text files because the ^Z's can cause problems when trying to print the string to a printer.

If the file is loaded in binary mode, it will be loaded just like it is with no changes.

Example

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

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

Then execute the following.

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

print y;

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

See also

load, **save**, **let**, **con**

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;

    TIME
    DIST
    TEMP
   FRICT
```

The above example assumes the data set **olsdat** contained 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**

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;

    time
    dist
    temp
   frict
```

The above example assumes the data set **olsdat** contained the variables: **time, dist, temp, frict**. Note the use of **vartypef** to determine the types of these variables.

See also

getname, **indcv**, **vartypef**

Purpose

Compute 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**

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;

       /gauss/temp.e
```

Source

getpath.src

Globals

None

Purpose

Causes a branch to a subroutine.

Format

gosub *label*;

⋮

label:

⋮

return;

Remarks

See Chapter 9 for multi-line recursive user-defined functions.

When a **gosub** statement is encountered the program will branch to the label and begin executing from there. When a **return** statement is encountered, the program will resume executing at the statement following the **gosub** statement. Labels are 1-32 characters long and are followed by a colon. The characters can be A-Z or 0-9 and they must begin with an alphabetic character. Uppercase or lowercase is allowed.

It is possible to pass parameters to subroutines and receive parameters from them when they return. See the second example below.

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 Chapter 9.

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:

1. **x** and **y** are pushed on the stack and the program branches to the label **mysub** in line 10.

11. the second argument that was pushed, **y**, is **pop**'ped into **b**.

12. the first argument that was pushed, **x**, is **pop**'ped into **a**.

14. **inv(b)*b+a** is assigned to **a**.

15. **a'b** is assigned to **b**.

16. **a** and **b** are pushed on the stack and the program branches to the statement following the **gosub**, which is line 2.

2. the second argument that was pushed, **b**, is **pop**'ped into **j**.

3. the first argument that was pushed, **a**, is **pop**'ped into **k**.

4. **j*k** is assigned to **t**.

5. **t** is printed.

6. the program is terminated with the **end** statement.

Matrices are pushed on a last-in/first-out stack in the **gosub()** and **return()** statements. They must be popped off in the reverse order. No intervening statements are allowed between the label and the **pop** or the **gosub** and the **pop**. Only one matrix may be popped per **pop** statement.

See also

goto, **proc**, **pop**, **return**

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;
```

```
      1.0000000   2.0000000   3.0000000
```

See also

gosub, **if**

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);
```

$$y = \begin{matrix} 82.98901842 & 0.00000000 & 0.00000000 \\ 0.00000000 & 281.19752975 & 0.00000000 \\ 0.00000000 & 0.00000000 & 1087.95414117 \end{matrix}$$

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`

Globals

None

See also

hessp

Purpose

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

Format

graph x,y;

Input

x	N×K matrix of horizontal coordinates.
y	L×M matrix, vertical coordinates, E×E conformable with x.

Remarks

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

The origin 0,0 is at the lower lefthand corner of the screen.

Note that the screen must be in graphics mode for this command to operate. See **setvmode**.

Example

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

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

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

See also

setvmode, **color**, **plot**

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

Purpose

Print 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 the letters:

't'	title is to be printed
'l'	lines are to bracket the title
'd'	a date and time is to be printed
'v'	version number of program is printed
'f'	file name being analyzed is printed

__title string, title for header.

Source

`gauss.src`

Globals

__header, **__title**

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 Charles F. Van Loan, **Using the Hessenberg Decomposition in Control Theory**, in *Algorithms and Theory in Filtering and Control*, D.C. Sorenson and R.J. Wets (eds), Mathematical Programming Study No. 18, North Holland, Amsterdam, pp. 102-111, 1982).

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);
```

$$h = \begin{matrix} 1.00000000 & -3.59700730 & -0.24806947 \\ -8.06225775 & 14.04615385 & 2.83076923 \\ 0.00000000 & 0.83076923 & -0.04615385 \end{matrix}$$

$$z = \begin{matrix} 1.00000000 & 0.00000000 & 0.00000000 \\ 0.00000000 & -0.49613894 & -0.86824314 \\ 0.00000000 & -0.86824314 & 0.49613894 \end{matrix}$$

See also

schur

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`

Globals

None

See also

gradp

Purpose

Returns the number of hundredths of a second since midnight.

Format

y = **hsec;**

Remarks

The number of hundredths of a second since midnight can also be accessed as the [4,1] element of the vector returned by the **date** function.

Example

```
x = rndu(100,100);
ts = hsec;
y = x*x;
et = hsec-ts;
```

In this example, **hsec** is used to time a 100×100 multiplication in **GAUSS**. A 100×100 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**

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

Purpose

Returns the imaginary part of a matrix.

Format

zi = **imag(***x***);**

Input

x	N×K matrix.

Output

zi	N×K matrix, the imaginary part of *x*.

Remarks

If *x* is real, *zi* will be an N×K matrix 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**

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 complete drive and pathname is specified for the file, then no additional searching will be attempted if the file is not found.

If a complete drive and pathname is not specified and a **src_path** configuration variable is defined in the **GAUSS** configuration file, then each path listed in **src_path** will be searched for the specified filename. Any drive or path on the user-specified filename will be stripped off prior to being combined with the paths found in **src_path**.

`#include /gauss/myprog.prc;` No additional search will be made if the file is not found.

`#include myprog.prc;` The paths listed in **src_path** will be searched for `myprog.prc` if the file is not found.

The **src_path** configuration variable can be defined in the **GAUSS** configuration file by editing the file with any text editor.

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**

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);
```

$$what = \begin{matrix} \text{AGE} \\ \text{PAY} \\ \text{SEX} \end{matrix}$$

$$where = \begin{matrix} \text{AGE} \\ \text{SEX} \\ \text{JOB} \\ \text{date} \\ \text{PAY} \end{matrix}$$

$$z = \begin{matrix} 1 \\ 5 \\ 2 \end{matrix}$$

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);
```

$$x = \begin{matrix} 1.0 \\ 4.0 \\ 3.3 \\ 4.2 \\ 6.0 \\ 5.7 \\ 8.1 \\ 5.5 \end{matrix}$$

$$v = \begin{matrix} 4 \\ 6 \end{matrix}$$

$$y = \begin{matrix} 4 \\ 5 \\ 6 \\ 8 \end{matrix}$$

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

Globals

None

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`

Globals

None

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);
```

$$what = \begin{matrix} 8 \\ 7 \\ 3 \end{matrix}$$

$$where = \begin{matrix} 2 \\ 7 \\ 8 \\ 4 \\ 3 \end{matrix}$$

$$z = \begin{matrix} 3 \\ 2 \\ 5 \end{matrix}$$

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.
_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;
gl = &g1|&g2;
_intord = 40;
_intrec = 0;
y = intgrat2(&f,xl,gl);
```

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

intgrat2, **_intord**, **_intq12**, **_intq16**, **_intq2**, **_intq20**, **_intq24**, **_intq3**, **_intq32**, **_intq4**, **_intq40**, **_intq6**, **_intq8**, **_intrec**

See also

intgrat3, **intquad1**, **intquad2**, **intquad3**, **intsimp**

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.

_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 ***** or **/**.

Example

```
proc f(x,y,z);
   retp(2);
endp;

proc g1(x);
    retp(sqrt(25-x^2));
endp;

proc g2(x);
    retp(-g1(x));
endp;

proc h1(x,y);
     retp(sqrt(25 - x^2 - y^2));
endp;

proc h2(x,y);
     retp(-h1(x,y));
endp;

xl = 5|-5;
gl = &g1|&g2;
hl = &h1|&h2;
_intrec = 0;
_intord = 40;
y = intgrat3(&f,xl,gl,hl);
```

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

intgrat3, **_intord**, **_intq12**, **_intq16**, **_intq2**, **_intq20**, **_intq24**, **_intq3**, **_intq32**, **_intq4**, **_intq40**, **_intq6**, **_intq8**, **_intrec**

See also

intgrat2, **intquad1**, **intquad2**, **intquad3**, **intsimp**

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.
_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**

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.
_intord	global scalar, the order of the integration. The larger **_intord**, the more precise the final result will be. **_intord** may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.
_intrec	global scalar. This variable is used to keep track of the level of recursion of **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

intquad2, **_intord**, **_intq12**, **_intq16**, **_intq2**, **_intq20**, **_intq24**, **_intq3**, **_intq32**, **_intq4**, **_intq40**, **_intq6**, **_intq8**, **_intrec**

See also

intquad1, **intquad3**, **intsimp**, **intgrat2**, **intgrat3**

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.
_intord	global scalar, the order of the integration. The larger **_intord**, the more precise the final result will be. **_intord** may be set to 2, 3, 4, 6, 8, 12, 16, 20, 24, 32, 40. Default = 12.
_intrec	global scalar. This variable is used to keep track of the level of recursion of **intquad3** and may start out with a different value if your program terminated inside of the integration function on a previous run. Always set **_intrec** explicitly to 0 before any calls to **intquad3**.

Output

y	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

intquad3, **_intord**, **_intq12**, **_intq16**, **_intq2**, **_intq20**, **_intq24**, **_intq3**, **_intq32**, **_intq4**, **_intq40**, **_intq6**, **_intq8**, **_intrec**

See also

intquad1, **intquad2**, **intsimp**, **intgrat2**, **intgrat3**

Purpose

Interleaves the rows of two files that have been sorted on a common variable, to give a single file sorted on that variable.

Format

intrleav(*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, i.e., the same number of columns AND the same variable names. They must both already be sorted on the key column. This procedure will combine them into one large file, sorted by the key variable.

If the inputs are null or 0, the procedure will ask for them.

Example

```
intrleav("freq.dat","freqdata.dat","intfile","AGE",1);
```

Source

`sortd.src`

Globals

None

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

Globals

None

Example

```
let v1 = mary jane linda dawn;
let v2 = mary sally lisa ruth linda;
y = intrsect(v1,v2,0);
```

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]*.

Remarks

Example

```
proc f(x);
   retp(sin(x));
endp;

let xl = { 1,
           0 };

y = intsimp(&f,xl,1E-8);
```

This will integrate the function between 0 and 1.

Source

intsimp.src

Globals

None

See also

intquad1, **intquad2**, **intquad3**, **intgrat2**, **intgrat3**

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.

Output

y	N×N matrix containing the inverse of x.

Remarks

x can be any legitimate matrix expression that returns a matrix that is legal for the function.

For **inv**, x must be square and invertible.

For **invpd**, x must be symmetric and positive definite.

If the input matrix is not invertible by these functions, they will either terminate the program with an error message or return an error code which can be tested for with the **scalerr** function. This depends on the **trap** state as follows:

trap 1, return error code

inv	**invpd**
50	20

trap 0, terminate with error message

inv	**invpd**
"Matrix singular"	"Matrix not positive definite"

If the input to **invpd** is not symmetric, it is possible that the function will (erroneously) appear to operate successfully.

Positive definite matrices can be inverted by **inv**. However, for symmetric, positive definite matrices (such as moment matrices), **invpd** is about twice as fast as **inv**.

Example

```
n = 4000;
x1 = rndn(n,1);
x = ones(n,1)~x1;
btrue = { 1, 0.5 };
y = x*btrue + rndn(n,1);
bols = invpd(x'x)*x'y;
```

bols = 1.017201 0.484244

This example simulates some data and computes the **ols** coefficient estimator using the **invpd** function. First, the number of observations is specified. Second, a vector **x1** of standard Normal random variables is generated and is concatenated with a vector of 1's (to create a constant term). The true coefficients are specified, and the dependent variable **y** is created. Then the **ols** coefficient estimates are computed.

Technical Notes

For complex matrices, **inv** uses the ZGECO, ZGEDI path in the LINPACK routines. For real matrices, it uses the **croutp** function.

The **inv** function uses the Crout decomposition. The advantage of this routine is that on some platforms it allows most of the intermediate results to be computed in extended precision.

The **invpd** function uses the Cholesky decomposition and is based upon the LINPACK routines for positive definite matrices. On OS/2 and DOS, if **prcsn** 80 is in effect, all intermediate calculations and intermediate results will be in the 80-bit extended precision of the 80x87 temporary real format. The final results will be rounded to 64-bit double precision.

The tolerance used to determine singularity is 1.0e−14. This can be changed. See Appendix E.

See also

scalerr, **trap**, **prcsn**

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 four Moore-Penrose conditions:

1. $xyx = x$
2. $yxy = y$
3. xy is symmetric
4. yx is symmetric

The tolerance used to determine if a pivot element is zero is taken from the **crout** singularity tolerance. The corresponding row and column are zeroed out. See Appendix E.

Example

```
let x[3,3] = 1 2 3 4 5 6 7 8 9;
y = invswp(x);
```

$$y = \begin{matrix} -1.6666667 & 0.66666667 & 0.0000000 \\ 1.3333333 & -0.33333333 & 0.0000000 \\ 0.0000000 & 0.0000000 & 0.0000000 \end{matrix}$$

Purpose

Returns whether a matrix is complex or real.

Format

y = **iscplx(**x**);**

Input

x	N×K matrix.

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**

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**

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 or 0.

Remarks

y will be a scalar 1 if the matrix x contains any missing values, otherwise it will be a 0.

An element of x is considered to be a missing if and only if it contains a missing value in the real part. Thus, for $x = 1 + .\text{i}$, **ismiss**(x) will return a 0.

Example

```
x = { 1 6 3 4 };
y = ismiss(x);
```

$y =$ 0.0000000

See also

scalmiss, **miss**, **missrv**

Purpose

Tests whether a matrix is a sparse matrix.

Format

r = **isSparse(**x**);**

Input

x	M×N sparse or dense matrix.

Output

r	scalar, 1 if x is sparse, 0 otherwise.

Source

`sparse.src`

Purpose

Returns the ASCII value of the next key available in the keyboard buffer.

Format

y = **key;**

Portability

Unix, **OS/2**, **Windows**

key gets input from the active window. If you are working in terminal mode, **key** doesn't "see" any keystrokes until **Enter** is pressed.

Remarks

The value returned will be zero if no key is available in the buffer or it will equal the ASCII value of the key if one is available. The key is taken from the buffer at this time and the next call to **key** will return the next key.

Here are the values returned on a PC 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	Pg Up
1075	Cursor Left
1077	Cursor Right
1079	End
1080	Cursor Down
1081	Pg Dn
1082	Ins
1083	Del
1084-1093	Shift F1-F10
1094-1103	Ctrl F1-F10
1104-1113	Alt F1-F10
1114	Ctrl-PrtSc
1115	Ctrl-Cursor Left
1116	Ctrl-Cursor Right
1117	Ctrl-End
1118	Ctrl-PgDn
1119	Ctrl-Home
1120-1131	Alt 1,2,3,4,5,6,7,8,9,0,-,=
1132	Ctrl-PgUp

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**

Purpose

Waits for and gets a key.

Format

k = **keyw;**

Output

k	scalar, ASCII value of the key pressed.

Portability

Unix, OS/2, Windows

keyw gets the next key from the input buffer of the active window.

If you are working in terminal mode, **GAUSS** will not see any input until you press the **Enter** key.

Remarks

keyw gets the next key from the keyboard buffer. If the keyboard buffer is empty, **keyw** waits for a keystroke. For normal keys, **keyw** returns the ASCII value of the key. See **key** for a table of return values for extended and function keys.

See also

key

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 Chapter 9.

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**

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`

Globals

None

See also

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

Globals

None

See also

lag1

Purpose

Creates a matrix from a list of numeric or character values. The result is always of type MATRIX.

Format

let *x* **=** *constant_list*;

Example

```
let x;
```

$$x = 0$$

```
let x = { 1 2 3, 4 5 6, 7 8 9 };
```

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

```
let x[3,3] = 1 2 3 4 5 6 7 8 9;
```

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

```
let x[3,3] = 1;
```

$$x = \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

```
let x[3,3];
```

$$x = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$

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

$$x = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{matrix}$$

```
let x = dog cat;
```

$$x = \begin{matrix} \text{DOG} \\ \text{CAT} \end{matrix}$$

```
let x = "dog" "cat";
```

$$x = \begin{matrix} \text{dog} \\ \text{cat} \end{matrix}$$

Remarks

Expressions and matrix names are not allowed in the **let** command, expressions such as this:

```
let x[2,1] = 3*a b
```

are illegal. To define matrices by combining matrices and expressions, use an expression containing the concatenation operators: $\sim$ and `|` .

Numbers can be entered in scientific notation. The syntax is $d\text{E}\pm n$, where d is a number and n is an integer (denoting the power of 10).

```
let x = 1e+10 1.1e-4 4.019e+2;
```

Complex numbers can be entered by joining the real and imaginary parts with a sign (+ or -); there can be no spaces between the numbers and the sign. Numbers with no real part can be entered by appending an "i" to the number.

```
let x = 1.2+23 8.56i 3-2.1i -4.2e+6i 1.2e-4-4.5e+3i;
```

If curly braces are used, the **let** is optional. You will need the **let** for statements that you want to protect from the beautifier using the **-l** flag on the beautifier command line.

```
let x = { 1 2 3, 4 5 6, 7 8 9 };

x = { 1 2 3, 4 5 6, 7 8 9 };
```

If indices are given, a matrix of that size will be created:

```
let x[2,2] = 1 2 3 4;
```

$$x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

If indices are not given, a column vector will be created:

```
let x = 1 2 3 4;
```

$$x = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

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 Section 7.6.2 for more information on empty matrices.

Character elements are allowed in a **let** statement:

```
let x = age pay sex;
```

$$x = \begin{matrix} \text{AGE} \\ \text{PAY} \\ \text{SEX} \end{matrix}$$

Lowercase elements can be created if quotation marks are used. Note each element must be quoted.

```
let x = "age" "pay" "sex";
```

$$x = \begin{matrix} \text{age} \\ \text{pay} \\ \text{sex} \end{matrix}$$

See also

con, **cons**, **declare**, **load**

Purpose

To build and update 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 ramdisk. 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

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 listing will reside in a file called `gausslib.lst` located on the directory defined by the **lib_path** configuration variable. Any existing file by that name will be overwritten.

See Chapter 10 for more information about the library system.

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. The default libraries and autoload/autodelete settings can be set at the **Compile Options** menu (press **Alt-C**,**C**).

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

If **dog** calls **cat** and **cat** calls **bird** and they are all in separate files, they will all be found by the autoloader. They must be procedures. If these are **fn** functions, you will need to declare them in an **external** statement in your calling program.

The help facility activated by **Alt-H,H** will search for **dog** in exactly the same sequence as the autoloader. The file containing **dog** will be displayed on the screen and you can scroll up and down and look at the code and comments. If you request "`svd.src`" or "`lib/gauss.lcg`", you have direct access to examine the file requested. If no path is given, it will be searched for in the default directory and then along the **src_path**. Library files have a `.lcg` extension and will be searched for in the **lib_path** directory. Only one library path is supported.

Library files are simple ASCII files that you can create with the editor. Here is an example:

```
/*
** This is a GAUSS library file.
*/

eig.src
    eig      : proc
    eigsym   : proc
    _eigerr  : matrix
svd.src
    cond     : proc
    pinv     : proc
    rank     : proc
    svd      : proc
    _svdtol  : matrix
```

The lines not indented are the file names. The lines that are indented are the symbols defined in that file. As you can see, a **GAUSS** library is a dictionary of files and the global symbols they contain.

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
```

Any line beginning with "/*", "**" or "*/" is considered a comment. Blank lines are okay.

Here's a debugging hint. If your program is acting strange and you suspect it is autoloading the wrong copy of a procedure from some dark corner of your hard disk, use the help facility to locate the suspected function. It will use the same search path that the autoloader uses.

See also

declare, **external**, **lib**, **proc**

Purpose

To draw lines on the graphics screen.

Format

line $x[\![, y]\!]$**;**

Portability

This function is supported under DOS only.

Input

x	N×K matrix.
y	L×M matrix, E×E conformable with x.

Remarks

x and y must each have at least 2 elements or no lines will be drawn. The first line starts at x[1,1], y[1,1]. $P-1$ lines will be drawn where $P = max(N, L) * max(K, M)$. In other words the elements of x and y will be combined in the same way as with other element-by-element operators.

If there is only one argument,

```
line x;
```

it must be an N×4 or N×5 matrix. N lines will be drawn. Each row describes a line. Each column describes:

[N,1]	x coordinate of beginning of line.
[N,2]	y coordinate of beginning of line.
[N,3]	x coordinate of end of line.
[N,4]	y coordinate of end of line.
[N,5]	optional color of line.

The screen must be in graphics mode. All coordinates are in pixels with (0,0) in the lower left.

See also

setvmode, **color**, **graph**

Purpose

The **#lineson** command causes **GAUSS** to imbed 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 imbedding 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 a bit. In programs with a lot of 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 on the screen and run from the command line), since there are no line numbers in such programs anyway.

Line number tracking can be turned on and off at the **Run Options** menu (press **Alt-C**,**R**), but the **#lineson** and **#linesoff** commands will override that.

The line numbers and file names given at runtime 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, you may need to use the **trace** command to locate exactly where the error occurs.

The "Currently active call" message will always be correct. So if it says it was executing procedure **xyz** at line number **nnn** in file **ABC** and **xyz** has no line **nnn** or is not even 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 that the error was in within the limits stated above.

See also

trace

Purpose

Computes the natural log of all elements of a matrix.

Format

y = **ln**(x);

Input

x	N×K matrix.

Output

y	N×K matrix containing the natural log values of the elements of x.

Remarks

ln is defined for $x \neq 0$.

If x is negative, complex results are returned.

You can turn the generation of complex numbers for negative inputs on or off in the **GAUSS** configuration file, and with the **sysstate** function, case 8. If you turn it off, **ln** will generate an error for negative inputs.

If x is already complex, the complex number state doesn't matter; **ln** will compute a complex result.

x can be any expression that returns a matrix.

Example

```
y = ln(16);
```

$y =$ 2.7725887

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, $lnPr(X < x1, X < x2 \mid r)$.

Remarks

x1, *x2*, and *r* must be E×E conformable.

See also

cdfbvn, lncdfmvn

Source

`lncdfn.src`

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.

See also

cdfmvn, lncdfbvn

Source

`lncdfn.src`

Purpose

Computes natural log of Normal cumulative distribution function.

Format

y = **lncdfn(**x**);**

Input

x	N×K matrix, abscissae.

Output

y	N×K matrix, $ln\ Pr(X < x)$.

Source

`lncdfn.src`

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, ExE 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:

$\|x\| \leq 1$ and $dx \leq 1$	$\pm 1e-14$
$1 < \|x\| < 37$ and $\|dx\| < 1/\|x\|$	$\pm 1e-13$
$\min(x, x+dx) >$ -37 and $y > -690$	$\pm 1e-11$ or better

A relative error of $\pm 1e-14$ implies that the answer is accurate to better than ± 1 in the 14th digit.

Example

```
print lncdfn2(-10,29);

 -7.6198530241605269e-24

print lncdfn2(0,1);

 -1.0748623268620716e+00

print lncdfn2(5,1);

  -1.5068446096529453e+01
```

See also

cdfn2

Source

lncdfn.src

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`

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, all elements must be positive.

Output

y	N×K matrix containing the natural log of the factorial of each of the elements of x.

Remarks

For integer x, this is (approximately) $ln(x!)$. However, the computation is done using a formula, and the function is defined for noninteger x.

In most formulae in which the factorial operator appears, it is possible to avoid computing the factorial directly, and to use **lnfact** instead. The advantage of this is that **lnfact** does not have the overflow problems that the factorial (**!**) operator has.

For $x \geq 1$, this function has at least 6 digit accuracy, for $x > 4$ it has at least 9 digit accuracy, and for $x > 10$ it has at least 12 digit accuracy. For $0 < x < 1$, accuracy is not known completely but is probably at least 6 digits.

Sometimes log gamma is required instead of log factorial. These functions are related by:

lngamma(x**) = lnfact(**$x-1$**);**

Example

```
let x = 100 500 1000;
y = lnfact(x);
```

$$y = \begin{matrix} 363.739375560 \\ 2611.33045846 \\ 5912.12817849 \end{matrix}$$

Source

`lnfact.src`

Globals

None

Technical Notes

For $x > 1$, Stirling's formula is used.

For $0 < x \leq 1$, **ln(gamma(**x**+1))** is used.

See also

gamma, **!**

Purpose

Computes standard Normal log-probabilities.

Format

z = **lnpdfn(**x**);**

Input

x	N×K matrix, data.

Output

z	N×K matrix, log-probabilities.

Remarks

This computes the log of the scalar Normal density function for each element of x. z could be computed by the following **GAUSS** code:

y = **-ln(sqrt(2*pi))-**x**.***x**/2;**

For multivariate log-probabilities, see **lnpdfmvn**.

Example

```
proc lnpdfn(x);
    retp( -.918938533204672741 - (x.*x) / 2 );
endp;
```

Purpose

Computes multivariate Normal log-probabilities.

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.

Purpose

Load 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 with.

load, loadm	matrix
loads	string
loadf	function (**fn**)
loadk	keyword (**keyword**)
loadp	procedure (**proc**)

If no filename is given as with *x* above, then the symbol name the file is to be loaded into is used as the filename and the proper extension is added.

If more than one item is to be loaded in a single statement, the names should be separated by commas.

The filename can be either a literal or a string. If the filename is in a string variable, then the ^ (caret) operator must precede the name of the string, as in:

```
filestr = "mydata/char";
loadm x = ^filestr;
```

If no extension is supplied, the proper extension for each type of file will be used automatically as follows:

load	`.fmt` - matrix file or delimited ASCII file
loadm	`.fmt` - matrix file or delimited ASCII file
loads	`.fst` - string file
loadf	`.fcg` - user-defined function (**fn**) file
loadk	`.fcg` - user-defined keyword (**keyword**) file
loadp	`.fcg` - user-defined procedure (**proc**) file

These commands also signal to the compiler what type of object the symbol is so that later references to it will be compiled correctly.

A dummy definition must exist in the program for each symbol that is loaded in using **loadf**, **loadk**, or **loadp**. This resolves the need to 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 reshaped to the desired form. You could, for instance, put the number of rows and columns of the matrix right in the file as the first and second elements and reshape the remainder of the vector to the desired form using those values.

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 your file contains nine numbers (1 2 3 4 5 6 7 8 9), then the matrix x that was created would be as follows:

```
load x[1,9] = data.asc;
```

$$x = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{matrix}$$

```
load x[3,3] = data.asc;
```

$$x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

```
load x[2,2] = data.asc;
```

$$x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$$

```
load x[2,9] = data.asc;
```

$$x = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{matrix}$$

```
load x[3,5] = data.asc;
```

$$x = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 & 6 \end{matrix}$$

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, **save**, **let**, **con**, **cons**, **sysstate**

Purpose

Load a small 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

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 example above will be treated as a procedure whenever it is accessed in the procedure. What is actually passed in is a pointer to a procedure.

See Chapter 9.

See also

proc

Purpose

This statement positions the cursor on the screen.

Format

locate m**,** n**;**

Portability

Unix

locate locates the cursor in the active window. The row argument is ignored for window 1 and TTY windows, making **locate** act like **tab** in those windows.

OS/2, Windows

locate locates the cursor in the active window.

Remarks

m and n denote the row and column, respectively, at which the cursor is to be located.

The origin (1,1) is the upper left corner.

m and n may be any expressions that return scalars. Nonintegers will be truncated to an integer.

Example

```
r = csrlin;
c = csrcol;
cls;
locate r,c;
```

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

See also

csrlin, **csrcol**

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.

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.

Globals

_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.

Remarks

Based on "Robust Locally Weighted Regression and Smoothing Scatterplots", William S. Cleveland. 1979, _JASA_. 74:829-836.

Source

`loess.src`

Purpose

Computes the log base 10 of all elements of a matrix.

Format

y = **log(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix containing the log 10 values of the elements of x.

Remarks

log is defined for $x \neq 0$.

If x is negative, complex results are returned.

You can turn the generation of complex numbers for negative inputs on or off in the **GAUSS** configuration file, and with the **sysstate** function, case 8. If you turn it off, **log** will generate an error for negative inputs.

If x is already complex, the complex number state doesn't matter; **log** will compute a complex result.

x can be any expression that returns a matrix.

Example

```
x = round(rndu(3,3)*10+1);
y = log(x);
```

$$x = \begin{matrix} 4.0000000000 & 2.0000000000 & 1.0000000000 \\ 10.0000000000 & 4.0000000000 & 8.0000000000 \\ 7.0000000000 & 2.0000000000 & 6.0000000000 \end{matrix}$$

$$y = \begin{matrix} 0.6020599913 & 0.3010299957 & 0.3010299957 \\ 1.0000000000 & 0.6020599913 & 0.9030899870 \\ 0.8450980400 & 0.3010299957 & 0.7781512504 \end{matrix}$$

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;

math 401
```

See also

upper

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

$$L = \begin{matrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 1 & -2 & 1 \end{matrix}$$

$$L1 = \begin{matrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & -2 & 1 \end{matrix}$$

Source

diag.src

Globals

None

See also

upmat, **upmat1**, **diag**, **diagrv**, **crout**, **croutp**

Purpose

Returns the current position of the print head within the printer buffer for the printer.

Format

y = **lpos;**

Portability

This function is supported under DOS only.

Remarks

This function is basically equivalent to function **csrcol** but this returns the current column position for the standard printer.

The value returned is the column position of the next character to be printed to the printer buffer. This does not necessarily reflect the actual physical position of the print head at the time of the call.

If this function returns a number greater than 1, there are characters in the buffer for the standard printer which have not yet been sent to the printer. This buffer can be flushed at any time by **lprint**'ing a carriage return/line feed sequence, or a form feed character.

Example

```
if lpos > 60;
    lprint;
endif;
```

In this example, if the print buffer contains 60 characters or more, a carriage return/line feed sequence will be printed.

See also

lprint, **lpwidth**

Purpose

Controls printing to the standard printer, LPT1.

Format

lprint [[/*typ*]] [[/*fmted*]] [[/*mf*]] [[/*jnt*]] [[*list of expressions separated by spaces*]][[;]];

Portability

This function is supported under DOS only.

Remarks

lprint statements work in essentially the same way that **print** statements work. The main difference is that **lprint** statements cannot be directed to the auxiliary output. Also, the **locate** statement has no meaning with **lprint**.

Two semicolons following an **lprint** statement will suppress the final carriage return/line feed.

See under **print** for information on */typ*, */fmted*, */mf* and */jnt*.

A list of expressions is a list of **GAUSS** expressions, separated by spaces. In **lprint** statements, because a space is the delimiter between expressions, no spaces are allowed inside expressions unless they are within index brackets, they are in quotes, or the whole expression is in parentheses.

Printer width can be specified by the **lpwidth** statement:

```
lpwidth 132;
```

This statement remains in effect until cancelled. The default printer width is 80. That is, **GAUSS** automatically sends a line feed and carriage return to the printer after printing 80 characters.

lpos can be used to determine the (column) position of the next character that will be printed in the buffer.

An **lprint** statement by itself will cause a blank line to be printed:

```
lprint;
```

The printing of special characters is accomplished by the use of the backslash (\) within double quotes. The options are:

"\b"	**backspace** (ASCII 8)
"\e"	**escape** (ASCII 27)
"\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).

To beep the cursor, use "\007".

GAUSS also has an *automatic line printer mode* which causes the results of all global assignment statements to be printed out on the printer. This is controlled by the **lprint on**; and **lprint off**; commands. See the discussion under those headings.

Example

```
lprint 3*4 5+2;
```

See also

print, **lprint on**, **lpos**, **lpwidth**, **format**

Purpose

These commands switch the automatic line printer mode on and off.

Format

lprint on;

lprint off;

Portability

These functions are supported under DOS only.

Remarks

After the **lprint on** command is encountered, the results of any global assignment statements will be printed on the standard printer. Assignments to local matrices or strings will not be printed.

The name and dimensions of the resulting matrix will also be printed. If the result being assigned will be inserted into a submatrix of the target matrix, the name will be followed by an empty set of square brackets.

The **lprint off** command switches this feature off.

These commands are similar to the **print on** and **print off** commands, except that they control printing to the printer instead of to the screen or the auxiliary output.

Example

```
y = rndn(1000,3);
lprint on;
s = stdc(y)';
m = meanc(y)';
mm = diag(y'y)';
lprint off;

S[1,3]=
      1.021418       1.034924       1.040514

M[1,3]=
     -0.047845       0.007045      -0.035395

MM[3,1]=
   1045.583840    1071.118064    1083.923128
```

In this example, a large (1000×3) matrix of random numbers is created, and some statistics are calculated and printed out.

See also

print on

Purpose

Specifies the width of the printer.

Format

lpwidth *n*;

Portability

This function is supported under DOS only.

Remarks

n is a scalar which specifies the width of the printer in columns (characters). That is, after printing *n* characters on a line, **GAUSS** will send a carriage return and a line feed, so that the print head will move to the beginning of the next line.

If a matrix is being printed, the line feed sequence will always be inserted between separate elements of the matrix rather than being inserted between digits of a single element.

n may be any scalar-valued expression. Nonintegers will be truncated to an integer.

The default is 80 columns.

NOTE - This does not send control characters to the printer to automatically switch the mode of the printer to a different character pitch because each printer is different. This only controls the frequency of carriage return/line feed sequences.

Example

```
lpwidth 132;
```

This statement will change the printer width to 132 columns.

See also

lprint, **lpos**, **outwidth**

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.

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]**.

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;
```

$$x = \begin{matrix} 0.1523 + 0.7685i & -0.8957 + 0.0342i & 2.4353 + 2.7736i \\ -1.1953 + 1.2187i & 1.2118 + 0.2571i & -0.0446 - 1.7768i \\ 0.8038 + 1.3668i & 1.2950 - 1.6929i & 1.6267 + 0.2844i \end{matrix}$$

$$l = \begin{matrix} 0.2589 - 0.3789i & -1.2417 - 0.5225i & 1.0000 \\ 1.0000 & 0.0000 & 0.0000 \\ 0.2419 - 0.8968i & 1.0000 & 0.0000 \end{matrix}$$

$$u = \begin{matrix} -1.1953 + 1.2187i & 1.2118 + 0.2571i & -0.0446 - 1.7768i \\ 0.0000 & 0.7713 - 0.6683i & 3.2309 + 0.6742i \\ 0.0000 & 0.0000 & 6.7795 + 5.7420i \end{matrix}$$

$$x2 = \begin{matrix} 0.1523 + 0.7685i & -0.8957 + 0.0342i & 2.4353 + 2.7736i \\ -1.1953 + 1.2187i & 1.2118 + 0.2571i & -0.0446 - 1.7768i \\ 0.8038 + 1.3668i & 1.2950 - 1.6929i & 1.6267 + 0.2844i \end{matrix}$$

See also

crout, **croutp**, **chol**

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.

Remarks

If b has more than one column, each column is solved for separately, i.e., **lusol** solves $LUx[., i] = b[., i]$.

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**

Purpose

Returns a column vector containing the largest element in each column of a matrix.

Format

y = **maxc(**x**);**

Input

x	N×K 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);
```

$$x = \begin{matrix} -2.124474 & 1.376765 \\ 0.348110 & 1.172391 \\ -0.027064 & 0.796867 \\ 1.421940 & -0.351313 \end{matrix}$$

$$y = \begin{matrix} 1.421940 \\ 1.376765 \end{matrix}$$

See also

minc, **maxindc**, **minindc**

Purpose

Returns a column vector containing the index (i.e., row number) of the maximum element in each column in 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, **maxc** uses the complex modulus (**abs(**x**)**) to determine the largest elements.

To find the index of the maximum element in each row of a matrix, transpose the matrix before applying **maxindc**.

If there are two or more "largest" elements in a column (i.e., two or more elements equal to each other and greater than all other elements), then **maxindc** returns the index of the first one found, which will be the smallest index.

Example

```
x = round(rndn(4,4)*5);
y = maxc(x);
z = maxindc(x);
```

$$x = \begin{matrix} 1 & -11 & 0 & 5 \\ 0 & 0 & -2 & -6 \\ -8 & 0 & 3 & 2 \\ -11 & 5 & -4 & 5 \end{matrix}$$

$$y = \begin{matrix} 1 \\ 5 \\ 3 \\ 5 \end{matrix}$$

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

See also

maxc, **minindc**, **minc**

Purpose

Returns maximum vector length allowed.

Format

y = **maxvec;**

Global Input

__maxvec scalar, maximum vector length allowed.

Output

y	scalar, maximum vector length.

Remarks

maxvec returns the value in the global scalar **__maxvec**, which can be reset in the calling program. This must never be set to 8190.

maxvec is called by Run-Time Library functions and applications when determining how many rows can be read from a data set in one call to **readr**.

For the nonvirtual memory version you can use 536870910. For the virtual memory version, a smaller value like 20000-30000 is necessary to prevent excessive disk thrashing. The trick is to allow the algorithm making the disk reads to execute entirely in core. See Section B.3.

Example

```
y = maxvec;
print y;

  20000.000
```

Source

`system.src`

Globals

__maxvec

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} orders.

Remarks

For the functions that permit you to specify the order, the returned matrix contains a sequence of modified or exponentially scaled modified Bessel values of different orders. For the i^{th} row of y:

$$y[i,.] = \begin{array}{cccc} I_\alpha(x[i]) & I_{\alpha+1}(x[i]) & \cdots & I_{\alpha+n-1}(x[i]) \end{array}$$

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 (*Statistical Analysis of Circular Data*, N.I. Fisher, 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 (op.cit., page 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 (op. cit., page 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;
y = data[.,1];
x = data[.,2:cols(data)];

b0 = 2*ones(cols(x),1);

{ b,fct,grd,ret } = QNewton(&lpr,b0);

cov = invpd(hessp(&lpr,b));

print "estimates   standard errors";
print;
print b~sqrt(diag(cov));
```

Source

ribesl.src

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));
```

$$x = \begin{matrix} 0.492446 \\ 0.503543 \\ 0.502905 \\ 0.509283 \end{matrix}$$

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

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);
```

$$y = \begin{matrix} 6.0000000 \\ 7.0000000 \end{matrix}$$

Source

median.src

Globals

None

Purpose

Optional full-screen matrix editor. Handles both character and numeric elements.

Format

{ *y*,*yv*,*yfmt* } = **medit**(*x*,*xv*,*xfmt*);

Portability

This function is supported under DOS only.

Input

x L×M matrix to be edited.

xv scalar, vector or matrix. *xv* is reshaped into a 1×M vector. The nonzero elements in *xv* mark the respective columns of *x* as numeric, 0's mark them as character.

xfmt scalar, string or matrix. *xfmt* sets the initial column formats. If *xfmt* is a scalar then the following default formats are used:

Type	Format	Width	Precision
Numeric column	**"*.*lg "**	16	8
Character column	**"*.*s "**	8	8

If *xfmt* is a string, the first 8 characters are used for the format and the precision and field are set to a default value to give the global column format.

If *xfmt* is a matrix, it must have 3 columns. It will be reshaped to an M×3 format matrix.

In all cases, the type values in *xv* override the formats specified in *xfmt*.

Output

y L×M edited matrix.

yv 1×M vector of ones and zeros, 1 if the respective column of *y* is numeric, 0 if it is character.

yfmt M×3 matrix, each row containing format information (suitable for use with **printfm**) for the respective column of *y*.

Remarks

medit displays only the real part of the matrix if the imaginary part is missing or all zeros. The display changes to complex when an edited element has a nonzero imaginary part or a row or column is added with a complex fill value.

Alt-X terminates the editor. If you are editing an element or executing a command, press **Esc** to abandon the element or terminate the command. When **Alt-X** is used to terminate the editor, the edited matrix is saved in the file `_medit_x.fmt`. **Esc** also exits from the editor, but the results of the editing are not returned and the matrix is not saved to the file. The original value of the matrix remains unchanged.

The following keys allow you to move around the matrix.

Left	Move one cell left.
Right	Move one cell right.
Up	Move one cell up.
Down	Move one cell down.
PgUp	Move one page up.
PgDn	Move one page down.
Ctrl-Left	Move one page left.
Ctrl-Right	Move one page right.
Home	Beginning of row.
Home Home	Beginning of matrix.
End	End of row.
End End	End of matrix.
Enter	Move forward one cell.
Backspace	Move back one cell.
Ctrl-Enter	Toggle the direction of **Enter** and **Backspace**.
Alt-G	Go to row and column.

The following alternate Wordstar keystrokes are supported.

Ctrl-G	**Del**	**Ctrl-R**	**PgUp**
Ctrl-H	**Backspace**	**Ctrl-C**	**PgDn**
Ctrl-S	**Left**	**Ctrl-A**	**Ctrl-Left**
Ctrl-D	**Right**	**Ctrl-F**	**Ctrl-Right**
Ctrl-E	**Up**	**Ctrl-Q S**	**Home**
Ctrl-X	**Down**	**Ctrl-Q D**	**End**

The operation of the **Enter** and **Backspace** keys depends on the **Ctrl-Enter** setting. **Ctrl-Enter** toggles the direction of movement for **Enter** from right to down to no movement. **Backspace** moves in the opposite direction to **Enter**. The arrow at the top left of the matrix indicates the current setting.

Alt-G allows you to jump to any element in the matrix. Pressing **Alt-G** prompts you for the row and column. Enter the row and column numbers separated by either a space or

a comma and then press **Enter** to execute the command. Invalid entries will cause a beep and can be edited. Pressing **Esc** will abandon the command.

The following editing keys are available to you when editing a matrix element.

Left	Move left.
Right	Move right.
Del	Delete character at cursor.
Backspace	Delete character to left of cursor.
Home	Move to beginning of input.
End	Move to end of input.
Enter	Save the new value.
Ctrl-Enter	Toggle the direction of movement.
Alt-P	Inserts π on the input line.
Alt-E	Inserts e on the input line.
Esc	Abandon editing of the element.

The current element is displayed at the top of the screen. Character elements are displayed surrounded by " " quotes. Numeric elements are displayed in exponential format to full precision. To edit the current element, just type in the new value. To save the new value, press **Enter**. If the value is valid, it will be saved. Invalid values will cause a beep and can be edited. Pressing **Esc** abandons the editing and leaves the element unchanged.

Numbers can be entered in either integer, floating point or exponential format. To enter a missing value in a numeric element, type '**.**' and press **Enter**. If the numeric value entered overflows, ∞ with the appropriate sign will be stored. If the numeric value entered underflows, +0 will be stored.

Complex numbers are of the form *number* $\pm$ *number*i or *number*i, e.g., 0.5 − 1i, . + .i, 5i. Note that there must be a number before the 'i', i.e., 5+i is invalid. If the complex part is zero it is not displayed by **medit** or printed by **printfm**.

Character elements are limited to 8 characters. To enter an empty character string, press **Space** and then **Backspace** to delete the space, then press **Enter** to store the empty string. All strings are padded to 8 characters with nulls before storing.

The following command keys are available to you when moving around the matrix.

Alt-H	Help.
Esc	Abandon matrix and quit **medit**.
Alt-X	Exit **medit**. Result saved in `_medit_x.fmt`.
alt-r	Select rows.
Alt-C	Select columns.
Alt-V	Set fill value, use " " for character fill values.
Alt-I or **Ins**	Insert row, column or block.
Alt-F	Format column.

Alt-R and **Alt-C** allow a block of rows or columns to be selected using any of the cursor movement keys. The selected block can then be cut to the scrap buffer using **Grey -**, copied using **Grey +**, or the block can be deleted by pressing **Del**.

Alt-F allows the format of the current column to be changed. **Alt-F N** sets the default numeric format, **Alt-F C** sets the default character format and **Alt-F E** allows you to edit the column format. The variable type vector *yv* and format matrix *yfmt* are updated to reflect the new format.

Using the format editor, the column format can be selected as well as the trailing character, column width and precision. The format editor is intelligent and will automatically adjust the column width or precision to ensure all numbers in the column can be displayed in the selected format. Very large numbers in the column will cause the decimal format to automatically switch to an exponential format.

printfm can be used to print the matrix in the format displayed by **medit**.

```
{ x,v,fmt } = medit(x,1,1);
call printfm(x,v,fmt);
```

Note however that UNN's in **medit** are printed as numbers by **printfm**.

Alt-V sets the fill value to be used for inserting new rows and columns. Pressing **Alt-V** displays the current value and allows a new value to be entered. Enclose character values with " ". Character values must not exceed 8 characters. **Esc** abandons the entry of the new value and leaves the current fill value unchanged. The default fill value is a missing value.

The insert commands work with the fill value to insert a new row or column into the matrix in front of the current cursor position. The new row or column is filled with the current fill value. **Alt-I R** inserts a new row and **Alt-I C** inserts a new column. For columns, the type of the fill value determines the type of the column. The **Alt-I B** command allows a previously cut or copied block to be inserted in front of the cursor position. **Ins** can be used instead of **Alt-I**.

If you have a CGA display adaptor and **medit** is causing snow or flicker when you page up or page down, edit the `medit.src` file to change CGA to 1.

Example

```
{ x,v,fmt } = medit(zeros(10,5),1,1);
```

This example edits a matrix of zeros and assigns the result to **x**. The second and third arguments are reshaped to the correct size and the column formats are set to the default numeric format. These vectors are then updated by the format commands during the editing and the results assigned to **v** and **fmt**.

```
{ x,v,fmt } = medit(x,1~0,"le");
```

This example sets all the odd columns to a scientific format and sets all the even columns to the default character format.

```
def_fmt = "-lf,"~4~3;
{ x,v,fmt } = medit(x,1,def_fmt);
```

This example sets all the columns to decimal format, left-justified with a trailing comma. It also sets the precision to 3 and the minimum width to 4. The necessary width will be automatically determined by **medit**.

Source

`medit.src`

Globals

_med_1, **_med_2R**, **_med_2C**

See also

printfm, **con**, **editm**

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 the key variable in the first column.
2. All of the values of the key variable within a file are unique.
3. Each file is already sorted on that variable.

The output file will contain the key variable in its first column.

It is not necessary for the two files to have the same number of rows. For each row for which the key variables match, a row will be created in the output file. *outfile* will contain the columns from *infile1* followed by the columns of *infile2* minus the key column from the second file.

If the inputs are null or 0, the procedure will ask for them.

Example

```
mergeby("freq","freqdata","mergex",1);
```

Source

sortd.src

Globals

None

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

Globals

None

See also

makevars

Purpose

Returns a column vector containing the smallest element in each column in a matrix.

Format

y = **minc(**x**);**

Input

x	N×K 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);
```

$$x = \begin{matrix} -1.061321 & -0.729026 \\ -0.021965 & 0.184246 \\ 1.843242 & -1.847015 \\ 1.977621 & -0.532307 \end{matrix}$$

$$y = \begin{matrix} -1.061321 \\ -1.847015 \end{matrix}$$

See also

maxc, **minindc**, **maxindc**

Purpose

Returns a column vector containing the index (i.e., row number) of the smallest element in each column in 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);
```

$$x = \begin{matrix} -5 & 6 & -4 & -1 \\ 2 & -2 & 1 & 3 \\ 6 & 0 & 1 & -7 \\ -6 & 0 & 8 & -4 \\ 7 & -4 & 8 & 3 \end{matrix}$$

$$y = \begin{matrix} -6 \\ -4 \\ -4 \\ -7 \end{matrix}$$

$$z = \begin{matrix} 4 \\ 5 \\ 1 \\ 3 \end{matrix}$$

See also

maxindc, **minc**, **maxc**

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**.

In other matrix operations, missing values will be propagated in the obvious way as long as the **disable** command is on. For instance, the sum or product of a missing and anything else is a missing. A report is given at the end of a program if missing values have been encountered. If **enable** is on, then the program is stopped when a missing value is encountered in a standard matrix operation and an error message is given.

As long as you know a matrix contains no missings to begin with, **miss** and **missrv** can be used to convert one set of numbers into another. For example:

y=**missrv(miss(***x***,0),1);**

will convert 0's to 1's.

Example

```
v = -1~4~5;
y = miss(x,v);
```

If **x** has 3 columns, all -1's in the first column will be changed to missings, along with all 4's in the second column and 5's in the third column.

See also

/, **counts**, **ismiss**, **maxc**, **maxindc**, **minc**, **minindc**, **missex**, **moment**, **packr**, **scalmiss**, **sortc**

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`

Globals

None

See also

miss, **missrv**

Purpose

Computes a cross-product matrix. This is the same as $x'x$.

Format

y = **moment(**x,d**);**

Input

x	N×K matrix.
d	scalar, controls handling of missing values.

0	missing values will not be checked for. This is the fastest option.
1	"listwise deletion" is used. Any row that contains a missing value in any of its elements is excluded from the computation of the moment matrix. If every row in x contains missing values, then **moment(**x**,1)** will return a scalar zero.
2	"pairwise deletion" is used. Any element of x that is missing is excluded from the computation of the moment matrix. Note that this is seldom a satisfactory method of handling missing values, and special care must be taken in computing the relevant number of observations and degrees of freedom.

Output

y	K×K matrix which equals $x'x$.

Remarks

The fact that the moment matrix is symmetric is taken into account to cut execution time almost in half.

If there is no missing data then $d = 0$ should be used because it will be faster.

The **/** operator (matrix division) will automatically form a moment matrix (performing pairwise deletions if **trap** 2 is set) and will compute the **ols** coefficients of a regression. However, it can only be used for data sets that are small enough to fit into a single matrix. In addition, the moment matrix and its inverse cannot be recovered if the **/** operator is used.

Example

```
xx = moment(x,2);
ixx = invpd(xx);
b = ixx*missrv(x,0)'y;
```

In this example, the regression of y on x is computed. The moment matrix (xx) is formed using the **moment** command (with pairwise deletion, since the second parameter is 2). Then xx is inverted using the **invpd** function. Finally, the **ols** coefficients are computed. **missrv** is used to emulate pairwise deletion by setting missing values to 0.

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.

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.

__con global scalar, default 1.

1	a constant term will be added.
0	no constant term will be added.

__miss global 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 global scalar, default 0, the number of rows to read per iteration of the read loop.

If 0, the number of rows will be calculated internally.

If you get an "Insufficient memory" error, 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

Globals

__con, **__miss**, **__row**, **indices**, **maxvec**

Purpose

This command 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**, **lprint**, and **printfm** statements.

When converting ASCII to binary in **loadm** and **let** statements, only the first character is significant. In other words,

```
msym HAT;
```

will cause 'H' to be converted to missing on input.

This does not affect **writer** which outputs data in binary format.

See also

print, **lprint**, **printfm**

Purpose

Provides support for programs following the upper/lowercase convention in **GAUSS** data sets. See the *File I/O* chapter in Volume I. Returns a vector of names of the correct case and a 1/0 vector of type information.

Format

{ *vname*,*vtype* } = **nametype(***vname*,*vtype***);**

Input

vname	N×1 character vector of variable names.
vtype	scalar or N×1 vector of 1's and 0's to determine the type and therefore the case of the output *vname*. If this is scalar 0 or 1 it will be expanded to N×1. If -1 **nametype** will assume that *vname* follows the upper/lowercase convention.

Output

vname	N×1 character vector of variable names of the correct case, uppercase if numeric, lowercase if character.
vtype	N×1 vector of ones and zeros, 1 if variable is numeric, 0 if character.

Example

```
vn = { age, pay, sex };
vt = { 1, 1, 0 };
{ vn, vt } = nametype(vn,vt);
print $vn;
```

$$vn = \begin{matrix} AGE \\ PAY \\ sex \end{matrix}$$

Source

nametype.src

Globals

vartype

Purpose

Examines the status of the math coprocessor and checks whether any exceptions have been generated.

Format

y = **ndpchk(***mask***);**

Portability

This function is supported under DOS only. Exceptions are masked in Unix, OS/2 and Windows, so **ndpchk** will never see them.

Input

mask	scalar mask value used to test various bits in the math coprocessor status word.

Output

y	scalar, the result of a bitwise logical AND of the status word and the mask.

Remarks

Relevant values that can be used to check various exceptions are:

1	Invalid Operation
2	Denormalized Operand
4	Zero Divide
8	Overflow
16	Underflow

For example, suppose you want your program to check to see if an underflow has occurred during a critical calculation. The statement:

```
y = ndpchk(16);
```

will return 16 if the math coprocessor has detected an underflow and zero otherwise. It performs a bitwise logical AND using the argument and the math coprocessor status flags and returns the result.

This does not test the math coprocessor status word directly, but tests a system variable in **GAUSS** where we keep track of the flags that have been set by ORing the status word with the variable.

The values can be added together to test more than one flag in a single call. For example:

```
if ndpchk(24);
    gosub error;
endif;
```

This code would call the subroutine **error** if either overflow or underflow had occurred in a program.

GAUSS keeps track of all the exception flags in the math coprocessor as it executes. All exceptions are supported with interrupt handling code. Zero Divide and Invalid Operation are enabled by default. **ndpcntrl** is used to get and set the control word. You can also enable or disable the Invalid Operation interrupt with the **enable** and **disable** commands. For example, if you want to add two matrices together that contain the missing value codes, you will have to disable the Invalid Operation interrupt or the program will stop and print an error message. If exceptions are generated during the execution of a program, a report will be given when **GAUSS** drops to command level.

You can refer to the literature available from Intel for complete information on coprocessor exceptions. This is the best reference for those who want to understand how the math coprocessor works. See the *iAPX 86/88,186/188 User's Manual*, Intel Corp., 1983 available from Intel.

Another excellent reference is: Startz, R. 8087 *Applications and Programming for the IBM PC and Other PCs*, Robert J. Brady, 1983.

See also

ndpclex

Purpose

Clears the math coprocessor exception flags.

Format

ndpclex;

Portability

This function is supported under DOS only.

Input

None

Output

The flags noted above are cleared.

Remarks

This will prevent the exceptions message that can occur when **GAUSS** drops to command level.

See also

error, **scalerr**, **trapchk**, **trap**, **ndpchk**

Purpose

To get and set the numeric processor control word.

Format

y = **ndpcntrl(***new*,*mask***);**

Portability

This function is supported under DOS only.

Input

new	scalar, integer in the range 0-65535 containing the new bit values for the control word.
mask	scalar, integer in the range 0-65535. If a bit in *mask* is 1, the corresponding bit in the control word will be set to the corresponding bit in *new*. If a bit in *mask* is 0, the corresponding bit in the control word will be left as is.

Output

y	scalar, the new control word.

Remarks

We assume familiarity with the 80x87 family of math coprocessors. Here are the mask values for the various bits in the control word in hexadecimal:

0x003f	**Interrupt Exception Masks**
0x0001	invalid
0x0002	denormal
0x0004	zero divide
0x0008	overflow
0x0010	underflow
0x0020	inexact (precision)

0x1000	**Infinity Control**
0x1000	affine
0x0000	projective

0x0c00	**Rounding Control**
0x0c00	chop
0x0800	up
0x0400	down
0x0000	near

0x0300	**Precision Control**
0x0000	24 bits
0x0200	53 bits
0x0300	64 bits

For those of you who are fluent in C:

```
new_control = ((old_control & ~mask) | (new & mask))
```

The bold values above are the mask values in hexadecimal for the various functions that are controlled by the NDP control word.

The inexact result exception mask is set whenever rounding occurs. For example:

```
y = 1/3;
```

This is not a very useful exception to unmask in most **GAUSS** programs.

Changing these values will not have a completely global effect because some functions locally change control word values.

Example

```
oldtrol = ndpcntrl(0,0);
call ndpcntrl(0x0800,0x0c00);
  .
  .
  .
call ndpcntrl(oldtrol,0xffff);
```

In the example above, the original control word value is retrieved in the variable **oldtrol**. Nothing in the control word can be changed if the mask value is zero. In the second line, the rounding control is set to round up. In the last line, the original control word value is restored.

```
oldtrol = ndpcntrl(0,0);
call ndpcntrl(0x0004,0x0004);
  .
  .
  .
call ndpcntrl(oldtrol,0xffff);
```

In the example above, the original control word value is retrieved in the variable **oldtrol**. Nothing in the control word can be changed if the mask value is zero. In the second line, the zero divide exception is masked. In the last line, the original control word value is restored.

```
call ndpcntrl(0x0001+0x0020,0x003f);
```

In this example, invalid operation and inexact result are masked. All other exceptions are unmasked.

```
call ndpcntrl(0x0004,0x0004+0x0001);
```

In this example, zero divide is masked and invalid operation is unmasked. All other exceptions are left as is because the mask value allowed only these two bits to be changed.

Purpose

Erases everything in memory including the symbol table; closes all open files, the auxiliary output and turns the screen on if it was off; also allows the size of the new symbol table and the main program space to be specified.

Format

new ⟦*nos* ⟦,*mps*⟧⟧;

Input

nos scalar, which indicates the maximum number of global symbols allowed. See your supplement for the maximum number of globals allowed in this implementation.

mps scalar, which indicates the number of bytes of main program space to be allocated. See your supplement for the maximum amount allowed in this implementation.

Portability

Unix

new closes all open windows except window 1.

OS/2, Windows

new closes all user created windows.

Remarks

Procedures, user-defined functions, and global matrices strings and string arrays are all global symbols.

The main program space is the amount of memory available for nonprocedure, nonfunction program code.

This command can be used with arguments as the first statement in a program to clear the symbol table and to allocate only as much space for program code as your program actually needs. When used in this manner, the auxiliary output will not be closed. This will allow you to open the auxiliary output from command level and run a program without having to remove the **new** at the beginning of the program. If this command is not the first statement in your program, it will cause the program to terminate.

Example

```
new;              /* clear global symbols. */

new 300;          /* clear global symbols,set maximum
                     number of global symbols to 300,
                     and leave program space
                     unchanged. */

new 200,10000;    /* clear global symbols, set
                     maximum number of global symbols
                     to 200, and allocate 10000 bytes
                     for main program code. */

new ,10000;       /* clear global symbols, allocate
                    10000 bytes for main program
                    code, and leave maximum number
                    of globals unchanged. */
```

See also

clear, **delete**, **output**

Purpose

Returns allowable matrix dimensions for computing FFT's.

Format

n = **nextn(***n0***);**

n = **nextevn(***n0***);**

Input

n0	scalar, the length of a vector or the number of rows or columns in a matrix.

Output

n	scalar, the next allowable size for the given dimension for computing an FFT or RFFT. $n >= n0$.

Remarks

nextn and **nextnevn** determine allowable matrix dimensions for computing FFT's. The Temperton FFT routines (see table below) can handle any matrix whose dimensions can be expressed as:

$$2^p \times 3^q \times 5^r \times 7^s, \qquad \begin{array}{l} p, q, r \text{ nonnegative integers} \\ s = 0 \text{ or } 1 \end{array}$$

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`

Globals

None

See also

fftn, **optn**, **optnevn**, **rfftn**, **rfftnp**

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**

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:

$$x * b = 0 \quad \text{(N×K matrix of zeros)}$$

and

$$b'b = I \quad \text{(M×M identity matrix)}$$

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**

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.

If *dataset* is a null string, the procedure assumes that the actual data has been passed in the next two arguments.

depvar dependent variable.

If *dataset* contains the name of a **GAUSS** data set, this is interpreted as:

string name of dependent variable.

scalar index of dependent variable. If scalar 0, the last column of the data set will be used.

If *dataset* is a null string or 0, this is interpreted as:

N×1 vector the dependent variable.

indvars independent variables.

If *dataset* contains the name of a **GAUSS** data set, this is interpreted as:

K×1 character vector names of independent variables.

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, this is interpreted as:

N×K matrix the independent variables.

Defaults are provided for the following global input variables, so they can be ignored unless you need control over the other options provided by this procedure.

__altnam global vector, default 0.

This can be a (K+1)×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 global 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 global scalar, default 0.

0 there are no missing values (fastest).

1 listwise deletion, drop any cases in which missings occur.

2 pairwise deletion, this is equivalent to setting missings to 0 when calculating m. The number of cases computed is equal to the total number of cases in the data set.

__output global scalar, default 1.

1 print the statistics.

0 do not print statistics.

__row global scalar, the number of rows to read per iteration of the read loop. Default 0.

If 0, the number of rows will be calculated internally. If you get an "Insufficient memory" error while executing **ols**, you can supply a value for **__row** that works on your system.

The answers may vary slightly due to rounding error differences when a different number of rows is read per iteration. You can use **__row** to control this if you want to get exactly the same rounding effects between several runs.

_olsres global scalar, default 0.

1 compute residuals (*resid*) and Durbin-Watson statistic (*dwstat*).

0 *resid* = 0, *dwstat* = 0.

Output

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** was put into a file called `ols.out` as well as being printed on the screen. This example will compute a least squares regression of **y** on **x**. The return values were discarded by using a **call** statement.

```
data = "olsdat";
depvar = { score };
indvars = { region,age,marstat };
_olsres = 1;
output file = lpt1 on;
{ nam,m,b,stb,vc,std,sig,cx,rsq,resid,dbw } =
    ols(data,depvar,indvars);
output off;
```

In this example, the data set `olsdat.dat` was used to compute a regression. The dependent variable is **score**. The independent variables are: **region**, **age**, and **marstat**. The residuals and Durbin-Watson statistic will be computed. The output will be sent to the printer as well as the screen and the returned values are assigned to variables.

Source

`ols.src`

Globals

dotfeq, **indexcat**, **indices2**, **maxvec**, **_olsres**, **_olsrnam**, **__altnam**, **__con**, **__miss**, **__output**, **__row**, **__vpad**

See also

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.
_olsqtol	global 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`

Globals

_olsqtol, **_qrdc**, **_qrsl**

See also

ols, **olsqr2**, **orth**, **qqr**

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.
_olsqtol	global 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`

Globals

_olsqtol, **_qrdc**, **_qrsl**

See also

olsqr, **orth**, **qqr**

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);
```

$$x = \begin{matrix} 1.000000 & 1.000000 \\ 1.000000 & 1.000000 \\ 1.000000 & 1.000000 \end{matrix}$$

See also

zeros, **eye**

Purpose

Opens an existing **GAUSS** data file.

Format

open *fh=filename* ⟦**for** *mode*⟧ ⟦**varindxi** ⟦*offs*⟧ ⟧ **;**

Input

filename literal or ^string.

filename is the name of the file on the disk. The name can include a path if the directory to be used is not the current directory. This filename will automatically be given the extension `.dat`. If an extension is specified, the `.dat` will be overridden. If the file is an `.fmt` matrix file, the extension must be explicitly given. If the name of the file is to be taken from a string variable, the name of the string must be preceded by the `^` (caret) operator.

mode ⟦**read**⟧, **append**, **update**

The modes supported with the optional **for** subcommand are:

read This is the default file opening mode and will be the one used if none is specified. Files opened in this mode cannot be written to. The pointer is set to the beginning of the file and the **writer** function is disabled for files opened in this way. This is the only mode available for matrix files (`.fmt`), which are always written in one piece with the **save** command.

append Files opened in this mode cannot be read. The pointer will be set to the end of the file so that a subsequent write to the file with the **writer** function will add data to the end of the file without overwriting any of the existing data in the file. The **readr** function is disabled for files opened in this way. This mode is used to add additional rows to the end of a file.

update Files opened in this mode can be read from and written to. The pointer will be set to the beginning of the file. This mode is used to make changes in a file.

offs scalar.

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* is an offset that will be added to the index variables. This is useful if data from multiple files are concatenated horizontally in one matrix. It can be any scalar expression. The default is 0.

The index variables are useful for creating submatrices of specific variables without requiring that the positions of the variables be known. For instance, if there are two variables, **xvar** and **yvar** in the data set, the index variables will have the names **ixvar**, **iyvar**. If **xvar** is the first column in the data file, and **yvar** is the second, and if no offset, *offs*, has been specified, then **ixvar** and **iyvar** will equal 1 and 2 respectively. If an offset of 3 had been specified, then these variables would be assigned the values 4 and 5 respectively.

The **varindxi** and **varindx** options 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.

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 below.

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**.

On most systems the maximum number of files that can be simultaneously open is 10.

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 was in the string variable **fname**. In this example the file handle is tested to see if the file was opened successfully. The matrix **x** is written to this data set. The number of columns in **x** must be the same as the number of columns in the existing data set. The first row in **x** will be placed after the last row in the existing data set. The **writer** function will return the number of rows actually written. If this does not equal the number of rows that were attempted, then the disk is probably full.

```
open fin = mydata for read;
open fout = mydata for update;
do until eof(fin);
    x = readr(fin,100);
    x[.,1 3] = ln(x[.,1 3];
    call writer(fout,x);
endo;
closeall fin,fout;
```

In the above example, the same file, `mydata.dat`, is opened twice with two different file handles. It is opened for read with the handle **fin**, and it is opened for update with the handle **fout**. This will allow the file to be transformed in place without taking up the extra space necessary for a separate output file. Notice that **fin** is used as the input

handle and **fout** is used as the output handle. The loop will terminate as soon as the input handle has reached the end of the file. Inside the loop the file is read into a matrix called **x** using the input handle, the data are transformed (columns 1 and 3 are replaced with their natural logs), and the transformed data is written back out using the output handle. This type of operation works 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 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 give the matrix **y**. The index variables will thus give the correct positions of the variables in **y**.

```
open fx = x.fmt;
i = 1; rf = rowsf(fx);
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 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

create, **close**, **closeall**, **readr**, **writer**, **seekr**, **eof**

Purpose

Returns optimal matrix dimensions for computing FFT's.

Format

n = **optn(**$n0$**);**

n = **optevn(**$n0$**);**

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 below) can handle any matrix whose dimensions can be expressed as:

$$2^p \times 3^q \times 5^r \times 7^s, \qquad \begin{array}{l} p, q, r \text{ nonnegative integers} \\ s = 0 \text{ or } 1 \end{array}$$

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

Globals

None

See also

fftn, **nextn**, **nextnevn**, **rfftn**, **rfftnp**

Purpose

Computes an orthonormal basis for the column space of a matrix.

Format

y = **orth**(x);

Input

x	N×K matrix.

Global Input

_orthtol	global 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);
```

$$y = \begin{matrix} -0.58123819 & -0.81373347 \\ -0.81373347 & 0.58123819 \end{matrix}$$

Source

qqr.src

Globals

_orthtol, **_qrdc**, **_qrsl**

See also

qqr, **olsqr**

Purpose

This command makes it possible to direct the output of **print** statements to two different places simultaneously. One output device is always the screen or standard output. The other can be selected by the user to be any disk file or other suitable output device such as a printer.

Format

output ⟦**file=***filename*⟧ ⟦**on**|**off**|**reset**⟧**;**

Input

filename literal or ^string.

The **file=***filename* subcommand selects the file or device to which output is to be sent.

If the name of the file is to be taken from a string variable, the name of the string must be preceded by the ^ (caret) operator.

The default file name is `output.out`.

on, **off**, **reset** literal, mode flag

- **on** opens the auxiliary output file or device and causes the results of all **print** statements to be sent to that file or device. If the file already exists, it will be opened for appending. If the file does not already exist, it will be created.
- **off** closes the auxiliary output file and turns off the auxiliary output.
- **reset** similar to the **on** subcommand, except that it always creates a new file. If the file already exists, it will be destroyed and a new file by that name will be created. If it does not exist, it will be created.

Portability

Unix, **OS/2**, **Windows**

output commands affect the active window. Each window "remembers" its own settings, even when it is no longer the active window.

Each window can have its own auxiliary output file, or several windows can write to the same output file.

filename can be any legal file name or applicable device. Therefore, **CON**, **LPT1**, **LPT2** and **LPT3** are all legal.

DOS only

In command mode, **Ctrl-F3** will close the auxiliary output file and load it into the editor. That is, after a program has been run, pressing **Ctrl-F3** will perform **output off** and put you into EDIT mode with the output file as the file being edited. The file must be closed prior to being edited.

Remarks

After you have written to an output file you have to close the file before you can print it or edit it with the **GAUSS** editor. Use **output off**.

The selection of the auxiliary output file or device remains in effect until a new selection is made, or until you 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 on the screen.

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** commands. 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 a **new** statement in COMMAND mode. 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 screen 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**

Purpose

Specifies the width of the auxiliary output.

Format

outwidth *n*;

Portability

Unix, **OS/2**, **Windows**

outwidth affects the active window. Each window "remembers" its own setting, even when it is no longer the active window.

Remarks

n specifies the width of the auxiliary output in columns (characters). After printing *n* characters on a line, **GAUSS** will output a carriage return and 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

lpwidth, **output**, **print**

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);
```

$$x = \begin{matrix} . & 9 & 10 \\ 4 & 2 & . \\ 3 & 4 & 9 \end{matrix}$$

$$y = \begin{matrix} 3 & 4 & 9 \end{matrix}$$

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**

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

Purpose

Pauses for a specified number of seconds.

Format

pause(*sec***);**

Input

sec	seconds to pause.

Source

`pause.src`

Globals

None

See also

wait

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);
```

$$x = \begin{matrix} -1.828915 & 0.514485 \\ -0.550219 & -0.275229 \end{matrix}$$

$$y = \begin{matrix} 0.074915 & 0.349488 \\ 0.342903 & 0.384115 \end{matrix}$$

Purpose

Returns the (scalar) mathematical constant π.

Format

$y =$ **pi;**

Example

```
format /rdn 16,14;
print pi;

3.14159265358979
```

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	global scalar, any singular values less than **_svdtol** are treated as zero in determining the rank of the input matrix. The default value for **_svdtol** is $1.0e-13$.

Output

y	M×N matrix that satisfies the 4 Moore-Penrose conditions:

$XYX = X$
$YXY = Y$
XY is symmetric
YX is symmetric

Global Output

_svderr	global scalar, if not all of the singular values can be computed **_svderr** will be nonzero.

Example

```
y = pinv(x);
```

$$y = \begin{matrix} 0.22017139 & -0.16348055 \\ -0.052076467 & 0.13447594 \\ -0.015161503 & 0.077125906 \end{matrix}$$

Source

svd.src

Globals

_svdtol, **_svderr**

Purpose

Plots the elements of two matrices against each other in text mode.

Format

plot x, y;

Portability

OS/2, **Windows**

plot writes to the main GAUSS window.

Unix

plot writes to the active window. It is supported only for Text windows.

Remarks

x and y are two matrices that must be conformable in the standard fashion of element-by-element operators. Horizontal elements come from the first matrix listed. Vertical coordinates come from the second matrix listed. Noninteger elements are rounded to the nearest integer to obtain the coordinates.

Plotting is done in text mode on the screen, not graphics mode.

All that is displayed by **plot** are the points being plotted. Axes are not automatically displayed.

The screen is treated as an axis system, with (0,0) in the lower left, and (79,24) in the upper right of the screen.

If coordinates fall outside of the allowable range, points are plotted along the corresponding edge of the screen.

The symbol or character to be plotted is controlled by the **plotsym** statement. The default symbol is an asterisk: *.

Example

```
print "Number of points: ";;
n = con(1,1);
cls;
x = floor(rndu(n,1)*80); y = floor(rndu(n,1)*25);
plot x,y;
```

In this example two matrices of uniform random numbers, scaled so their elements will fall in the right range, are plotted. The **con** command is used to ask you to specify the size of the matrices.

See also

graph, **plotsym**, **screen**

Purpose

Controls the symbol or character to be plotted by **plot**.

Format

plotsym *n*;

Remarks

n is a scalar containing the ASCII value of the symbol to be plotted. This value must be in the range 0-255.

n may be any legal expression that returns a scalar. Nonintegers will be truncated to an integer before being evaluated by **plotsym**.

The default symbol is an asterisk (*), which has ASCII value 42. Thus, if no **plotsym** statement is encountered, an asterisk will be used by **plot**.

The **plotsym** statement remains in effect until the next **plotsym** statement is encountered.

Example

```
plotsym 249;
```

Assuming a standard ASCII terminal font (as under DOS), this example will cause centered dots to be used as the symbol for **plot**.

See also

plot, **graph**, **screen**

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:

$$p(z) = c[1] * z^n + c[2] * z^{(n-1)} + \ldots + c[n] * z + c[n+1];$$

Remarks

The coefficient of z^n is set to unity ($c[1] = 1$).

Source

`poly.src`

Globals

polymake

See also

polymake, **polymult**, **polyroot**, **polyeval**

Purpose

Evaluates polynomials. Can either be 1 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).

Source

poly.src

Globals

None

See also

polymake, **polychar**, **polymult**, **polyroot**

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.
_poldeg	global scalar, the degree of polynomial required, default 6.

Output

y	result of interpolation or extrapolation.
_polerr	global scalar, interpolation error.

Remarks

Calculates an N^{th} order polynomial interpolation or extrapolation of X on Y given the vectors xa and ya and the scalar x. The procedure uses Neville's algorithm to determine an up to N^{th} order polynomial and an error estimate.

Polynomials above degree 6 are not likely to increase the accuracy for most data. Test **_polerr** to determine the required **_poldeg** for your problem.

Technical Notes

Press, W.P., Flannery, B.P., Teukolsky, S.A., and Vettering, W.T., (1986), *Numerical Recipes: The Art of Scientific Computing*, (New York: Cambridge Press)

Source

`polyint.src`

Globals

_polerr, **_poldeg**

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);
```

$$c = \begin{matrix} 1.0000000 \\ -6.0000000 \\ 11.000000 \\ -6.0000000 \end{matrix}$$

Source

`poly.src`

Globals

None

See also

polychar, **polymult**, **polyroot**, **polyeval**

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 contain the second powers, and so on.

Remarks

To do polynomial regression use **ols**:

```
{ vnam,m,b,stb,vc,stderr,sigma,cx,rsq,resid,dwstat } =
            ols(0,y,polymat(x,p));
```

Source

`polymat.src`

Globals

None

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.

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.)

Example

```
c1 = { 2, 1 };
c2 = { 2, 0, 1 };
c = polymult(c1,c2);
```

$$c = \begin{matrix} 4.0000000 \\ 2.0000000 \\ 2.0000000 \\ 1.0000000 \end{matrix}$$

Source

poly.src

Globals

None

See also

polymake, **polychar**, **polyroot**, **polyeval**

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:

$$p(z) = c[1] * z^n + c[2] * z^{(n-1)} + \ldots + c[n] * z + c[n+1]$$

Zero leading terms will be stripped from c. When that occurs the order of y will be the order of the polynomial after the leading zeros have been stripped.

$c[1]$ need not be normalized to unity.

Output

y	N×1 vector, the roots of c.

Source

`poly.src`

See also

polymake, **polychar**, **polymult**, **polyeval**

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 the statements:

```
goto label(x,y,z);
  .
  .
  .
label:
    pop c;
    pop b;
    pop a;
```

$c = z$ $b = y$ $a = x$

Note that matrices are **pop**'ped in reverse order, and that there is a separate **pop** statement for each matrix popped.

See also

gosub, **goto**, **return**

Purpose

Sets the computational precision of some of the matrix operators.

Format

prcsn n**;**

Input

n	scalar, 64 or 80

Portability

Unix, **Windows**

This function has no effect under Unix or Windows. All computations are done in 64-bit precision (except for operations done entirely within the 80x87 on Intel machines).

Remarks

n is a scalar containing either 64 or 80. The operators affected by this command are **chol**, **solpd**, **invpd**, and b/a (when neither a nor b is scalar and a is not square).

prcsn 80 is the default. Precision is set to 80 bits (10 bytes), which corresponds to about 19 digits of precision.

prcsn 64 sets the precision to 64 bits (8 bytes), which is standard IEEE double precision. This corresponds to 15–16 digits of precision. 80 bit precision is still maintained within the 80x87 math coprocessor so that actual precision is better than double precision.

When **prcsn** 80 is in effect, all temporary storage and all computations for the operators listed above are done in 80 bits. When the operator is finished, the final result is rounded to 64 bit double precision.

See also

chol, **solpd**, **invpd**

Purpose

Prints matrices or strings to the screen and/or auxiliary output.

Format

print ⟦**/flush**⟧ ⟦*/typ*⟧ ⟦*/fmted*⟧ ⟦*/mf*⟧ ⟦*/jnt*⟧ *list_of_expressions*⟦**;**⟧**;**

Portability

Unix, **OS/2**, **Windows**

print writes to the active window. The refresh setting of the window determines when output is displayed. To force immediate display, use the **/flush** flag:

print /flush *expr1 expr2* **...;;**

Input

/typ literal, symbol type flag.

/mat, /sa, /str Indicate which symbol types you are setting the output format for: matrices (**/mat**), string arrays (**/sa**), and/or strings (**/str**). You can specify more than one */typ* flag; the format will be set for all types indicated. If no */typ* flag is listed, **print** assumes **/mat**.

/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.

/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 an **/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 with a comma.

t The number will be followed immediately with a tab character.

n No trailing character.

The default when **GAUSS** is first started is:

```
format /m1 /r0 16,8;
```

;; double semicolons following a **print** statement will suppress the final carriage return/line feed.

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 screen. See the discussion of the **output** command. 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.

A **print** statement by itself will cause a blank line to be printed:

```
print;
```

GAUSS also has an *automatic print mode* which causes the results of all global assignment statements to be printed out. This is controlled by the **print on** and **print off** commands. See the discussion under **print on**.

Example

```
x = rndn(3,3);
format /rd 16,8;
   print x;
format /re 12,2;
   print x;
   print /rd/m3 x;

    0.14357994      -1.39272762      -0.91942414
    0.51061645      -0.02332207      -0.02511298
   -1.54675893      -1.04988540       0.07992059

 1.44E-001   -1.39E+000   -9.19E-001
 5.11E-001   -2.33E-002   -2.51E-002
-1.55E+000   -1.05E+000    7.99E-002

Row 1
        0.14        -1.39        -0.92
Row 2
        0.51        -0.02        -0.03
Row 3
       -1.55        -1.05         0.08
```

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

lprint, **print on**, **lprint on**, **printfm**, **printdos**

Purpose

Switches the automatic screen print mode on and off.

Format

print [[**on**|**off**]];

Portability

Unix, **OS/2**, **Windows**

print on|**off** affects the active window. Each window "remembers" its own setting, even when it is no longer the active window.

Remarks

After the **print on** command is encountered, the results of any global assignment statements (that is, statements that make assignments to global matrices or strings) will be printed on the screen. Assignments to local matrices or strings will not be printed.

The name and dimensions of the resulting matrix will also be printed. If the result being assigned will be inserted into a submatrix of the target matrix, the name will be followed by an empty set of square brackets.

Example

```
y = rndn(1000,3);
print on;
s = stdc(y)';
m = meanc(y)';
mm = diag(y'y);
print off;

S[1,3]=
      1.021418       1.034924       1.040514

M[1,3]=
     -0.047845       0.007045      -0.035395

MM[3,1]=

   1045.583840
   1071.118064
   1083.923128
```

In this example, a large (1000×3) matrix of random numbers is created, and some statistics are printed out using **print on**.

See also

lprint on

Purpose

Prints a string to the standard output.

Format

printdos *s*;

Input

s	string, containing the string to be printed to the standard output.

Portability

Unix, **OS/2**, **Windows**

printdos writes to the active window. The refresh setting of the window determines when output is displayed. To force immediate display, use the **WinRefresh** command.

DOS

printdos uses the **DOS** interrupt 21 hex, function 9. The **DOS** function 9 uses a dollar sign as a terminator, so this function will not print a dollar sign.

Remarks

This function is useful for printing messages to the screen when **screen off** is in effect. The output of this function will not go to the auxiliary output.

This function can also be used to send escape sequences to the `ansi.sys` device driver.

Example

```
printdos "\27[7m";  /* set for reverse video */
printdos "\27[0m";  /* set for normal text   */
```

See the **DOS** manuals for more complete information.

Technical Notes

See also

print, **lprint**, **printfm**, **screen**

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 string (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.

Portability

Unix, **OS/2**, **Windows**

printfm writes to the active window. The refresh setting of the window determines when output is displayed. To force immediate display, use the **WinRefresh** command.

Remarks

The mask is applied to the matrix *x* following the rules of standard element-by-element operations. If the corresponding element of mask is 0, then that element of *x* is printed as a character string of up to eight 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 be able to use this pretty much as you please.

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+000
PAY 1.235, 1.2346E+000
SEX 1.145, 3.4471E+000
JOB 4.114, 8.5564E+000
```

When the column of x to be printed contains all string elements, use a format string of **"*.*s"** if you want it right-justified, or **"-*.*s"** if you want it left-justified. If the column is mixed string and numeric elements, then use the correct numeric format and **printfm** will substitute a default format string for those elements in the column that are strings.

Remember, the mask value controls whether an element will be printed as a number or a string.

See also

print, **lprint**, **printdos**

Purpose

Print character, numeric, or mixed matrix using a default format controlled by the globals **__fmtnv** and **__fmtcv**.

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

The global format vectors can be modifed by calls to **formatnv** for the numeric format and to **formatcv** for the character format.

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**

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 } = dog(i,j,k);
```

Returns no items:

```
dog(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.

See Chapter 9 for more details on writing procedures.

See also

keyword, **call**, **endp**, **local**, **retp**

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);
```

$$y = \begin{matrix} 28 \\ 80 \\ 162 \end{matrix}$$

See also

sumc, **meanc**, **stdc**

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, ASCII (0) or binary (1).
append	scalar, file write mode, overwrite (0) or append (1).

Output

ret	scalar, return code.

Remarks

If *mode* is set to binary (1), a string of length *len* will be written to *filename*. If *mode* is set to ASCII (0), 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 overwrite (0), the current contents of *filename* will be destroyed. If *append* is set to append (1), *filename* will be created if it does not already exist.

If an error occurs, **putf** will either return an error code or terminate the program with an error message, depending on the **trap** state. If bit 2 (the 4's bit) of the trap flag is 0, **putf** will terminate with an error message. If bit 2 of the trap flag is 1, **putf** will return an error code. The value of the trap flag can be tested with **trapchk**.

ret can have the following values:

0	normal return
1	null file name
2	file open error
3	file write error
4	output string too long
5	null output string, or illegal *mode* value
6	illegal *append* value
16	append (1) specified but file did not exist; file was created (warning only)

- ## Source

  putf.src

- ## See also

  getf

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.

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

Portability

Unix

If you are running in terminal mode, **GAUSS** will not see any input until you press **Enter**.

Remarks

Pressing C on the keyboard will terminate iterations, and pressing P will toggle iteration output.

Example

This example computes maximum likelihood coefficients and standard errors for a Tobit model:

```
/*
**  qnewton.e  -  a Tobit model
*/

z = 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;



Tobit Model
```

```
coefficients  standard errors

     0.010417884      0.080220019
     -0.20805753      0.094551107
    -0.099749592      0.080006676
      0.65223067      0.099827309
```


Globals

_qn_RelGradTol scalar, convergence tolerance for relative gradient of estimated coefficients. Default = 1e-5.

_qn_GradProc scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. This procedure must have a single input argument, a Kx1 vector of parameter values, and a single output argument, a Kx1 vector of gradients of the function with respect to the parameters evaluated at the vector of parameter values. If **_qn_GradProc** is 0, **QNewton** uses **gradp**.

_qn_MaxIters scalar, maximum number of iterations. Default = 1e+5. Termination can be forced by pressing C on the keyboard.

_qn_PrintIters scalar, if 1, print iteration information. Default = 0. Can be toggled during iterations by pressing P on the keyboard.

_qn_ParNames K×1 vector, labels for parameters

_qn_PrintResults scalar, if 1, results are printed.

Source

qnewton.src

Purpose

Solves the quadratic programming problem.

Format

{ *x*,*u1*,*u2*,*u3*,*u4*,*ret* } = **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 Mx1 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 Nx1 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 $\pm 1e200$.

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`

Globals

_qprog_maxit scalar, maximum number of iterations. Default = 1000.

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 Px(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`

Globals

_qrdc, **_qrsl**

See also

qre, **qrep**, **qtyr**, **qtyre**, **qtyrep**, **qyr**, **qyre**, **qyrep**, **olsqr**

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 = qrsol(Q′Y,R1) | zeros(N−P,1);

The explicit formation here of Q, which can be a very large matrix, can be avoided by using the function **qtyre**.

For further discussion of QR factorizations see the remarks under **qqr**.

Source

`qqr.src`

Globals

_qrdc, **_qrsl**

See also

qqr, **qtyre**, **olsqr**

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`

Globals

_qrdc, **_qrsl**

See also

qqr, **qre**, **olsqr**

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 Px(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`

Globals

_qrdc, **_qrsl**

See also

qqr, **qrep**, **qtyre**

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 = qrsol(Q′Y,R1) | zeros(N−P,1);

The explicit formation here of Q, which can be a very large matrix, can be avoided by using the function **qtyre**.

For further discussion of QR factorizations see the remarks under **qqr**.

Source

`qr.src`

Globals

_qrdc

See also

qqr, **olsqr**

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`

Globals

_qrdc

See also

qr, **qre**, **qqrep**

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**

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**

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

Globals

_qrdc, **_qrsl**

See also

qqr, **qtyre**, **qtyrep**, **olsqr**

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 = qrsol(Q′Y,R1) | zeros(N−P,1);

Source

qtyr.src

Globals

_qrdc, **_qrsl**

See also

qqr, **qre**, **qtyr**

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);
```

$$qty = \begin{matrix} -6.9347609 & -9.9498744 & -3.0151134 \\ 4.0998891 & 3.5527137e-15 & 2.1929640 \\ 3.4785054 & 6.3245553 & 0.31622777 \end{matrix}$$

$$r = \begin{matrix} -13.266499 & -9.6483630 & -8.1408063 \\ 0.0000000 & -0.95346259 & 4.7673129 \\ 0.0000000 & 0.0000000 & 3.1622777 \end{matrix}$$

$$e = \begin{matrix} 1.0000000 \\ 2.0000000 \\ 3.0000000 \end{matrix}$$

Source

qtyr.src

Globals

_qrdc, **_qrsl**

See also

qrep, **qtyre**

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);
```

$$qy = \begin{matrix} 4.6288991 & 9.0506281 \\ -3.6692823 & -7.8788202 \\ 3.1795692 & 1.0051489 \end{matrix}$$

$$r = \begin{matrix} -7.3484692 & -4.6268140 \\ 0.0000000 & 10.468648 \end{matrix}$$

Source

qyr.src

Globals

_qrdc, **_qrsl**

See also

qqr, **qyre**, **qyrep**, **olsqr**

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**.

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.

Example

```
x = { 1 11, 7 3, 2 1 };
y = { 2 6, 5 10, 4 3 };
{ qy, r, e } = qyre(y,x);
```

$$qy = \begin{matrix} -0.59422765 & -3.0456088 \\ -6.2442636 & -11.647846 \\ 2.3782485 & -0.22790230 \end{matrix}$$

$$r = \begin{matrix} -11.445523 & -2.9705938 \\ 0.0000000 & -6.7212776 \end{matrix}$$

$$e = \begin{matrix} 2.0000000 \\ 1.0000000 \end{matrix}$$

Source

qyr.src

Globals

_qrdc, **_qrsl**

See also

qqr, **qre**, **qyr**

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 Px(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`

Globals

_qrdc, **_qrsl**

See also

qr, **qqrep**, **qrep**, **qtyrep**

Purpose

Computes the rank of a matrix, using the singular value decomposition.

Format

k = **rank(**x**);**

Input

x	N×P matrix.

Global Input

_svdtol	global scalar, the tolerance used in determining if any of the singular values are effectively 0. The default value is 10^{-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	global scalar, if not all of the singular values can be computed **_svderr** will be nonzero.

Source

`svd.src`

Globals

_svderr, **_svdtol**

Purpose

Returns the vector of ranks (rank index) 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);
```

$$r = \begin{matrix} 4 \\ 1 \\ 5 \\ 2 \\ 3 \end{matrix}$$

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**

Purpose

Returns the real part of a matrix.

Format

zr = **real(***x***);**

Input

x	N×K matrix.

Output

zr	N×K matrix, the real part of *x*.

Remarks

If *x* is not complex, *zr* will be equal to *x*.

Example

```
x = { 1 11,
      7i 3,
      2+i 1 };
zr = real(x);
```

$$zr = \begin{matrix} 1.0000000 & 11.000000 \\ 0.0000000 & 3.0000000 \\ 2.0000000 & 1.0000000 \end{matrix}$$

See also

complex, **imag**

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);
```

$$x = \begin{matrix} 20 \\ 45 \\ 32 \\ 63 \\ 29 \end{matrix}$$

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

$$v = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

$$y = \begin{matrix} 20 \\ 3 \\ 2 \\ 63 \\ 1 \end{matrix}$$

Source

`datatran.src`

Globals

None

See also

code, **substute**

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**

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**

Purpose

Computes a recursive series involving division.

Format

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

Input

x	1×K or K×1 matrix.
z	N×K matrix (the columns of z must equal the number of elements in x).

Output

y N×K matrix in which each column is a series generated by a recursion of the form:

$$\begin{aligned} y[1] &= x \bmod z[1] \\ y[2] &= \text{trunc}(x/z[1]) \bmod z[2] \\ y[3] &= \text{trunc}(x/z[2]) \bmod z[3] \\ &\cdots \\ y[n] &= \text{trunc}(x/z[n-1]) \bmod z[n] \end{aligned}$$

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

Globals

None

See also

recserar, **recsercp**

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);
```

If $x = \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$ then $y = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{matrix}$

If $x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$ then $y = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 & 2 & 3 \end{matrix}$

If $x = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{matrix}$ then $y = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{matrix}$

If $x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$ then $y = \begin{matrix} 1 & 2 & 3 & 4 & 1 & 2 \\ 3 & 4 & 1 & 2 & 3 & 4 \end{matrix}$

If $x = 1$ then $y = \begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$

See also

submat, **vec**

Purpose

Return from a procedure or keyword.

Format

retp;

retp(x**,**y**,**...**);**

Remarks

For more details, see Chapter 9.

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**

Purpose

Return 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**

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);
```

$$x = \begin{matrix} 10 & 7 & 8 \\ 7 & 4 & -9 \\ -11 & 0 & -3 \\ 3 & 18 & 0 \\ 9 & -1 & 20 \end{matrix}$$

$$y = \begin{matrix} 9 & -1 & 20 \\ 3 & 18 & 0 \\ -11 & 0 & -3 \\ 7 & 4 & -9 \\ 10 & 7 & 8 \end{matrix}$$

See also

sortc

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 real 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 $1/N$.

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 = { 6 9, 8 1 };
y = rfft(x);
```

$$y = \begin{matrix} 6.0000000 & 1.0000000 \\ 1.5000000 & -2.5000000 \end{matrix}$$

See also

rffti, **fft**, **ffti**, **fftm**, **fftmi**

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);
```

$$y = \begin{matrix} 6.0000000 & 9.0000000 \\ 8.0000000 & 1.0000000 \end{matrix}$$

See also

rfft, **fft**, **ffti**, **fftm**, **fftmi**

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:

$$2^p \times 3^q \times 5^r \times 7^s, \qquad \begin{array}{l} p, q, r \geq 0 \\ s = 0 \text{ or } 1 \end{array}$$

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**

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:

$$2^p \times 3^q \times 5^r \times 7^s, \qquad \begin{array}{ll} p,q,r \geq 0 & \text{for rows of matrix} \\ p > 0, \quad q,r \geq 0 & \text{for columns of matrix} \\ p > 0, \quad q,r \geq 0 & \text{for length of vector} \\ s = 0 \text{ or } 1 & \text{for all dimensions} \end{array}$$

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% 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 **nextevn** 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**

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 spectrun or to perform filtering on the waveform. In both these cases only the positive frequencies are required. See **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:

$$2^p \times 3^q \times 5^r \times 7^s, \qquad \begin{array}{ll} p,q,r \geq 0 & \text{for rows of matrix} \\ p > 0, \quad q,r \geq 0 & \text{for columns of matrix} \\ p > 0, \quad q,r \geq 0 & \text{for length of vector} \\ s = 0 \text{ or } 1 & \text{for all dimensions} \end{array}$$

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% 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 **nextevn** 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**

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 spectrun or to perform filtering on the waveform. In both these cases only the positive frequencies are required. See **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**

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:

$$\begin{aligned} E(x) &= a/(a+b) \\ Var(x) &= a \times b/(a+b+1) \times (a+b)^2 \\ x &> 0 \\ x &< 1 \\ a &> 0 \\ b &> 0 \end{aligned}$$

Source

`random.src`

Purpose

Resets the parameters of the uniform and Normal random number generators.

Format

rndcon *c*;

rndmod *m*;

rndmult *a*;

rndseed *seed*;

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. All of the parameters of this generator must be integers in the range:
$0 < parameter \leq 2^{31} - 1$.

The procedure used to generate the uniform random numbers is as follows. First, the current "seed" is used to generate a new seed:

$$new\_seed = (a * seed + c) \ \% \ m$$

(where % is the mod operator). Then a number between 0 and 1 is created by dividing the new seed by m:

$$x = new\_seed \ / \ m$$

rndcon resets c. The default is $c = 0$.

rndmod resets m. The default is $m = 2^{31} - 1$.

rndmult resets a. The default is $a = 397204094$.

rndseed resets *seed*. This is the initial seed for the generator. The default is that **GAUSS** uses the system clock to generate an initial seed when **GAUSS** is invoked.

GAUSS goes to the clock to seed the generator only when it is first started up. Therefore, if **GAUSS** is allowed to run for a long time, and if large numbers of random numbers are generated, there is a possibility of recycling (that is, the sequence of "random numbers" will repeat itself). However, the generator used has an extremely long cycle, and so that should not usually be a problem.

The parameters set by these commands remain in effect until new commands are encountered, or until **GAUSS** is restarted.

See also

rndu, **rndn**, **rndus**, **rndns**

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`

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 or use the alternate function **rndns**. 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);
```

$$\begin{aligned} m &= 0.002810 \\ s &= 0.997087 \end{aligned}$$

In this example, a sample of 8100 Normal random numbers is drawn, and the mean and standard deviation are computed for the sample.

See also

rndu, **rndns**, **rndcon**

Technical Notes

This function uses the fast acceptance-rejection algorithm proposed by Kinderman, A. J. and J. G. Ramage, "Computer Generation of Normal Random Numbers," Journal of the American Statistical Association, December 1976, Volume 71, Number 356, pp. 893–896.

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:

$$\begin{aligned} E(x) &= k \times p/(1-p) \\ Var(x) &= k \times p/(1-p)^2 \\ x &= 0, 1, 2, \cdots, k \\ k &> 0 \\ p &> 0 \\ p &< 1 \end{aligned}$$

Source

`random.src`

Purpose

An alternate way of generating Normal (**rndns**) or uniform **(rndus**) random numbers using a seed value as one of the arguments.

Format

y = **rndns(**r,c,s**);**

y = **rndus(**r,c,s**);**

Input

r	scalar, row dimension.
c	scalar, column dimension.
s	scalar, starting seed.

Output

y	R×C matrix of numbers satisfying the properties of either the Normal distribution **(rndns**) or the uniform distribution (**rndus**).

Remarks

The value of the seed must be in the range: $0 < \text{seed} < 2^{31} - 1$.

The seed must be a variable; it cannot be a constant or an expression. It will be updated automatically during the call.

An example of the use of these functions is as follows:

```
seed1 = 3937841;
x = rndns(1000,2,seed1);
```

With this statement a 1000×2 matrix of Normal random variables will be created using the value in **seed1** as the starting seed. **seed1** will be updated during the call.

See also

rndn, **rndu**

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:

$$\begin{aligned} E(x) &= lambda \\ Var(x) &= lambda \\ x &= 0, 1, 2, \cdots \\ lambda &> 0 \end{aligned}$$

Source

`random.src`

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×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 or by using **rndus**.

The seed is automatically updated as a random number is generated (see above under **rndcon**). Thus, if **GAUSS** is allowed to run for a long time, and if large numbers of random numbers are generated, there is a possibility of recycling. This is a 32-bit generator, though, so the range is sufficient for most applications.

Example

```
x = rndu(8100,1);
y = meanc(x);
z = stdc(x);
```

$$\begin{aligned} y &= 0.500205 \\ z &= 0.289197 \end{aligned}$$

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., 1980, pp. 136-147.

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

Purpose

Round to the nearest integer.

Format

y = **round(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix 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**

Purpose

Returns the number of rows in a matrix.

Format

y = **rows(**x**);**

Input

x	N×K 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**

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**

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($m \mid n$**) * 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`

Globals

None

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 pathname is specified for the file, then no additional searching will be attempted if the file is not found.

```
run /gauss/myrog.prc;
```

If a pathname is not specified and a **src_path** configuration variable is defined, then if the file is not found, each path listed in **src_path** will be searched for the specified filename and the first one found will be run. Any path on the user-specified filename will be stripped off prior to being combined with the paths found in **src_path**.

```
run myprog.prc;

run /gauss/myprog.prc;
```

The **src_path** configuration variable can be set in the **GAUSS** configuration file.

Programs can also be run by typing the filename on the OS command line when starting **GAUSS**.

Example

```
run myprog.prg;

name = "myprog.prg";
run ^name;
```

See also

#include, **edit**

Purpose

Saves matrices, strings, or procedures to a disk file.

Format

save ⟦*vflag*⟧ ⟦**path=***path*⟧ *x*,⟦*lpath=*⟧*y*;

Input

vflag version flag.

-v89	supported on DOS, OS/2, Windows NT
-v92	supported on Unix
-v96	supported on all platforms

See the *File I/O* chapter in Volume I of the manual 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 saved:

matrix	`.fmt`
string	`.fst`
procedure	`.fcg`
function	`.fcg`
keyword	`.fcg`

if the **path=** subcommand is used with **save**, the path string will be remembered until changed in a subsequent command. This path will be used whenever none is specified. The save path can be overridden in any particular save by specifying an explicit path and filename.

Example

```
spath = "/gauss";
save path = ^spath x,y,z;
```

Save **x**, **y**, and **z** using `/gauss` as a path. This path will be used for the next save if none is specified.

```
svp = "d:\\gauss\\bin";
save path = ^svp n, k, d:\gauss\quad1=quad;
```

This is a Windows example. **n** and **k** will be saved using `d:\gauss\bin` as the save path, **quad** will be saved in `d:\gauss` with the name `quad1.fmt`. On platforms that use the backslash as the path separator, the double backslash is required inside double quotes to get a backslash, because it is the escape character in quoted strings. It is not required when specifying literals.

```
save path=/procs;
```

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

load, **saveall**, **saved**

Purpose

To save 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**

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, 0 if fail.

Remarks

If *dataset* is null or 0, the data set name will be `temp.dat`.

if *vnames* is a null or 0, the variable names will begin with "X" and be numbered 1-K.

If *vnames* is a string or has fewer elements than *x* has columns, it will be expanded as explained under **create**.

The output data type is double precision.

Example

```
x = rndn(100,3);
dataset = "MYDATA";
let vnames = height weight age;
if not saved(x,dataset,vnames);
    errorlog "Write error";
    end;
endif;
```

Source

`saveload.src`

Globals

None

See also

loadd, **writer**, **create**

Purpose

Tests for a scalar error code.

Format

y = **scalerr(**c**);**

Input

c	N×K matrix, generally the return argument of a function or procedure call.

Output

y	scalar, which is returned as a 0 if its argument is not a scalar error code. If the argument is an error code, then **scalerr** returns the value of the error code as an integer.

Remarks

Error codes in **GAUSS** are **NaN's** (Not A Number). These are not just scalar integer values. They are special floating point encodings that the math chip recognizes as not representing a valid number. See **error**.

scalerr can be used to test for either those error codes which are predefined in **GAUSS** or an error code which the user has defined using **error**.

If c is an empty matrix, **scalerr** will return 65535.

Certain functions will either return an error code or terminate a program with an error message, depending on the trap state. The **trap** command is used to set the trap state. The error code that will be returned will appear to most commands as a missing value code, but the **scalerr** function can distinguish between missing values and error codes and will return the value of the error code.

Here are some of the functions that are affected by the trap state:

	trap 1	**trap 0**
function	error code	error message
chol	10	matrix not positive definite
invpd	20	matrix not positive definite
solpd	30	matrix not positive definite
/	40	matrix not positive definite *(second argument not square)*
	41	matrix singular *(second argument is square)*
inv	50	matrix singular

Example

```
trap 1;
cm = invpd(x);
trap 0;
if scalerr(cm);
    cm = inv(x);
endif;
```

In this example **invpd** will return a scalar error code if the matrix **x** is not positive definite. If **scalerr** returns with a nonzero value, the program will use the **inv** function, which is slower, to compute the inverse. Since the trap state has been turned off, if **inv** fails the program will terminate with a "Matrix singular" error.

See also

error, **trap**, **trapchk**

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, , a 0 for x = 1 + .i.

Example

```
clear s;
do until eof(fp);
    y = readr(fp,nr);
    y = packr(y);
    if scalmiss(y);
        continue;
    endif;
    s = s+sumc(y);
endo;
```

In this example the **packr** function will return a scalar missing if every row of its argument contains missing values, otherwise it will return a matrix that contains no missing values. **scalmiss** is used here to test for a scalar missing returned from **packr**. If that is true, then the sum step will be skipped for that iteration of the read loop because there were no rows left after the rows containing missings were packed out.

Purpose

To reduce 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-diagional 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

Globals

None

See also

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);
```

$$s = \begin{matrix} 16.11684397 & 4.89897949 & 0.00000000 \\ -0.00000000 & -1.11684397 & -0.00000000 \\ 0.00000000 & 0.00000000 & -0.00000000 \end{matrix}$$

$$z = \begin{matrix} 0.23197069 & 0.88290596 & 0.40824829 \\ 0.52532209 & 0.23952042 & -0.81649658 \\ 0.81867350 & -0.40386512 & 0.40824829 \end{matrix}$$

See also

hess

Purpose

Controls output to the screen.

Format

screen on;

screen off;

screen out;

screen;

Portability

Unix, **OS/2**, **Windows**

screen on|**off** affects the active window. Each window "remembers" its own setting, even when it is no longer the active window.

screen out affects the active window. Not supported for PQG windows.

Remarks

When this is **on**, the results of all print statements will be directed to the screen. When this is **off**, print statements will not be sent to the screen. 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 screen off will speed things up.

The **end** statement will automatically do **output off** and **screen on**.

screen out will take a snapshot of the screen and dump it to the current auxiliary output. If the output is open, the new data will be appended. If the output is closed, it will be opened for append and the screen will be written and then it will be closed again. This is handy for dumping a text screen image to a file or printer.

It also allows a complex screen image, created with **locate** statements, to be sent to a file or printer. This is not a graphics screen dump—the screen must be in text mode.

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 screen off will speed up execution. The **end** statement above will automatically perform **output off** and **screen on**.

See also

output, **end**, **new**

Purpose

Scrolls a section of the screen.

Format

scroll *v*;

Input

v	6×1 vector.

Portability

Unix, **OS/2**, **Windows**

scroll scrolls a region of the active window.

Unix only

Supported only for Text windows.

Remarks

The elements of *v* are defined as:

[1]	coordinate of upper left row.
[2]	coordinate of upper left column.
[3]	coordinate of lower right row.
[4]	coordinate of lower right column.
[5]	number of lines to scroll.
[6]	value of attribute.

This assumes the origin at (1,1) in the upper left just like the **locate** command. The screen will be scrolled the number of lines up or down (positive or negative 5^{th} element) and the value of the 6^{th} element will be used as the attribute as follows:

7	regular text
112	reverse video
0	graphics black

See the IBM technical reference manual for more information on attribute byte values.

If the number of lines (element 5) is 0, the entire window will be blanked.

Example

An example of a valid call to this function is as follows:

```
let v = 1 1 12 80 5 7;
scroll v;
```

This call would scroll a window 80 columns wide covering the upper twelve rows of the screen. The window would be scrolled up 5 lines and the new lines would be displayed in regular text mode.

See also

locate, **printdos**

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**

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.
e	N×1 vector of 1's and 0's.

Output

y	M×K matrix consisting of the rows of x for which there is a 1 in the corresponding row of e.

Remarks

The argument e will usually be generated by a logical expression using "dot" operators.

y will be a scalar missing if no rows are selected.

Example

```
y = selif(x,x[.,2] .gt 100);
```

selects all rows of **x** in which the second column is greater than 100.

```
let x[3,3] = 0 10 20
             30 40 50
             60 70 80;

e = (x[.,1] .gt 0) .and (x[.,3] .lt 100);
y = selif(x,e);
```

The resulting matrix y is:

```
30  40  50
60  70  80
```

All rows for which the element in column 1 is greater than 0 and the element in column 3 is less than 100 are placed into the matrix **y**.

Source

datatran.src

Globals

None

See also

delif, **scalmiss**

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×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**

Purpose

Returns the unique elements in one vector that are not present in a second vector.

Format

y = **setdif(***v1*,*v2*,*flag***);**

Input

v1	N×1 vector.	
v2	M×1 vector.	
flag	scalar,	if 0, do case-sensitive character comparison. if 1, do numeric comparison. if 2, do case-insensitive character comparison.

Output

y	L×1 sorted vector containing all unique values that are in *v1* and are not in *v2*, or a scalar missing.

Source

`setdif.src`

Globals

None

Example

```
let v1 = mary jane linda john;
let v2 = mary sally;
flag = 0;
y = setdif(v1,v2,flag);
```

```
       JANE
y =    JOHN
      LINDA
```

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

Globals

None

See also

makevars

Purpose

Set video mode.

Format

y = **setvmode(***mode***);**

Input

mode scalar, video mode to set.

-2	get current video mode
-1	hardware default mode
0	40 x 25 text, 16 grey
1	40 x 25 text, 16/8 color
2	80 x 25 text, 16 grey
3	80 x 25 text, 16/8 color
4	320 x 200, 4 color
5	320 x 200, 4 grey
6	640 x 200, BW
7	80 x 25 text, BW
8	720 x 348, BW Hercules
13	320 x 200, 16 color
14	640 x 200, 16 color
15	640 x 350, BW
16	640 x 350, 4 or 16 color
17	640 x 480, BW
18	640 x 480, 16 color
19	320 x 200, 256 color

Output

Returns an 8×1 vector:

[1]	number of pixels in x axis
[2]	number of pixels in y axis
[3]	number of text columns
[4]	number of text rows
[5]	number of actual colors
[6]	number of bits per pixel
[7]	number of video pages
[8]	video mode

If the mode requested is not supported by the hardware, this function will return a scalar zero.

If the argument (*mode*) is -2, the video mode will not be changed and the returned vector will reflect the current mode.

Portability

Unix, **OS/2**, **Windows**

setvmode affects the active window. This command forces the active window into a DOS emulation mode, and is supported for backwards compatibility only. Window size, resolution, font and/or colormap may be changed to accomodate the requested video mode. Not supported for window 1 and TTY windows.

Purpose

Executes an operating system shell.

Format

shell [[*s*]];

Portability

Unix

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

This function may be used in terminal mode.

OS/2, Windows

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

DOS

This is equivalent to the **DOS** command.

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).

You can use ">" instead of the word "shell".

Example

```
comstr = "ls ./src";
shell ^comstr;
```

This lists the contents of the `./src` subdirectory, then returns to **GAUSS**.

```
>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

exit, **exec**

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);
```

$$\text{If } x = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix} \text{ and } s = \begin{matrix} 1 \\ -1 \end{matrix} \text{ and } f = \begin{matrix} 99 \\ 999 \end{matrix}$$

$$\text{Then } y = \begin{matrix} 99 & 1 \\ 4 & 999 \end{matrix}$$

$$\text{If } x = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \text{ and } s = \begin{matrix} 0 \\ 1 \\ 2 \end{matrix} \text{ and } f = 0$$

$$\text{Then } y = \begin{matrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 7 \end{matrix}$$

See also

rotater

Purpose

Displays the global symbol table. The output from **lshow** is sent to the printer.

Format

show [[/*flags*]] [[*symbol*]];

lshow [[/*flags*]] [[*symbol*]];

Input

flags flags to specify the symbol type that is shown.

- **k** keywords
- **p** procedures
- **f** **fn** functions
- **m** matrices
- **s** strings
- **g** show only symbols with global references
- **l** show only symbols with all local references
- **n** no pause

symbol the name of the symbol to be shown. If the last character is an asterisk, all symbols beginning with the supplied characters will be shown.

Remarks

If there are no arguments, the entire symbol table will be displayed.

show is directed to the auxiliary output if it is open.

Here is an example listing with an explanation of the columns:

```
Memory used   Address    Name         Info  Cplx  Type       References

  32 bytes at [00081b74]  AREA          1=1        FUNCTION   local refs
  32 bytes at [00081a14]  dotfeq        1=2        PROCEDURE  global refs
1144 bytes at [0007f1b4]  indices2      4=3        PROCEDURE  local refs
 144 bytes at [0007f874]  X            3,3    C   MATRIX
 352 bytes at [0007f6ec]  _IXCAT      44,1        MATRIX
   8 bytes at [0007f6dc]  _olsrnam     7 char     STRING

32000 bytes program space, 0% used
4336375 bytes workspace, 4325655 bytes free
53 global symbols, 1500 maximum, 6 shown
```

The "memory used" column is the amount of memory used by the item.

The "address" column is the address where the item is stored (hexadecimal format). This will change as matrices and strings change in size.

The "name" column is the name of the symbol.

The "info" column depends on the type of the symbol. If the symbol is a procedure or a function, it gives the number of values that the function or procedure returns and the number of arguments that need to be passed to it when it is called. If the symbol is a matrix, then the info column gives the number of rows and columns. If the symbol is a string, then the info column gives the number of characters in the string. As follows:

Rets=Args	if procedure or function
Row,Col	if matrix
Length	if string

The "cplx" column contains a 'C' if the symbol is a complex matrix.

The "type" column specifies the symbol table type of the symbol. It can be **function**, **keyword**, **matrix**, **procedure**, or **string**.

If the symbol is a procedure, keyword or function, the "references" column will show if it makes any global references. If it makes only local references, the procedure or function can be saved to disk in an `.fcg` file with the **save** command. If the function or procedure makes any global references, it cannot be saved in an `.fcg` file.

The program space is the area of space reserved for all nonprocedure, nonfunction program code. It can be changed in size with the **new** command. The workspace is the memory used to store matrices, strings, procedures, and functions.

Example

```
show /fpg eig*;
```

This command will show all functions and procedures that have global references and begin with **eig**.

```
show /mn;
```

This command will show all matrices without pausing when the screen is full.

See also

new, **delete**

Purpose

Returns the sine of its argument.

Format

y = **sin(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix containing sine of x.

Remarks

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

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

Example

```
let x = { 0, .5, 1, 1.5 };
y = sin(x);
```

$$y = \begin{matrix} 0.00000000 \\ 0.47942554 \\ 0.84147098 \\ 0.99749499 \end{matrix}$$

See also

atan, **cos**, **sinh**, **pi**

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);
```

$$x = \begin{matrix} -1.570796 \\ -0.785398 \\ 0.000000 \\ 0.785398 \\ 1.570796 \\ 3.141593 \end{matrix}$$

$$y = \begin{matrix} -2.301299 \\ -0.868671 \\ 0.000000 \\ 0.868671 \\ 2.301299 \\ 11.548739 \end{matrix}$$

Source

trig.src

Globals

None

Purpose

Sleep for a specified number of seconds.

Format

unslept = **sleep(***secs***);**

Input

secs	scalar, number of seconds to sleep.

Output

unslept	scalar, number of seconds not slept.

Remarks

secs does not have to be an integer. If your system does not permit sleeping for a fractional number of seconds, *secs* will be rounded to the nearest integer, with a minimum value of 1.

If a program sleeps for the full number of *secs* specified, **sleep** returns 0; otherwise, if the program is awakened early (e.g., by a signal), **sleep** returns the amount of time not slept. The DOS version always sleeps the full number of seconds, so it always returns 0.

A program may sleep for longer than *secs* seconds, due to system scheduling.

Purpose

Solves a set of positive definite linear equations.

Format

x = **solpd(**b,a**);**

Input

b	N×K matrix.
A	N×N symmetric positive definite matrix.

Output

x	N×K matrix, the solutions for the system of equations, $Ax = b$.

Remarks

b can have more than one column. If so, the system of equations is solved for each column, i.e., $A * x[., i] = b[., i]$.

This function uses the Cholesky decomposition to solve the system directly. Therefore it is more efficient than using **inv(**A**)***b.

solpd does not check to see that the matrix A is symmetric. **solpd** will look only at the upper half of the matrix including the principal diagonal.

If the A matrix is not positive definite:

trap 1 return scalar error code 30.

trap 0 terminate with an error message.

One obvious use for this function is to solve for least squares coefficients. The effect of this function is thus similar to that of the **/** operator. See Section 8.2.1.

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 20,8;
A = rndn(n,n);
A = A'A;
x = rndn(n,1);
b = A*x;
x2 = solpd(b,A);
print "     X           solpd(b,A)        Difference";
print x~x2~x-x2;

     X            solpd(b,A)       Difference

-0.36334089      -0.36334089       0.00000000
 0.19683330       0.19683330       8.32667268E-017
 0.99361330       0.99361330       2.22044605E-016
-1.84167681      -1.84167681       0.00000000
-0.88455829      -0.88455829       1.11022302E-016
```

See also

scalerr, **chol**, **invpd**, **trap**

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 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);
```

$$x = \begin{matrix} 4 & 7 & 3 \\ 1 & 3 & 2 \\ 3 & 4 & 8 \end{matrix}$$

$$y = \begin{matrix} 1 & 3 & 2 \\ 3 & 4 & 8 \\ 4 & 7 & 3 \end{matrix}$$

See also

rev

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*.

Putting this file on a RAM disk can speed up the program considerably.

If the inputs are null or 0, the procedure will ask for them.

Source

`sortd.src`

Globals

None

Purpose

Sorts a matrix of numeric or character data.

Format

y = **sorthc(**x,c**);**

y = **sorthcc(**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.

sorthc assumes that the column to sort on is numeric. **sorthcc** assumes that the column to sort on contains character data.

The matrix may contain both character and numeric data, but the sort column must be all of one type. Missing values will sort as if their value is below $-\infty$.

The sort is in ascending order. This function uses the heap sort algorithm.

If you need to obtain the matrix sorted in descending order, you can use:

rev(sorthc(x,c**))**

Example

```
let x[3,3]= 4 7 3
            1 3 2
            3 4 8;
y = sorthc(x,1);
```

$$x = \begin{matrix} 4 & 7 & 3 \\ 1 & 3 & 2 \\ 3 & 4 & 8 \end{matrix}$$

$$y = \begin{matrix} 1 & 3 & 2 \\ 3 & 4 & 8 \\ 4 & 7 & 3 \end{matrix}$$

See also

sortc, **rev**

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 x contains numeric data. **sortindc** assumes x contains character data.

This function can be used to sort several matrices in the same way that some other reference matrix is sorted. To do this, create the index of the reference matrix, then use **submat** to rearrange the other matrices in the same way.

Example

```
let x = 5 4 4 3 3 2 1;
ind = sortind(x);
y = x[ind];
```

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.

Remarks

The function works recursively and the number of sort columns is limited by the available workspace.

Source

`sortmc.src`

Globals

sortmc

Purpose

Returns the number of columns in a sparse matrix.

Format

c = **sparseCols(**x**);**

Input

x	M×N sparse matrix.

Output

c	scalar, number of columns.

Source

`sparse.src`

Purpose

Returns a sparse identity matrix.

Format

y = **sparseEye(**n**);**

Input

n	scalar, order of identity matrix.

Output

y	N×N sparse identity matrix.

Example

```
y = sparseEye(3);
d = denseSubmat(y,0,0);
```

$$d = \begin{matrix} 1.0000000 & 0.0000000 & 0.0000000 \\ 0.0000000 & 1.0000000 & 0.0000000 \\ 0.0000000 & 0.0000000 & 1.0000000 \end{matrix}$$

Purpose

Dense to sparse matrix.

Format

y = **sparseFD(**x,eps**);**

Input

x	M×N dense matrix.
eps	scalar, elements of x less than eps will be treated as zero.

Output

y	M×N sparse matrix.

Remarks

A dense matrix is just a normal format matrix.

Source

`sparse.src`

Purpose

Packed matrix to sparse matrix.

Format

y = **sparseFP(**x,r,c**);**

Input

x	M×3 packed matrix, see remarks for format.
r	scalar, rows of output matrix.
c	scalar, columns of output matrix.

Output

y	R×C sparse matrix.

Remarks

x contains the nonzero elements of the sparse matrix. The first column of x contains the element value, the second column the row number, and the third column the column number.

Source

`sparse.src`

Purpose

Horizontally concatenates two sparse matrices.

Format

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

Input

y	M×N sparse matrix, left hand matrix.
x	M×L sparse matrix, right hand matrix.

Output

z	M×(N+L) sparse matrix.

Source

`sparse.src`

Purpose

Returns the number of nonzero elements in a sparse matrix.

Format

r = **sparseNZE(**x**);**

Input

x	M×N sparse matrix.

Output

r	scalar, number of nonzero elements in x.

Source

`sparse.src`

Purpose

generates sparse matrix of ones and zeros

Format

y = **sparseOnes(**x**,**r**,**c**);**

Input

x	M×2 matrix, first column contains row numbers of the ones, and the second column contains column numbers
r	scalar, rows of full matrix
c	scalar, columns of full matrix

Output

y	sparse matrix

Source

`sparse.src`

Purpose

Returns the number of rows in a sparse matrix.

Format

r = **sparseRows(**x**);**

Input

x	M×N sparse matrix.

Output

r	scalar, number of rows.

Source

`sparse.src`

Purpose

Resets sparse library global matrices to default values.

Format

sparseSet;

Globals

_sparse_ARnorm, **_sparse_Acond**, **_sparse_Anorm**, **_sparse_Atol**, **_sparse_Btol**, **_sparse_CondLimit**, **_sparse_Damping**, **_sparse_NumIters**, **_sparse_RetCode**, **_sparse_Rnorm**, **_sparse_Xnorm**

Source

`sparse.src`

Purpose

Solves $Ax = B$ for x when A is a sparse matrix.

Format

x = **sparseSolve(**A,B**);**

Input

A	M×N sparse matrix.
B	N×1 vector.

Output

x	N×1 vector, solution of $Ax = B$.

Globals

_sparse_Damping scalar, if nonzero, damping coefficient for damped least squares solve, i.e.,

$$\begin{bmatrix} A \\ dI \end{bmatrix} x = \begin{bmatrix} B \\ 0 \end{bmatrix}$$

is solved for X where d = **_sparse_Damping**, I is a conformable identity matrix, and 0 a conformable matrix of zeros.

_sparse_Atol scalar, an estimate of the relative error in A. If zero, **_sparse_Atol** is assumed to be machine precision. Default = 0.

_sparse_Btol an estimate of the relative error in B. If zero, **_sparse_Btol** is assumed to be machine precision. Default = 0.

_sparse_CondLimit upper limit on condition of A. Iterations will be terminated if a computed estimate of the condition of A exceeds **_sparse_CondLimit**. If zero, set to 1 / machine precision.

_sparse_NumIters maximum number of iterations.

Output globals:

_sparse_RetCode scalar, termination condition.

0	x is the exact solution, no iterations performed.

1	solution is nearly exact with accuracy on the order of **_sparse_Atol** and**_sparse_Btol**.
2	solution is not exact and a least squares solution has been found with accuracy on the order of **_sparse_Atol**.
3	the estimate of the condition of A has exceeded **_sparse_CondLimit**. The system appears to be ill-conditioned.
4	solution is nearly exact with reasonable accuracy.
5	solution is not exact and a least squares solution has been found with reasonable accuracy.
6	iterations halted due to poor condition given machine precision.
7	**_sparse_NumIters** exceeded.

_sparse_Anorm scalar, estimate of Frobenius norm of

$$\begin{bmatrix} A \\ dI \end{bmatrix}$$

_sparse_Acond estimate of condition of A.

_sparse_Rnorm estimate of norm of

$$\begin{bmatrix} A \\ dI \end{bmatrix} x - \begin{bmatrix} B \\ 0 \end{bmatrix}$$

_sparse_ARnorm estimate of norm of

$$\begin{bmatrix} A \\ dI \end{bmatrix}' \begin{bmatrix} A \\ dI \end{bmatrix}$$

_sparse_XAnorm estimate of norm of x.

Source

`sparse.src`

Purpose

Returns (sparse) submatrix of sparse matrix.

Format

c = **sparseSubmat(**x,r,c**);**

Input

x	M×N sparse matrix.
r	K×1 vector, row indices.
c	L×1 vector, column indices.

Output

e	K×L sparse matrix.

Remarks

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

Source

`sparse.src`

Purpose

Multiplies sparse matrix by dense matrix.

Format

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

Input

x	M×N sparse matrix.
y	N×L dense matrix.

Output

z	M×L dense matrix, the result of $x \times y$.

Source

`sparse.src`

Purpose

Multiplies sparse matrix transposed by dense matrix.

Format

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

Input

x	N×M sparse matrix.
y	N×L dense matrix.

Output

z	M×L dense matrix, the result of $x'y$.

Source

`sparse.src`

Purpose

Vertically concatenates two sparse matrices.

Format

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

Input

y	M×N sparse matrix, top matrix.
x	L×N sparse matrix, bottom matrix.

Output

z	(M+L)×N sparse matrix.

Source

`sparse.src`

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`

Purpose

solve 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

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. The 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

Globals

_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 CO 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 proc for this is:

```
proc eqproc(p);
   retp(p[1]*[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 CO 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 proc 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 = { -1e256 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 CO that a gradient procedure exists as well 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 OPTMUM 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.

Remarks

Pressing C on the keyboard will terminate iterations, and pressing P will toggle iteration output.

sqpSolve is recursive, that is, it can call a version of 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

Purpose

Computes the square root of every element in a matrix.

Format

y = **sqrt(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix, the square roots of each element of x.

Remarks

If x is negative, complex results are returned.

You can turn the generation of complex numbers for negative inputs on or off in the **GAUSS** configuration file, and with the **sysstate** function, case 8. If you turn it off, **sqrt** will generate an error for negative inputs.

If x is already complex, the complex number state doesn't matter; **sqrt** will compute a complex result.

Example

```
let x[2,2] = 1 2 3 4;
y = sqrt(x);
```

$$x = \begin{matrix} 1.00000000 & 2.00000000 \\ 3.00000000 & 4.00000000 \end{matrix}$$

$$y = \begin{matrix} 1.00000000 & 1.41421356 \\ 1.73205081 & 2.00000000 \end{matrix}$$

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

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, contains the contents of s.

Remarks

stocv in effect 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);
```

$$v = \begin{matrix} \text{“}Now\ is\ t\text{”} \\ \text{“}he\ time\ \text{”} \\ \text{“}for\ all\ \text{”} \\ \text{“}good\ men\text{”} \end{matrix}$$

See also

cvtos, **vget**, **vlist**, **vput**, **vread**

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**

Purpose

Stops a program and returns to COMMAND mode. Does not close files.

Format

stop;

Portability

Unix, **OS/2**, **Windows**

stop does not close any windows. If you want to close a program's windows when it is done, use **end**.

Remarks

This command has the same effect as **end**, except it does not close files or the auxiliary output.

It is not necessary to put a **stop** or an **end** statement at the end of a program. If neither is found, an implicit **stop** is executed.

See also

end, **new**, **system**

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 = "Whatchmacallit";
x = "call";
y = strindx(z,x,1);
```

$y = 9$

This function is used with **strsect** for extracting substrings.

See also

strrindx, **strlen**, **strsect**, **strput**

Purpose

Returns the length of a string.

Format

y = **strlen(**x**);**

Input

x	string or N×K matrix of character data.

Output

y	scalar containing the exact length of the string x or N×K matrix of the lengths of the elements in the matrix x.

Remarks

The null character (ASCII 0) is a legal character within strings (if they are of type **STRING**) 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**

Purpose

To lay 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;

maximum
```

Source

strput.src

Globals

None

Purpose

Finds the index of one string within another string. Searches from the end of the string to the beginning.

Format

y = **strrindx(***where*,*what*,*start***);**

Input

where	string or scalar, the data to be searched.
what	string or scalar, the substring to be searched for in *where*.
start	scalar, the starting point of the search in *where* for an occurrence of *what*. *where* will be searched from this point backward for *what*.

Output

y	scalar containing the index of the last occurrence of *what*, within *where*, which is less than or equal to *start*. If no occurrence is found, it will be 0.

Remarks

A negative value for *start* causes the search to begin at the end of the string. An example of the use of **strrindx** is extracting a file name from a complete path specification:

```
path = "/gauss/src/ols.src";
ps = "/";
pos = strrindx(path,ps,-1);
if pos;
    name = strsect(path,pos+1,strlen(path));
else;
    name = "";
endif;
```

$pos = 11$

$name = ols.src$

strrindx can be used with **strsect** for extracting substrings.

See also

strindx, **strlen**, **strsect**, **strput**

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**

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);
```

$$x = \begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$$

$$y = \begin{matrix} 2 & 4 \\ 10 & 12 \end{matrix}$$

$$z = \begin{matrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{matrix}$$

See also

diag, **vec**, **reshape**

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:

$$\begin{array}{rcccccl} & & x & \leq & v[1] & \rightarrow & s[1] \\ v[1] & < & x & \leq & v[2] & \rightarrow & s[2] \\ & & & & \cdots & & \\ v[p-1] & < & x & \leq & v[p] & \rightarrow & s[p] \\ & & x & > & v[p] & \rightarrow & \text{the original value of } x \end{array}$$

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);
```

$$y = \begin{matrix} 10 \\ 10 \\ 10 \\ 10 \\ 5 \\ 0 \\ 0 \\ 0 \\ 9 \\ 10 \end{matrix}$$

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:

$$x = \begin{array}{ccc} R & 55 & 30 \\ N & 57 & 18 \\ Y & 24 & 3 \\ N & 63 & 38 \\ R & 55 & 32 \\ N & 37 & 11 \end{array}$$

Source

`datatran.src`

Globals

None

See also

code, **recode**

Purpose

Computes the sum of each column of a matrix.

Format

y = **sumc(**x**);**

Input

x	N×K matrix.

Output

y	K×1 column vector containing the sums of each column of x.

Remarks

To find the sums of the elements in each row of a matrix, transpose before applying **sumc**. If x is complex, use the bookkeeping transpose (.′).

To find the sums of all of the elements in a matrix, use the **vecr** function before applying **sumc**.

Example

```
x = round(rndu(5,3)*5);
y = sumc(x);
```

$$x = \begin{matrix} 2 & 4 & 3 \\ 2 & 1 & 2 \\ 5 & 1 & 3 \\ 5 & 1 & 1 \\ 3 & 3 & 4 \end{matrix}$$

$$y = \begin{matrix} 17 \\ 10 \\ 13 \end{matrix}$$

See also

cumsumc, **meanc**, **stdc**

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.
_svderr	global 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);
```

$$y = \begin{matrix} 16.521787 \\ 3.3212254 \end{matrix}$$

Source

svd.src

Globals

_svderr

See also

svd1, **svd2**, **svds**

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.
_svderr	global scalar, if all of the singular values are correct, **_svderr** is 0. If not all of the singular values can be computed, **_svderr** is set and the diagonal elements of s with indices greater than **_svderr** are correct.

Remarks

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

trap 0	set **_svderr** and terminate with message
trap 1	set **_svderr** and continue execution

Example

```
x = rndu(3,3);
{ u, s, v } = svd1(x);
```

$$x = \begin{matrix} 0.97847012 & 0.20538614 & 0.59906497 \\ 0.85474208 & 0.79673540 & 0.22482095 \\ 0.33340653 & 0.74443792 & 0.75698778 \end{matrix}$$

$$u = \begin{matrix} -0.57955818 & 0.65204491 & 0.48882486 \\ -0.61005618 & 0.05056673 & -0.79074298 \\ -0.54031821 & -0.75649219 & 0.36847767 \end{matrix}$$

$$s = \begin{matrix} 1.84994646 & 0.00000000 & 0.00000000 \\ 0.00000000 & 0.60370542 & 0.00000000 \\ 0.00000000 & 0.00000000 & 0.47539239 \end{matrix}$$

$$v = \begin{matrix} -0.68578561 & 0.71062560 & -0.15719208 \\ -0.54451302 & -0.64427479 & -0.53704336 \\ -0.48291165 & -0.28270348 & 0.82877927 \end{matrix}$$

Source

`svd.src`

Globals

_svderr

See also

svd, **svd2**, **svdusv**

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.
_svderr	global scalar, if all of the singular values are correct, **_svderr** is 0. If not all of the singular values can be computed, **_svderr** is set and the diagonal elements of s with indices greater than **_svderr** are correct.

Remarks

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

trap 0	set **_svderr** and terminate with message
trap 1	set **_svderr** and continue execution

Source

svd.src

Globals

_svderr

See also

svd, **svd1**, **svdcusv**

Purpose

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

Format

{ u,s,v } = **svdcusv(**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, u is N×P containing only the P left singular vectors of x.
s	N×P or P×P diagonal matrix, the singular values of x arranged in descending order on the principal diagonal. If N > P, s is P×P.
v	P×P matrix, the right singular vectors of x.

Remarks

If not all 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 the singular values have been computed.

See also

svd2, **svds**, **svdusv**

Purpose

Computes the singular values of a matrix.

Format

s = **svds(***x***);**

Input

x	N×P matrix whose singular values are to be computed.

Output

s	min(N,P)×1 vector, the singular values of *x* arranged in descending order.

Remarks

If not all the singular values can be computed, *s[1]* is set to a scalar error code. Use **scalerr** to convert this to an integer. The elements of *s* with indices greater than **scalerr(***s[1]***)** are correct. If **scalerr(***s[1]***)** returns a 0, all the singular values have been computed.

See also

svd, **svdcusv**, **svdusv**

Purpose

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

Format

{ *u*,*s*,*v* } = **svdusv**(*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, the singular values of *x* arranged in descending order on the principal diagonal.
v	P×P matrix, the right singular vectors of *x*.

Remarks

If not all 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 the singular values have been computed.

See also

svd1, **svdcusv**, **svds**

Purpose

To get or set general system parameters.

Format

{ *rets...* } = **sysstate(***case*,*y***);**

Case 1: Version Information

Purpose

Returns the current **GAUSS** version information in a 7-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

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

Case 2-7: GAUSS System Paths

Purpose

Get or set **GAUSS** system path.

Format

oldpath = **sysstate(***case*,*path*);

Input

case scalar 2-7, path to set.

- **2** `exe` file location.
- **3** **loadexe** path.
- **4** **save** path.
- **5** **load**, **loadm** path.
- **6** **loadf**, **loadp** path.
- **7** **loads** path.

path scalar 0 to get path, or string containing the new path.

Output

oldpath string, original path.

Remarks

If *path* is of type matrix, the path will be returned but not modified.

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

Get and set trailing character for the imaginary part of a complex number.

Format

oldtrail = **sysstate(***9*,*trail***);**

Input

trail	scalar 0 to get character, or string containing the new trailing character.

Output

oldtrail	string, the original trailing character.

Remarks

The default character is '**i**'.

Case 10: Printer Width

Purpose

Get and set **lprint** width.

Format

oldwidth = **sysstate(***10*,*width***);**

Input

width	scalar, new printer width.

Output

oldwidth	scalar, the current original width.

Remarks

If *width* is 0, the printer width will not be changed.

This may also be set with the **lpwidth** command.

See also

lpwidth

Case 11: Auxiliary Output Width

Purpose

Get and set the auxiliary output width.

Format

oldwidth = **sysstate(***11***,***width***);**

Input

width	scalar, new output width.

Output

oldwidth	scalar, the original output width.

Remarks

If *width* is 0 then the output width will not be changed.

This may also be set with the **outwidth** command.

See also

outwidth

Case 12: Precision

Purpose

Get and set precision for positive definite matrix routines.

Format

oldprec = **sysstate(***12***,***prec***);**

Input

prec	scalar, 64 or 80.

Output

oldprec	scalar, the original value.

Remarks

The precision will be changed if *prec* is either 64 or 80. Any other number will leave the precision unchanged.

See also

prcsn

Case 13: LU Tolerance

Purpose

Get and set singularity tolerance for LU decomposition.

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.

See also

croutp, **inv**, **/**, Appendix E

Case 14: Cholesky Tolerance

Purpose

Get and set singularity tolerance for Cholesky decomposition.

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.

See also

chol, **invpd**, **solpd**, **/**, Appendix E

Case 15: Screen State

Purpose

Get and set screen state as controlled by **screen** command.

Format

oldstate = **sysstate(***15*,*state*);

Input

state	scalar, new screen state.

Output

oldstate	scalar, the original screen state.

Remarks

If *state* = 1, screen output is turned on. If *state* = 0, screen output is turned off. If *state* = -1, the state is returned unchanged.

See also

screen

Case 16: Automatic print Mode

Purpose

Get and set automatic **print** mode.

Format

oldmode = **sysstate(***16*,*mode*);

Input

mode	scalar, mode.

Output

oldmode	scalar, original mode.

Remarks

If *mode* = 1, automatic **print** mode is turned on. If *mode* = 0, it is turned off. If *mode* = -1, the mode is returned unchanged.

See also

print on/off

Case 17: Automatic lprint Mode

Purpose

Get and set automatic **lprint** mode.

Format

oldmode = **sysstate(***17*,*mode*);

Input

mode	scalar, mode.

Output

oldmode	scalar, original mode.

Remarks

If *mode* = 1, automatic **lprint** mode is turned on. If *mode* = 0, it is turned off. If *mode* = -1, the mode is returned unchanged.

See also

lprint on/off

Case 18: Auxiliary Output

Purpose

Get auxiliary output parameters.

Format

{ *state*,*name* } = **sysstate(***18*,*dummy*);

Input

dummy	scalar, a dummy argument.

Output

state	scalar, auxiliary output state, 1 - on, 0 - off.
name	string, auxiliary output filename.

See also

output

Case 19: Get/Set Format

Purpose

Get and set format parameters.

Format

oldfmt = **sysstate(***19*,*fmt*);

Input

fmt scalar or 10×1 column vector containing the new format parameters. See description below.

Output

oldfmt 10×1 vector containing the current format parameters:

[1] format type.
[2] justification.
[3] sign.
[4] leading zero.
[5] trailing character.
[6] row delimiter.
[7] carriage line feed position.
[8] automatic line feed for row vectors.
[9] field.
[10] precision.

Remarks

If *fmt* is scalar 0, then the format parameters will be left unchanged.

See also

format

Case 21: Imaginary Tolerance

Purpose

Get and set the imaginary tolerance.

Format

oldtol = **sysstate(***21***,***tol***);**

Input

tol	scalar, the new tolerance.

Output

oldtol	scalar, the original tolerance.

Remarks

The imaginary tolerance is used to test whether the imaginary part of a complex matrix can be treated as zero or not. Functions that are not defined for complex matrices check the imaginary part to see if it can be ignored. The default tolerance is 2.23e−16, or machine epsilon.

If $tol < 0$, the current tolerance is returned.

See also

hasimag

Case 22: Source Path

Purpose

Get and set the path the compiler will search for source files.

Format

oldpath = **sysstate(***22***,***path***);**

Input

path scalar 0 to get path, or string containing the new path.

Output

oldpath string, original path.

Remarks

If *path* is a matrix, the current source path is returned.

This resets the **src_path** configuration variable. **src_path** is initially defined in the **GAUSS** configuration file, `gauss.cfg`.

path can list a sequence of directories, separated by semicolons.

Resetting **src_path** affects the path used for subsequent **run** and **compile** statements.

Case 24: Dynamic Library Directory

Purpose

Get and set the path for the default dynamic library directory.

Format

oldpath = **sysstate(***24***,***path***);**

Input

path scalar 0 to get path, or string containing the new path.

Output

oldpath string, original path.

Remarks

If *path* is a matrix, the current path is returned.

path should list a single directory, not a sequence of directories.

Changing the dynamic library path does not affect the state of any DLL's currently linked to **GAUSS**. Rather, it determines the directory that will be searched the next time **dlibrary** is called.

Unix

Changing the path has no effect on **GAUSS**'s default DLL, `libgauss.so`. `libgauss.so` must always be located in the GAUSSHOME directory.

OS/2, Windows

Changing the path has no effect on **GAUSS**'s default DLL, `gauss.dll`. `gauss.dll` must always be located in the same directory as the **GAUSS** executable, `gauss.exe`.

See also

dlibrary, **dllcall**

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 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.

Portability

Unix, **OS/2**, **Windows**

Remarks

Internally, **date** acquires the number of years since 1900. **sysstate 30** specifies whether **date** should add the base year to that value or not. If *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.

sysstate 30 has no effect in **GAUSS** for DOS. It always returns a 4-digit year.

Purpose

To quit **GAUSS** and return 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 normal interactive way to exit from **GAUSS** to DOS is to hit the **ESCAPE** key. The **system** command can be used in a program and allows you to return an exit code.

The **system** command is used at the end of batch programs to leave **GAUSS** completely and return to DOS. The exit code can be recovered by the **if errorlevel** subcommand of the DOS batch processor. See your DOS manual. The exit code can also be recovered by the **exec** function if you have **exec**'ed a second copy of **GAUSS** from a **GAUSS** program.

The **system** command exits **GAUSS** completely and returns back to the operating system. The **DOS** command allows access to a second level DOS shell from within **GAUSS**.

See also

dos, **exec**, **exit**

Purpose

Tab 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. You can tab backwards as well as forwards in Text windows, but only forward in TTY windows, including window 1. 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);
```

Purpose

Returns the tangent of its argument.

Format

y = **tan(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix.

Remarks

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

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

Example

```
let x = 0 .5 1 1.5;
y = tan(x);
```

$$y = \begin{matrix} 0.00000000 \\ 0.54630249 \\ 1.55740772 \\ 14.10141995 \end{matrix}$$

See also

atan, **pi**

Purpose

Computes the hyperbolic tangent.

Format

y = **tanh(**x**);**

Input

x	N×K matrix.

Output

y	N×K matrix containing the hyperbolic tangents of the elements of x.

Example

```
let x = -0.5  -0.25  0  0.25  0.5  1;
x = x * pi;
y = tanh(x);
```

$$x = \begin{matrix} -1.570796 \\ -0.785398 \\ 0.000000 \\ 0.785398 \\ 1.570796 \\ 3.141593 \end{matrix}$$

$$y = \begin{matrix} -0.917152 \\ -0.655794 \\ 0.000000 \\ 0.655794 \\ 0.917152 \\ 0.996272 \end{matrix}$$

Source

trig.src

Globals

None

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**

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;
```

7 : 31 : 46

Source

`time.src`

Globals

None

See also

date, **datestr**, **datestring**, **datestrymd**, **ethsec**, **etstr**, **time**

Purpose

Returns the number of seconds since January 1, 1970 Greenwhich Mean Time.

Format

tc = **timeutc;**

Output

tc	Scalar, number of seconds since January 1, 1970 Greenwhich Mean Time.

Example

```
tc = timeutc;
utv = utctodtv(tc);
```

$tc =$ 8.7374957e + 08

$utv =$ 1997. 9.000 8.000 12.00 12.00 50.00 1.000 250.0

See also

dtvnormal, **utctodtv**

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`

Globals

None

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);
```

$$x = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix}$$

$$y = \begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 2 & 3 \\ 4 & 3 & 2 & 1 & 2 \\ 5 & 4 & 3 & 2 & 1 \end{matrix}$$

Source

toeplitz.src

Globals

None

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

Globals

None

See also

parse

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

Globals

None

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. See the *Debugger* or *Error Handling and Debugging* chapter in your supplement.

The argument is converted to a binary integer with the following meanings:

bit	decimal	meaning
ones	1	trace calls/returns
twos	2	trace line numbers
fours	4	verbose trace
eights	8	output to screen
sixteens	16	output to print
thirty-twos	32	output to auxiliary output
sixty-fours	64	output to error log

You must set one or more of the output bits to get any output from **trace**. If you set **trace** to 4, you'll be doing a verbose trace of your program, but the output won't be displayed anywhere.

The trace output as a program executes will be as follows:

(+GRAD)	calling function or procedure **GRAD**
(-GRAD)	returning from **GRAD**
[47]	executing line 47

Note that the line number trace will only produce output if the program was compiled with line number records.

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

trace 4+8;    verbose trace to screen
```

See also

#lineson

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

```
oldval = trapchk(1);
trap 1,1;
y = inv(x);
trap oldval,1;
if scalerr(y);
    goto errout;
endif;
```

In this example the result of **inv** is trapped in case **x** is singular. The trap state is reset to the original value after the call to **inv**.

See also

scalerr, **trapchk**, **error**

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.

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);**

value of bit 8 in trap flag	0	1	value of bit 0 in trap flag
0	0	1	
1	256	257	

GAUSS functions that test the trap flag currently test only bits 0 and 1.

See also

scalerr, **trap**, **error**

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);
```

$$x = \begin{matrix} 0.76042751 & 0.33841579 & 0.01844780 \\ 0.05334503 & 0.38939785 & 0.65029973 \\ 0.93077511 & 0.06961078 & 0.04207563 \\ 0.53640701 & 0.06640062 & 0.07222560 \\ 0.14084669 & 0.06033813 & 0.69449247 \end{matrix}$$

$$y = \begin{matrix} 0.93077511 & 0.06961078 & 0.04207563 \\ 0.53640701 & 0.06640062 & 0.07222560 \end{matrix}$$

See also

submat, **rotater**, **shiftr**

Purpose

Converts numbers to integers by truncating the fractional portion.

Format

y = **trunc**(x);

Input

x	N×K matrix.

Output

y	N×K matrix containing the truncated elements of x.

Example

```
x = 100*rndn(2,2);
```

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

```
y = trunc(x);
```

$$y = \begin{matrix} 77.00 & -14.00 \\ 4.00 & -158.00 \end{matrix}$$

See also

ceil, **floor**, **round**

Purpose

Returns the symbol table type of the argument.

Format

t = **type(**x**);**

Input

x	matrix or string, can be an expression.

Output

t	scalar, 6 if matrix, 13 if string.

Remarks

type returns the type of a single symbol. The related function **typecv** will take a character vector of symbol names and return a vector of either their types or the missing value code for any that are undefined. **type** works for matrices and strings; **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

```
if type(k) == 6;
   k = "" $+ k;  /* force matrix to string */
endif;
```

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.

Format

y = **typecv**(x);

Input

x	string or N×1 character vector 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:

6	Matrix (Numeric, Character, or Mixed)
8	Procedure (**proc**)
9	Function (**fn**)
5	Keyword (**keyword**)
13	String

It will return the **GAUSS** missing value code if the symbol is not found, so **typecv** may be used to determine if a symbol is defined or not.

Example

```
xvar = sqrt(5);
yvar = "Montana";
fn area(r) = pi*r*r;
let names = xvar yvar area;
y = typecv(names);
```

$$names = \begin{matrix} \text{XVAR} \\ \text{YVAR} \\ \text{AREA} \end{matrix}$$

$$y = \begin{matrix} 6 \\ 13 \\ 9 \end{matrix}$$

See also

type, **typef**, **varput**, **varget**

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 for the **create** statement.

See also

colsf, **rowsf**

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);
```

$$x = \begin{matrix} \text{JANE} \\ \text{JOHN} \\ \text{LINDA} \\ \text{MARY} \\ \text{SALLY} \end{matrix}$$

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];
```

$$ind = \begin{matrix} 7 \\ 6 \\ 5 \\ 2 \\ 1 \end{matrix}$$

$$y = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix}$$

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);
```

$$y = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix}$$

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 the matrix. 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

Globals

None

See also

lowmat, **lowmat1**, **diag**, **diagrv**, **crout**

Purpose

Converts a string or matrix of character data to uppercase.

Format

y = **upper(**x**);**

Input

x	string or N×K matrix containing the character data to be converted to uppercase.

Output

y	string or N×K matrix containing the uppercase equivalent of data in x.

Remarks

If x is a numeric matrix, y will contain garbage. No error message will be generated since **GAUSS** does not distinguish between numeric and character data in matrices.

Example

```
x = "uppercase";
y = upper(x);
```

y = "UPPERCASE"

See also

lower

Purpose

To load a compiled file at the beginning of the compilation of a source program.

Format

use *fname*;

Input

fname	literal or ^string, the name of a compiled file created using the **compile** or the **saveall** command.

Remarks

The **use** command can be used ONCE at the TOP of a program to load in a compiled file which the rest of the program will be added to. In other words, if `xy.e` had the following lines:

```
library pgraph;
external proc xy;
x = seqa(0.1,0.1,100);
```

It could be compiled to `xy.gcg`. Then the following program could be run:

```
use xy;
xy(x,sin(x));
```

Which would be equivalent to:

```
new;
x = seqa(0.1,0.1,100);
xy(x,sin(x));
```

The **use** command can be used at the top of files that are to be compiled with the **compile** command. This can greatly shorten compile time for a set of closely related programs. For example:

```
library pgraph;
external proc xy,logx,logy,loglog,hist;
saveall pgraph;
```

This would create a file called `pgraph.gcg` containing all the procedures, strings and matrices needed to run Publication Graphics 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 COMMAND mode, to instantly load all the procedures for the Publication Graphics.

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 is supported from command level and can be used in a program stored in a disk file. **use** can appear only ONCE at the TOP of a program.

See also

compile, **run**, **saveall**

Purpose

Converts a number of seconds, since or before January 1, 1970 Greenwhich Mean Time, to a 1×8 date and time vector.

Format

dtv = **utctodtv(***utc***);**

Input

utc	Scalar, number of seconds since, or before January 1, 1970 Greenwhich Mean Time.

Output

dtv 1×8 date and time vector where:

Year	Year, four digit integer.
Month	1-12, Month in Year.
Day	1-31, Day of month.
Hour	0-23, Hours since midnight.
Min	0-59, Minutes.
Sec	0-59, Seconds.
DoW	0-6, Day of week, 0=Sunday.
DiY	0-365, Days since Jan 1 of current year.

Example

```
tc = timeutc;
utv = utctodtv(tc);
```

$tc = \ 8.7374957e+08$

$utv = \ 1997. \quad 9.000 \quad 8.000 \quad 12.00 \quad 12.00 \quad 50.00 \quad 1.000 \quad 250.0$

See also

dtvnormal, **timeutc**

Purpose

Computes the solution of U x = 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.

Remarks

utrisol applies a back solve to U x = b to solve for x. If b has more than one column, each column is solved for separately, i.e., **utrisol** applies a back solve to U x[.,i] = b[.,i].

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 the keyboard. When **key** returns a nonzero value, meaning a key has been pressed, the ASCII value it returns is tested to see if it is an uppercase or lowercase 'Y'. If it is, the program will jump to the label **doit**, otherwise the program will end.

See also

chrs, **ftos**, **stof**

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 matrix or string whose name is in s.

Remarks

This function searches the global symbol table for the symbol whose name is in s and returns the contents of the variable if it exists. If the symbol does not exist, the function will terminate with an "Undefined symbol" error. 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");
```

$$dog = \begin{matrix} -0.83429985 & 0.34782433 \\ 0.91032546 & 1.75446391 \end{matrix}$$

$$y = \begin{matrix} -0.83429985 & 0.34782433 \\ 0.91032546 & 1.75446391 \end{matrix}$$

See also

typecv, **varput**

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 matrix or string 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.

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

Purpose

Allows a matrix or string to be assigned to a global symbol whose name is given as a string argument.

Format

y = **varput**(x,n);

Input

x	N×K matrix or string 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;
```

$$source = \begin{matrix} -0.93519984 & 0.40642598 \\ -0.36867581 & 2.57623519 \end{matrix}$$

$$target = \begin{matrix} -0.93519984 & 0.40642598 \\ -0.36867581 & 2.57623519 \end{matrix}$$

See also

varget, **typecv**

Purpose

Allows a matrix or string to be assigned to a local symbol whose name is given as a string argument.

Format

y = **varputl(**x,n**);**

Input

x	N×K matrix or string 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;


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

Purpose

Returns a vector of ones and zeros that indicate whether variables in a data set are character or numeric.

Format

y = **vartype(***names***);**

Input

names	N×1 character vector of variable names retrieved from a data set header file with the **getname** function.

Output

y	N×1 vector of ones and zeros, 1 if variable is numeric, 0 if character.

Remarks

If a variable name in *names* is lowercase, a 0 will be returned in the corresponding element of the returned vector.

This corresponds to the way names are encoded by **atog** when character data are explicitly designated.

Example

```
names = getname("freq");
y = vartype(names);
print $names;
print y;
```

```
    AGE
    PAY
    sex
    WT


  1.0000000
  1.0000000
  0.0000000
  1.0000000
```

Source

`vartype.src`

Globals

None

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.

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`

Globals

None

See also

momentd

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)x1 vector, the columns of *x* appended to each other.
yr	(N*K)x1 vector, the rows of *x* appended to each other and the result transposed.

Remarks

vecr is much faster.

Example

```
x = { 1 2,
      3 4 };
yc = vec(x);
yr = vecr(x);
```

$$x = \begin{matrix} 1.000000 & 2.000000 \\ 3.000000 & 4.000000 \end{matrix}$$

$$yc = \begin{matrix} 1.000000 \\ 3.000000 \\ 2.000000 \\ 4.000000 \end{matrix}$$

$$yr = \begin{matrix} 1.000000 \\ 2.000000 \\ 3.000000 \\ 4.000000 \end{matrix}$$

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);
```

$$x = \begin{matrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{matrix}$$

$$v = \begin{matrix} 11 \\ 21 \\ 22 \\ 31 \\ 32 \\ 33 \end{matrix}$$

$$sx = \begin{matrix} 11 & 21 & 31 \\ 21 & 22 & 32 \\ 31 & 32 & 33 \end{matrix}$$

See also

xpnd

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

vpack.src

Globals

None

See also

vlist, **vput**, **vread**

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`

Globals

None

See also

vget, **vput**, **vread**

Purpose

Return 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**

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`

Globals

None

See also

vget, **vlist**, **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`

Globals

none

See also

vget, **vlist**, **vput**

Purpose

Return 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**

Purpose

Waits until any key is pressed.

Format

wait;

waitc;

Portability

Unix, **OS/2**, **Windows**

wait waits for a keystroke in the active window. If you are working in terminal mode, **wait** doesn't "see" any keystrokes until **Enter** is pressed.

waitc waits for a keystroke in the active window. If you are working in terminal mode, **waitc** doesn't "see" any keystrokes until **Enter** is pressed.

Remarks

waitc clears any pending keystrokes before waiting until another key is pressed.

Source

`wait.src, waitc.src`

Globals

None

See also

pause

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 to nearest as it is written out.

If the data contain character elements, the file must be double precision or the character information will be lost.

If the file being written to is the 2-byte integer data type, then missing values will be written out as -32768. These will not automatically be converted to missings on input. They can be converted with the **miss** function:

x = **miss(***x***,-32768);**

Trying to write complex data to a data set that was originally created to store real data will cause a program to abort with an error message. See the **create** command 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**

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);
```

$$x = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{matrix}$$

$$y = \begin{matrix} 1 & 2 & 4 & 7 \\ 2 & 3 & 5 & 8 \\ 4 & 5 & 6 & 9 \\ 7 & 8 & 9 & 10 \end{matrix}$$

See also

vech

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×C matrix of zeros.

Remarks

This is faster than **ones**.

Noninteger arguments will be truncated to an integer.

Example

```
y = zeros(3,2);
```

$$y = \begin{matrix} 0.000000 & 0.000000 \\ 0.000000 & 0.000000 \\ 0.000000 & 0.000000 \end{matrix}$$

See also

ones, **eye**

GAUSS for UNIX

oldfonth = **font(***fonthandle***);**
fonthandle = **FontLoad(***fontspec***);**
ret = **FontUnload(***fonthandle***);**
FontUnloadAll;
WinClear(*whandle***);**
WinClearArea(*whandle*,*row*,*col*,*nrows*,*ncols***);**
WinClearTTYLog(*ttywhandle***);**
ret = **WinClose(***whandle***);**
WinCloseAll;
whandle = **WinGetActive;**
wattr = **WinGetAttributes(***whandle***);**
rgb = **WinGetColorCells(***whandle*,*cellidx***);**
{ *r*,*c* } = **WinGetCursor(***whandle***);**
WinMove(*whandle*,*x*,*y***);**
whandle = **WinOpenPQG(***wattr*,*wtitle*,*ititle***);**
whandle = **WinOpenText(***wattr*,*wtitle*,*ititle***);**
whandle = **WinOpenTTY(***wattr*,*wtitle*,*ititle***);**
WinPan(*whandle*,*r*,*c***);**
WinPrint ⟦**/flush**⟧ ⟦*/typ*⟧ ⟦*/fmted*⟧ ⟦*/mf*⟧ ⟦*/jnt*⟧ *whandle* ⟦*expr1 expr2*...⟧;⟦**;**⟧
WinPrintPQG(*whandle*,*prtopts***);**
WinRefresh(*whandle***);**
WinRefreshArea(*whandle*,*row*,*col*,*nrows*,*ncols***);**
WinResize(*whandle*,*height*,*width***);**
whout = **WinSetActive(***whin***);**
oldback = **WinSetBackground(***whandle*,*color***);**
ret = **WinSetColorCells(***whandle*,*cellidx*,*colorspec***);**
mapinfo = **WinSetColormap(***whandle*,*cmap***);**
WinSetCursor(*whandle*,*r*,*c***);**
oldfore = **WinSetForeground(***whandle*,*color***);**
oldref = **WinSetRefresh(***whandle*,*refresh***);**
oldtwrap = **WinSetTextWrap(***whandle*,*textwrap***);**
WinZoomPQG(*whandle*,*x*,*y*,*xsize*,*ysize***);**

Appendix A

Fonts

There are four fonts available in Publication Quality Graphics.

Simplex	standard sans serif font
Simgrma	Simplex greek, math
Microb	bold and boxy
Complex	standard font with serif

The following tables show the characters available in each font and their ASCII values. See the **fonts** procedure to select fonts for your graph.

Simplex

33	!	61	=	89	Y	117	u
34	"	62	>	90	Z	118	v
35	#	63	?	91	[	119	w
36	$	64	@	92	\	120	x
37	%	65	A	93	]	121	y
38	&	66	B	94	^	122	z
39	'	67	C	95	_	123	{
40	(	68	D	96	`	124	|
41	)	69	E	97	a	125	}
42	*	70	F	98	b	126	~
43	+	71	G	99	c		
44	,	72	H	100	d		
45	-	73	I	101	e		
46	.	74	J	102	f		
47	/	75	K	103	g		
48	0	76	L	104	h		
49	1	77	M	105	i		
50	2	78	N	106	j		
51	3	79	O	107	k		
52	4	80	P	108	l		
53	5	81	Q	109	m		
54	6	82	R	110	n		
55	7	83	S	111	o		
56	8	84	T	112	p		
57	9	85	U	113	q		
58	:	86	V	114	r		
59	;	87	W	115	s		
60	<	88	X	116	t		

Simgrma

33	ϵ	61	$\neq$	89	Ψ	117	$\supset$
34	$($	62	$\geqq$	90	$\gtrsim$	118	$)$
35	$\equiv$	63	$\simeq$	91	$[$	119	ω
36	$\approx$	64	$\cup$	92	∂	120	ξ
37	$\uparrow$	65	$\frac{1}{2}$	93	$]$	121	ψ
38	$\surd$	66	$\frac{1}{3}$	94	$\cap$	122	ζ
39	$'$	67	H	95	$\downarrow$	123	$\{$
40	$\subset$	68	Δ	96	$''$	124	$\int$
41	$\supset$	69	$\frac{1}{8}$	97	α	125	$\}$
42	$\times$	70	Φ	98	β	126	$\propto$
43	$\pm$	71	Γ	99	η		
44	$\int$	72	X	100	δ		
45	$\mp$	73	$\frac{2}{3}$	101	ε		
46	$\cdot$	74	$\perp$	102	φ		
47	$\div$	75	$\frac{3}{8}$	103	γ		
48	∇	76	$\wedge$	104	χ		
49	$\surd$	77	$\frac{5}{8}$	105	ι		
50	$\oint$	78	$\frac{7}{8}$	106	ι		
51	$<$	79	$\frac{1}{4}$	107	κ		
52	$>$	80	Π	108	λ		
53	$/$	81	Θ	109	μ		
54	$\exists$	82	P	110	ν		
55	$\|$	83	Σ	111	o		
56	∞	84	$\lesssim$	112	π		
57	$\odot$	85	Υ	113	ϑ		
58	$\rightarrow$	86	$\leftrightarrow$	114	ρ		
59	$\leftarrow$	87	Ω	115	σ		
60	$\leqq$	88	Ξ	116	τ		

Microb

33	!	61	=	89	Y	117	u
34	"	62	>	90	Z	118	v
35	#	63	?	91	[	119	w
36	$	64	@	92	\	120	x
37	%	65	A	93	]	121	y
38	&	66	B	94	^	122	z
39	'	67	C	95	_	123	{
40	(	68	D	96	`	124	|
41	)	69	E	97	a	125	}
42	*	70	F	98	b	126	~
43	+	71	G	99	c		
44	,	72	H	100	d		
45	-	73	I	101	e		
46	.	74	J	102	f		
47	/	75	K	103	g		
48	0	76	L	104	h		
49	1	77	M	105	i		
50	2	78	N	106	j		
51	3	79	O	107	k		
52	4	80	P	108	l		
53	5	81	Q	109	m		
54	6	82	R	110	n		
55	7	83	S	111	o		
56	8	84	T	112	p		
57	9	85	U	113	q		
58	:	86	V	114	r		
59	;	87	W	115	s		
60	<	88	X	116	t		

Complex

33	!	61	=	89	Y	117	u
34	"	62	>	90	Z	118	v
35	#	63	?	91	[	119	w
36	$	64	@	92	\	120	x
37	%	65	A	93	]	121	y
38	&	66	B	94	^	122	z
39	'	67	C	95	_	123	{
40	(	68	D	96	`	124	|
41	)	69	E	97	a	125	}
42	*	70	F	98	b	126	~
43	+	71	G	99	c		
44	,	72	H	100	d		
45	-	73	I	101	e		
46	.	74	J	102	f		
47	/	75	K	103	g		
48	0	76	L	104	h		
49	1	77	M	105	i		
50	2	78	N	106	j		
51	3	79	O	107	k		
52	4	80	P	108	l		
53	5	81	Q	109	m		
54	6	82	R	110	n		
55	7	83	S	111	o		
56	8	84	T	112	p		
57	9	85	U	113	q		
58	:	86	V	114	r		
59	;	87	W	115	s		
60	<	88	X	116	t		

Appendix B

Performance Hints

Here are a few hints to maximize the performance of your new **GAUSS** System.

B.1 Library System

Some temporary files are created during the autoloading process. If you have a **tmp_path** configuration variable or a **tmp** environment string that defines a path on a ramdisk, the temporary files will be placed on the ramdisk.

```
set tmp=f:\tmp
```

This can speed things up quite a bit. (**tmp_path** takes precedence over the **tmp** environment variable.)

A disk cache will also help, as well as having your frequently used files in the first path in the **src_path**. You can optimize your library `.lcg` files by putting the correct drive and path on each file name listed in the library. The **lib** command will do this for you.

Use the **compile** command to precompile your large frequently used programs. This will completely eliminate compile time when those programs are rerun.

B.2 Loops

The use of the built-in matrix operators and functions rather than **do** loops will ensure that you are utilizing the potential of **GAUSS**.

Here is an example:

Given the vector **x** with 8000 normal random numbers.

```
x = rndn(8000,1);
```

You could get a count of the elements with an absolute value greater than 1 with a loop like this:

```
c = 0;
i = 1;
do while i <= rows(x);
    if abs(x[i]) > 1;
        c = c+1;
    endif;
    i = i+1;
endo;
print c;
```

Or you could do it like this:

```
c = sumc(abs(x) .> 1);
print c;
```

The loop takes over 40 times longer.

B.3 Virtual Memory

Following are a few hints for making the best use of virtual memory in **GAUSS**.

B.3.1 Data Sets

Large data sets can often be processed much faster by reading them in small sections (about 20000–40000 elements) instead of reading the whole thing in one piece. **maxvec** is used to control the size of a single disk read. Here is an example. The **ols** command can take either a matrix in memory or a data set on disk. Here are the times for a regression with 100 independent variables and 1500 observations on an IBM model 80 with 4 MB RAM running at 16 MHz.

ols(0,y,x)	matrices in memory	7 minutes 15.83 seconds
ols("olsdat",0,0)	data on disk (maxvec = 500000)	8 minutes 48.82 seconds
ols("olsdat",0,0)	data on disk (maxvec = 25000)	1 minute 42.77 seconds

As you can see from the table above, the fastest time was when the data was read from disk in small enough chunks to allow the **ols** procedure to execute entirely in core. This ensured that the only disk I/O was one linear pass through the data set.

The optimum size for **maxvec** depends on your available RAM and the algorithms you are using. **GAUSS** is being shipped with **maxvec** set to 20000. **maxvec** is a procedure defined in `system.src` that returns the value of the global scalar **__maxvec**. The value returned by a call to **maxvec** can be modified by editing `system.dec` and changing the value of **__maxvec**. The value returned when running **GAUSS Light** is always 8192.

Complex numbers use twice the space of real numbers, so the optimum single disk read size for complex data sets is half that for real data sets. You can set **__maxvec** for real data sets, then use **maxvec/2** when processing complex data sets. **iscplxf** will tell you if a data set is complex.

B.3.2 Precision

Under DOS and OS/2, the operators that use the Cholesky decomposition, **chol**, **invpd**, **solpd**, and **/** when used for regression, all use 80-bit temporary storage by default. The **prcsn** command controls the precision of these operators. Very substantial speed increases can be realized if 64-bit precision is selected, which is still full double precision. This speed increase is accentuated when virtual memory is being used heavily. The Unix and Windows versions of **GAUSS** always use 64-bit precision.

B.3.3 Hard Disk Maintenance

DOS, Windows, OS/2

The hard disk used for the swap file should probably be optimized occasionally with a disk optimizer. It is also important to ensure that the disk media is in good shape by using a disk maintenance program.

B.3.4 Algorithms

If possible, programs should be written so that variables that change in size often are located "above" variables that remain static in size. This means locating them toward the end of the **local** statement if they are local variables. If the variables are globals, they should be declared last. This can be accomplished by explicitly declaring the other globals at the top of your program in a **declare** statement. You can verify where certain variables are located in memory with the **show** command. The variables that change in size should be located at a higher address.

B.3.5 CPU Cache

There is a line for cache size in the `gauss.cfg` file. Set it to the size of the CPU data cache for your machine. The default is 32K.

This affects the choice of algorithms used for matrix multiply functions.

This won't change the results you get, but it can radically affect performance for large matrices.

Appendix C

Reserved Words

The following words are used for **GAUSS** intrinsic functions. You cannot use these names for variables or procedures in your programs.

abs	**cdfni**	**comlog**	**dbopen**
and	**cdftc**	**compile**	**dbstrerror**
atan	**cdftci**	**complex**	**debug**
atan2	**cdftvn**	**con**	**declare**
balance	**cdir**	**conj**	**delete**
band	**ceil**	**cons**	**det**
bandchol	**cfft**	**continue**	**detl**
bandcholsol	**cffti**	**conv**	**dfree**
bandltsol	**changedir**	**coreleft**	**diag**
bandrv	**chol**	**cos**	**diagrv**
bandsolpd	**choldn**	**counts**	**disable**
besselj	**cholsol**	**countwts**	**dlibrary**
bessely	**cholup**	**create**	**dllcall**
break	**chrs**	**crout**	**do**
call	**cint**	**croutp**	**dos**
callexe	**clear**	**csrcol**	**dtvnormal**
cdfbeta	**clearg**	**csrlin**	**dtvtoutc**
cdfbvn	**close**	**csrtype**	**ed**
cdfchic	**closeall**	**cvtos**	**edit**
cdffc	**cls**	**date**	**editm**
cdfgam	**color**	**dbcommit**	**eig**
cdfn	**cols**	**dbconnect**	**eigh**
cdfnc	**colsf**	**dbdisconnect**	**eighv**

eigv
else
elseif
enable
end
endfor
endif
endo
endp
envget
eof
eq
eqv
erf
erfc
error
errorlog
exec
exp
external
eye
fcheckerr
fclearerr
fflush
fft
ffti
fftn
fgets
fgetsa
fgetsat
fgetst
fileinfo
files
filesa
fix
floor
fmod
fn
font
fontload
fontunload
fontunloadall
fopen
for
format
fputs
fputst
fseek
fstrerror
ftell
ftocv
ftos
gamma
ge
getf
getname
getnamef
gosub
goto
graph
graphsev3
gt
hasimag
hess
hsec
if
imag
indcv
indexcat
indnv
int
inv
invpd
invswp
iscplx
iscplxf
ismiss
key
keyw
keyword
le
let
lib
library
line
ln
load
loadexe
loadf
loadk
loadm
loadp
loads
local
locate
log
lower
lpos
lprint
lpwidth
lshow
lt
ltrisol
lu
lusol
matrix
maxc
maxindc
meanc
minc
minindc
miss
missrv
moment
msym
ndpchk
ndpclex
ndpcntrl
ne
new
not
oldfft
oldffti
ones
open
openpqg
or
output
outwidth
packr
parse
pdfn
pi
plot
plotsym
pop
pqgwin
prcsn
print
printdos
printfm
proc
prodc
push
rankindx
rcondl
readr
real
recserar
recsercp
reshape
retp
return
rev
rfft
rffti
rfftip
rfftn
rfftnp
rfftp
rndcon
rndmod
rndmult
rndn
rndns
rndseed
rndu
rndus
rotater
round
rows
rowsf

run
save
saveall
scalerr
scalmiss
schur
screen
scroll
seekr
seqa
seqm
setvmode
shiftr
show
showpqg
sin
sleep
solpd
sortc
sortcc
sorthc
sorthcc
sortind
sortindc
sqrt
stdc
stocv
stof
stop
strindx
string
strlen
strrindx
strsect
submat
subscat
sumc
svdcusv
svds
svdusv
sysstate
system
tab
tan
tempname
time
timeutc
trace
trap
trapchk
trim
trimr
trunc
type
typecv
typef
union
uniqindx
unique
until
upper
use
utctodtv
utrisol
vals
varget
vargetl
varput
varputl
vartypef
vec
vech
vecr
vfor
while
winclear
wincleararea
winclearttylog
winclose
wincloseall
winconvertpqg
wingetactive
wingetattributes
wingetcolorcells
wingetcursor
winmove
winopenpqg
winopentext
winopentty
winpan
winprint
winprintpqg
winrefresh
winrefresharea
winresize
winsetactive
winsetbackground
winsetcolor
winsetcolorcells
winsetcolormap
winsetcursor
winsetforeground
winsetrefresh
winsettextwrap
winwrite
winzoompqg
writer
xor
xpnd
zeros

Appendix D

Operator Precedence

The order in which an expression is evaluated is determined by the precedence of the operators involved and the order in which they are used. For example, the ***** and **/** operators have a higher precedence than the **+** and − operators. In expressions which contain the above operators, the operand pairs associated with the ***** or **/** operator are evaluated first. Whether ***** or **/** is evaluated first depends on which comes first in the particular expression.

The expression:

```
-5+3/4+6*3
```

is evaluated as

$$(-5) + (3/4) + (6 * 3)$$

Within a term, operators of equal precedence are evaluated from left to right.

The precedence of all operators from the highest to the lowest is listed in the following table:

operator	precedence	operator	precedence
.′	90	**.lt**	65
′	90	**.ne**	65
!	89	**.not**	64
.^	85	**.and**	63
^	85	**.or**	62
(unary −)	83	**.xor**	61
*****	80	**.eqv**	60
***~**	80	**$/=**	55
.*	80	**$<**	55
.*.	80	**$<=**	55
./	80	**$==**	55
/	80	**$>**	55
%	75	**$>=**	55
$+	70	**/=**	55
+	70	**<**	55
−	70	**<=**	55
~	68	**==**	55
|	67	**>**	55
.$/=	65	**>=**	55
.$<	65	**eq**	55
.$<=	65	**ge**	55
.$==	65	**gt**	55
.$>	65	**le**	55
.$>=	65	**lt**	55
./=	65	**ne**	55
.<	65	**not**	49
.<=	65	**and**	48
.==	65	**or**	47
.>	65	**xor**	46
.>=	65	**eqv**	45
.eq	65	**(space)**	35
.ge	65	**:**	35
.gt	65	**=**	10
.le	65		

Appendix E

Singularity Tolerance

The tolerance used to determine whether or not a matrix is singular can be changed. The default value is 1.0e-14 for both the LU and the Cholesky decompositions. The tolerance for each decomposition can be changed separately. The following operators are affected by a change in the tolerance:

Crout LU Decomposition

crout(x**)**	
croutp(x**)**	
inv(x**)**	
det(x**)**	
y/x	when neither x nor y is scalar and x is square.

Cholesky Decomposition

chol(x**)**	
invpd(x**)**	
solpd(y**,**x**)**	
y/x	when neither x nor y is scalar and x is not square.

E.1 Reading and Setting the Tolerance

The tolerance value may be read or set using the **sysstate** function, cases 13 and 14.

E.2 Determining Singularity

There is no perfect tolerance for determining singularity. The default is 1.0e-14. You can adjust this as necessary.

A numerically better method of determining singularity is to use **cond** to determine the condition number of the matrix. If the equation:

1 / cond(x**) + 1 eq 1**

is true, then the matrix is usually considered singular to machine precision. See LINPACK for a detailed discussion on the relationship between the matrix condition and the number of significant figures of accuracy to be expected in the result.

Dos, **OS/2 only**

It should be kept in mind that the math coprocessor uses 80-bit precision internally and the Cholesky-based functions in **GAUSS** keep all intermediate results in 80-bit precision, as long as there is enough free memory and **prcsn** is set to 80. In the Crout LU algorithm, the inner products are accumulated in 80-bit precision as well, if **prcsn** is set to 80, so the final results may be more accurate than the estimation would suggest.

We have chosen not to add the overhead of estimating the reciprocal condition automatically. Users can compute the condition number directly when they feel it is appropriate.

Appendix F

Saving Compiled Procedures

When a file containing a procedure definition is run, the procedure is compiled and is then resident in memory. The procedure can be called as if it were an intrinsic function. If the **new** command is executed or you quit **GAUSS** and exit to the operating system, the compiled image of the procedure disappears and the file containing the procedure definition will have to be compiled again.

If a procedure contains no global references, i.e., if it does not reference any global matrices or strings and it does not call any user-defined functions or procedures, it can be saved to disk in compiled form in an `.fcg` file with the **save** command and loaded later with the **loadp** command whenever it is needed. This will usually be faster than recompiling. The syntax is:

```
save path = c:\gauss\cp proc1,proc2,proc3;

loadp path = c:\gauss\cp proc1,proc2,proc3;
```

The name of the file will be the same as the name of the procedure with an `.fcg` extension. See **loadp** and **save** for more details.

It is strongly recommended that all compiled procedures be saved in the same subdirectory so that there is no question where they are located when it is necessary to reload them. The **loadp** path can be set in your `startup` file to reflect this. Then, to load in procedures you can just say:

```
loadp proc1,proc2,proc3;
```

Procedures that are saved in `.fcg` files will NOT be automatically loaded. It is necessary to explicitly load them with **loadp**. This feature should only be used when the time necessary for the autoloader to compile the source is too great. Also, unless these procedures have been compiled with **#lineson**, debugging will be more complicated.

Appendix G

transdat

transdat converts data sets, matrix files and string files from one file format to another. It currently supports the following formats:

- **v89** Intel (DOS)
- **v92** SPARC, RS/6000, HP 9000, SGI, DEC, Intel (Solaris x86)
- **v96** SPARC, RS/6000, HP 9000, SGI, DEC, Intel (Solaris x86, DOS, OS/2, Windows NT)

The **v89** format is supported only by **GAUSS** for DOS. The **v92** format is supported only by **GAUSS** for UNIX. The **v96** format is supported on all platforms.

The **v89** format keeps its file header information in a separate `.dht` file. The **v92** and **v96** formats keep their header information in the `.dat` file.

The **v92** and **v96** formats support both forward- and backward-byte ordering of the actual data in the file, depending on the hardware platform. SPARC, RS/6000, HP 9000, and SGI are forward-byte platforms, DEC and Intel are backward-byte.

To run **transdat** from the UNIX system prompt, type:

`transdat` [[*flag...*]] *filename* [[*flag...*]] [[*filename...*]]

There are five types of flags: data format flags, byte order flags, file permissions flags, matrix file type flags and the information flag. Data format flags take the form '−**v#**', where '−**v#**' is one of the following:

−**v89**	v89 format (DOS only)
−**v92**	v92 format (UNIX only)
−**v96**	v96 format (all platforms)

The −**v89** flag implies backward-byte order; if you specify it, any byte order flags are ignored.

The byte order flags indicate the desired byte order for numeric data in the converted data file. They are:

−**fby**	forward-byte order (SPARC, RS/6000, HP 9000, SGI)
−**bby**	backward-byte order (DEC, Intel)
−**swap**	swap the byte order

String data is always stored the same so you do not need to specify a byte order flag when converting string files.

The **-swap** flag tells **transdat** to determine the current byte order of the file and reverse it.

Matrix file type flags indicate the type of data in a matrix file:

−**n**	data is numeric
−**c**	data is character

The −**c** flag is used to prevent byte-order reversal when converting character data matrix files. −**n** and −**c** are ignored for data sets and string files.

File permissions flags take the form '−**0###**', where '**###**' is an octal number indicating owner, group and world permissions. This is the same as the numeric file permissions argument used by the UNIX **chmod** command. So, for example, you could specify '−**0664**' for the following permissions:

Owner	**read/write**
Group	**read/write**
World	**read**

The information flag is:

−**i**	format information only

When you specify **-i**, transdat will report the formats of the files specified without converting them. Note that you can turn conversion back on with a subsequent **-v#** format flag.

All flags are optional. If you do not specify a data format flag, files will be converted to the format of the machine you are running **transdat** on. If you do not specify a matrix file type flag, matrix files will be handled as numeric data. If you do not specify a permissions flag, the file will be converted with the default **chmod** permissions of **0644**:

Owner	**read/write**
Group	**read**
World	**read**

Multiple flags can be listed; files will be converted to the format of the format flag they follow, with the permissions settings of the permissions flag they follow. Matrix files will be handled according to the type flag they follow.

You can list data set files without an extension; **transdat** assumes `.dat` as the default. To convert matrix and string files, you must supply the extension. **transdat** currently recognizes only the standard extensions for the various types of files—`.dht`, `.dat`, `.fmt` and `.fst`. It will not try to convert files with alternate extensions.

WARNING!! MAKE SURE YOU HAVE BACKUPS before converting your files. Because data sets can be very large, **transdat** converts files in place without making backups.

Run **transdat** without any parameters for a reminder on how to use it.

Appendix H

Error Messages

Following is a list of error messages that are intrinsic to the **GAUSS** programming language. Other error messages generated by library functions are not included here.

G0002 File too large

load Input file too large.

getf Input file too large.

G0003 Indexing a matrix as a vector

A single index can be used only on vectors. Vectors have only one row or only one column.

G0004 Compiler stack overflow - too complex

An expression is too complex. Break it up into smaller pieces. Please notify Aptech Systems of the occurrence of this error.

G0005 File is already compiled

G0006 Statement too long

G0007 End of file encountered

G0008 Syntax error

Compiler Unrecognizable or incorrect syntax. Semicolon missing on previous statement.

create Unrecognizable statement in command file or **numvar** or **outvar** statement error.

G0009 Compiler pass out of memory

Compiler pass has run out of memory. Please notify Aptech Systems of the circumstances under which this happened.

G0010 Can't open output file

G0011 Compiled file must have correct extension

GAUSS requires a **.gcg** extension.

G0012 Invalid drive specifier

G0013 Invalid filename

G0014 File not found

G0015 Directory full

G0016 Too many #includes

#included files are nested too deep.

G0017 WARNING: local statement outside of procedure

A **local** statement has been found outside a procedure definition. The **local** statement will be ignored.

G0018 Read error in program file

G0019 Can't edit .gcg file

G0020 Not implemented yet

Command not yet supported in this implementation.

G0021 use must be at the beginning of a program

G0022 User keyword cannot be used in expression

G0023 Illegal attempt to redefine symbol to an index variable

G0025 Undefined symbol

A symbol has been referenced that has not yet been given a definition.

G0026 Too many symbols

The global symbol table is full. The limit can be set with the **new** command.

G0027 Invalid directory

G0028 Can't open configuration file

GAUSS can't find its configuration file.

G0029 Missing left parenthesis

G0030 Insufficient workspace memory

The space used to store and manipulate matrices and strings is not large enough for the operations attempted.

G0031 Execution stack too deep - expression too complex

An expression is too complex. Break it up into smaller pieces. Please notify Aptech Systems of the occurrence of this error.

G0032 fn function too large

G0033 Missing right index bracket

G0034 Missing arguments

G0035 Argument too large

G0036 Matrices are not conformable

See the description of the function or operator being used and also the conformability rules in the *Operators* chapter in Volume I.

G0037 Result too large

The size of the result of an expression is greater than the limit for a single matrix.

G0038 Not all the eigenvalues can be computed

G0039 Matrix must be square to invert

G0040 Not all the singular values can be computed

G0041 Argument must be scalar

A matrix argument was passed to a function that requires a scalar.

G0042 Matrix must be square to compute determinant

G0043 Not implemented for complex matrices

G0044 Matrix must be real

G0045 Attempt to write complex data to real data set

Data sets, unlike matrices, cannot change from real to complex after they are created. Use **create complex** to create a complex data set.

G0046 Columns don't match

The matrices must have the same number of columns.

G0047 Rows don't match

The matrices must have the same number of rows.

G0048 Matrix singular

The matrix is singular using the current tolerance.

G0049 Target matrix not complex

G0050 Out of memory for program

The main program area is full. See the configuration file.

G0051 Program too large

The main program area is full. See the configuration file.

G0052 No square root - negative element

G0053 Illegal index

An illegal value has been passed in as a matrix index.

G0054 Index overflow

An illegal value has been passed in as a matrix index.

G0055 retp outside of procedure

A **retp** statement has been encountered outside of a procedure definition.

G0056 Too many active locals

The execution stack is full. There are too many local variables active. You will have to restructure your program. Please inform Aptech Systems that you have encountered this error and the circumstances of its appearance.

G0057 Procedure stack overflow - expression too complex

The execution stack is full. There are too many nested levels of procedure calls. You will have to restructure your program. Please inform Aptech Systems that you have encountered this error and the circumstances of its appearance.

G0058 Index out of range

You have referenced a matrix element that is out of bounds for the matrix being referenced.

G0059 exec command string too long

G0060 Nonscalar index

G0061 Cholesky downdate failed

G0062 Zero pivot encountered

crout The Crout algorithm has encountered a diagonal element equal to 0. Use **croutp** instead.

G0063 Operator missing

An expression contains two consecutive operands with no intervening operator.

G0064 Operand missing

An expression contains two consecutive operators with no operand in between.

G0065 Division by zero!

G0066 Must be recompiled under current version

You are attempting to use compiled code from a previous version of **GAUSS**. Recompile the source code under the current version.

G0067 Symbol to examine

G0068 Program compiled under GAUSS-386 real version

G0069 Program compiled under GAUSS-386i complex version

G0070 Procedure calls too deep

You have probably got a runaway recursive procedure.

G0071 Type mismatch

You are using a string where a matrix is called for or vice versa.

G0072 Too many files open

The limit on simultaneously open files is 10.

G0073 Redefinition of

declare An attempt has been made to initialize a variable that is already initialized. This is an error when **declare :=** is used. **declare !=** or **declare ?=** may be a better choice for your application.

declare An attempt has been made to redefine a string as a matrix or procedure or vice versa. You may be able to **delete** the symbol and try again. If this is happening in the context of a single program, you have a programming error. If this is a conflict between different programs, you can use a **new** statement before running the second program.

let A string is being forced to type matrix. Use an **external matrix** *symbol*; statement before the **let** statement.

G0074 Can't run program compiled under GAUSS Light

G0075 GSCROLL input vector the wrong size

G0076 Call Aptech Systems Technical Support

G0077 New size cannot be zero

You can't **reshape** a matrix to a size of zero.

G0078 vargetl outside of procedure

G0079 varputl outside of procedure

G0080 File handle must be an integer

G0081 Error renaming file

G0082 Error reading file

G0083 Error creating temporary file

G0084 Too many locals

A procedure has too many local variables.

G0085 Invalid file type

You can't use this kind of file in this way.

G0086 Error deleting file

G0087 Couldn't open

The auxiliary output file could not be opened. Check the file name and make sure there is room on the disk.

G0088 Not enough memory to convert the whole string

G0089 Warning: duplicate definition of local

G0090 Label undefined

Label referenced has no definition.

G0091 Symbol too long

Symbols can be no longer than 32 characters.

G0092 Open comment

A comment was never closed.

G0093 Locate off screen

G0094 Argument out of range

G0095 Seed out of range

G0096 Error parsing string

G0097 String not closed

A string must have double quotes at both ends.

G0098 Invalid character for imaginary part of complex number

G0099 Illegal redefinition of user keyword

G0100 Internal E R R O R ###

Please notify Aptech Systems of the circumstances of this error and the number.

G0101 Argument cannot be zero

The argument to **ln** or **log** cannot be zero in **GAUSS**.

G0102 Subroutine calls too deep

Too many levels of **gosub**. Restructure your program.

G0103 return without gosub

You have run into a subroutine without executing a **gosub**.

G0104 Argument must be positive

G0105 Bad expression or missing arguments

Check the expression in question. You probably forgot an argument.

G0106 Factorial overflow

G0107 Nesting too deep

Break up the offending expression into smaller statements.

G0108 Missing left bracket [

G0109 Not enough data items

You left out some data in a **let** statement.

G0110 Found) expected] -

G0111 Found] expected) -

G0112 Matrix multiplication overflow

G0113 Unclosed (

G0114 Unclosed [

G0115 Illegal redefinition of function

You are trying to turn a function into a matrix or string. If this is a name conflict, you can **delete** the function.

G0116 sysstate: invalid case

G0117 Invalid argument

G0118 Argument must be integer

G0119 Invalid window handle

G0120 Illegal type for save

G0121 Matrix not positive definite

The matrix is either not positive definite or singular using the current tolerance.

G0122 Bad file handle

The file handle does not refer to an open file or is not in the valid range for file handles.

G0123 File handle not open

The file handle does not refer to an open file.

G0124 readr call too large

You are trying to read too much in one call.

G0125 Read past end of file

You have already reached the end of the file.

G0126 Error closing file

G0127 File not open for write

G0128 File already open

G0129 File not open for read

G0130 No output variables specified

G0131 Can't create file, too many variables

G0132 Can't write, disk probably full

G0133 Function too long

G0134 Can't seekr in this type of file

G0135 Can't seek to negative row

G0136 Too many arguments or misplaced assignment op...

You have an assignment operator (=) where you want a comparison operator (==), or you have too many arguments.

G0137 Negative argument - erf or erfc

G0138 User keyword must have one argument

G0139 Negative parameter - Incomplete Beta

G0140 Invalid second parameter - Incomplete Beta

G0141 Invalid third parameter - Incomplete Beta

G0142 Nonpositive parameter - gamma

G0143 NaN or missing value - cdfchic

G0144 Negative parameter - cdfchic

G0145 Second parameter $<$ 1.0 - cdfchic

G0146 Parameter too large - Incomplete Beta

G0147 Bad argument to trig function

G0148 Angle too large to trig function

G0149 Matrices not conformable

See the description of the function or operator being used and also the section on conformability rules in the *Operators* chapter in Volume I.

G0150 Matrix not square

G0151 Sort failure

G0152 Variable not initialized

You have referenced a variable that has not been initialized to any value yet.

G0153 Unsuccessful close on auxiliary output

The disk is probably full.

G0154 Illegal redefinition of string

G0155 Nested procedure definition

A **proc** statement was encountered inside a procedure definition.

G0156 Illegal redefinition of procedure

You are trying to turn a procedure into a matrix or string. If this is a name conflict, you can **delete** the procedure.

G0157 Illegal redefinition of matrix

G0158 endp without proc

You are trying to end something you never started.

G0159 Wrong number of parameters

You called a procedure with the wrong number of arguments.

G0160 Expected string variable

G0161 User keywords return nothing

G0162 Can't save proc/keyword/fn with global references

You can remove the global references or just leave this in source code form and let the autoloader handle it. See **library**.

G0163 Wrong size format matrix

G0164 Bad mask matrix

G0165 Type mismatch or missing arguments

G0166 Character element too long

The maximum length for character elements is 8 characters.

G0167 Argument must be column vector

G0168 Wrong number of returns

The procedure was defined to return a different number of items.

G0169 Invalid pointer

You are attempting to call a local procedure using an invalid procedure pointer.

G0170 Invalid use of ampersand

G0171 Called symbol is wrong type

You are attempting to call a local procedure using a pointer to something else.

G0172 Can't resize temporary file

G0173 varindx failed during open

The global symbol table is full.

G0174 '.' and ' ' operators must be inside [] brackets

These operators are for indexing matrices.

G0175 String too long to compare

G0176 Argument out of range

G0177 Invalid format string

G0178 Invalid mode for getf

G0179 Insufficient heap space

G0180 Trim too much

You are attempting to trim more rows than the matrix has.

G0181 Illegal assignment - type mismatch

G0182 2nd and 3rd arguments different order

G0184-G0272 OS error messages

These error messages are operating system error messages.

G0274 Invalid parameter for conv

G0275 Parameter is NaN (Not A Number)

The argument is a **NaN**.

G0276 Illegal use of reserved word

G0277 Null string illegal here

G0278 proc without endp

You need to terminate a procedure definition with an **endp** statement.

G0279 NDP invalid operation

G0280 NDP denormalized operand

G0281 NDP zero divide

G0282 NDP overflow

G0283 NDP underflow

G0284 NDP unknown exception

G0286 Multiple assign out of memory

G0287 Seed not updated

The seed argument to **rndns** and **rndus** must be a simple local or global variable reference. It cannot be an expression or constant.

G0288 Found break not in loop

G0289 Found continue not in loop

G0290 Library not found

The specified library cannot be found on the **lib_path** path. Make sure you followed the installation correctly.

G0291 Compiler pass out of memory

Notify Aptech Systems if you get this error.

G0292 File listed in library not found

A file listed in a library could not be opened.

G0293 Procedure has no definition

The procedure was not initialized. Define it and try again.

G0294 Error opening temporary file

One of the temporary files could not be opened. The directory may be full.

G0295 Error writing temporary file

One of the temporary files could not be written to. The disk is probably full.

G0296 Can't raise negative number to nonintegral power

G0297 This version linked without built-in graphics

G0300 File handle must be a scalar

G0301 Syntax error in library

G0302 File has been truncated or corrupted

getname File header cannot be read.

load Can't read input file or file header cannot be read.

open File size does not match header specifications or file header cannot be read.

G0317 Can't open temp file

G0319 No marked block

G0320 Block too large

G0330 Disk full - trying to save:

G0331 No command start character

G0332 No command stop character

G0333 Block too large for scrap

G0334 No scrap to insert

G0335 Insert OK, scrap lost

G0336 Disk full

G0339 Can't debug compiled program

G0341 File too big

G0347 Can't allocate that many globals

G0351 Warning: Not reinitializing : declare ?=

The symbol is already initialized. It will be left as is.

G0352 Warning: Reinitializing : declare !=

The symbol is already initialized. It will be reset.

G0355 Wrong size line matrix

G0356 loadexe - code buffer too small

G0358 callexe - code buffer too small

G0359 Can't allocate stack space

G0360 Write error

G0364 Paging error

G0365 Unsupported executable file type

G0368 Unable to allocate translation space

G0369 Unable to allocate buffer

G0370 Syntax Error in code statement

G0371 Syntax Error in recode statement

G0372 Token verify error

Please inform Aptech Systems that you have encountered this error and the circumstances of its appearance.

G0373 Procedure definition not allowed

A procedure name appears on the left side of an assignment operator.

G0374 Invalid make statement

G0375 make Variable is a Number

G0376 make Variable is Procedure

G0377 Cannot make Existing Variable

G0378 Cannot make External Variable

G0379 Cannot make String Constant

G0380 Invalid vector statement

G0381 vector Variable is a Number

G0382 vector Variable is Procedure

G0383 Cannot vector Existing Variable

G0384 Cannot vector External Variable

G0385 Cannot vector String Constant

G0386 Invalid extern statement

G0387 Cannot extern number

G0388 Procedures always external

A procedure name has been declared in an **extern** statement. This is a warning only.

G0389 extern variable already local

A variable declared in an **extern** statement has already been assigned local status.

G0390 String constant cannot be external

G0391 Invalid code statement

G0392 code variable is a number

G0393 code variable is procedure

G0394 Cannot code existing variable

G0395 Cannot code external variable

G0396 Cannot code string constant

G0397 Invalid recode statement

G0398 Recode variable is a number

G0399 Recode variable is procedure

G0400 Cannot recode external variable

G0401 Cannot recode string constant

G0402 Invalid keep statement

G0403 Invalid drop statement

G0404 Cannot define number

G0405 Cannot define string

G0406 Invalid select statement

G0407 Invalid delete statement

G0408 Invalid outtyp statement

G0409 outtyp already defaulted to 8

Character data has been found in the output data set before an **outtyp 2** or **outtyp 4** statement. This is a warning only.

G0410 outtyp must equal 2, 4, or 8

G0411 outtyp override...precision set to 8

Character data has been found in the output data set after an **outtyp 2** or **outtyp 4** statement. This is a warning only.

G0412 default not allowed in recode statement

default only allowed in **code** statement.

G0413 Missing file name in dataloop statement

G0414 Invalid listwise statement

G0415 Invalid lag statement

G0416 Lag Variable is a Number

G0417 Lag Variable is a Procedure

G0418 Cannot lag External Variable

G0419 Cannot lag String Constant

G0420 Can't memory map file

G0421 compile command not supported in Run-Time module

G0422 Invalid font handle

G0423 X Windows version required

G0424 Cannot allocate colorcell

G0425 Command not available for Window 1

G0426 Video mode not supported

G0427 Font not available

G0428 Cannot use DEBUG command inside program

G0429 Invalid number of subdiagonals

G0430 Internal Xlib error

G0431 Error closing dynamic library

G0432 Error opening dynamic library

G0433 Cannot find DLL function

G0434 Error opening default dynamic library

G0435 Invalid mode

G0436 Matrix is empty

G0437 loadexe not supported under UNIX; use dlibrary instead

G0438 callexe not supported under UNIX; use dllcall instead

G0439 File has wrong bit order

G0440 File has wrong byte order

G0441 Type vector malloc failed

G0442 No type vector in gfblock

G0443 Expected flag

G0444 Invalid flag

G0445 Illegal left-hand side reference in procedure

G0447 vfor called with illegal loop level

G0448 Window not resizable

G0449 Invalid window type

G0450 Command not valid for active window

Appendix I

Limits

The current implementation of **GAUSS** has the following limits:

	GAUSS	GAUSS Light
Columns in data set	limited by RAM	8192
Elements in matrix	limited by RAM	8192
Global symbols	2000	2000
Levels of recursion	1000	1000
Local variables active	2000	2000
Maximum code block, bytes	limited by RAM	limited by RAM
Square matrix	limited by RAM	90x90
Statement length	10000	10000
String length	limited by RAM	limited by RAM

A **code block** is a contiguous section of compiled pseudo-code. This then is the maximum size for the main program segment or for a compiled procedure.

For the items above that are listed as **limited by RAM**, the assumption is made that the amount of RAM in the computer is less than 4GB or 4,294,967,296 bytes.

The "RAM" amount is actually limited by:

1. The amount of free space on your hard disk, or

2. The size of workspace you allocate (specified by the **workspace** variable in `gauss.cfg`.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z