GAUSS™

Obsoleted Commands

user. A full copy of this license and the JNA source code have been included with the distribution.

Version 11
Documentation Revision: 1074 March 1, 2012

Contents

1 Obsoleted Command Reference

color

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 **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.

coreleft

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.

csrtype

PURPOSE Sets the cursor shape.

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

This function is not supported in terminal mode.

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);
```

denseSubmat

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.

This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE sparse.src

NOW USE **spDenseSubmat**

dfree

PURPOSE Returns the amount of room left on a diskette or hard disk.

disable

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.

disable

PURPOSE Disables the invalid operation interrupt of the numeric processor. This affects the way missing values are handled in most calculations.

FORMAT **disable;**

PORTABILITY **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.**

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: *d*E+*n* or *d*E-*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.

eigcg

PURPOSE Computes the eigenvalues of a complex, general matrix.

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.

GLOBAL OUTPUT

_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**

NOW USE **eig**

eigcg2

PURPOSE Computes eigenvalues and eigenvectors of a complex, general matrix.

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.

GLOBAL OUTPUT

_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**

NOW USE **eigv**

eigch

PURPOSE Computes the eigenvalues of a complex, hermitian matrix.

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.

GLOBAL OUTPUT
_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**

NOW USE **eigh**

eigch2

PURPOSE Computes eigenvalues and eigenvectors of a complex, hermitian matrix.

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.

GLOBAL OUTPUT

_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**

NOW USE **eighv**

eigrg

PURPOSE Computes the eigenvalues of a real, general matrix.

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.

“ttfamily “bfseries “upshape LaTeX Error: “beginitList on input line 709 ended by “endargumentlist.ΩΩSee the LaTeX manual or LaTeX Companion for explanation.ΩType´H´¡return¿´for´immediate´help

GLOBAL OUTPUT **_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`**

NOW USE **`eig`**

eigrg2

PURPOSE Computes eigenvalues and eigenvectors of a real, general matrix.

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.

GLOBAL OUTPUT

_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**

NOW USE **eigv**

eigrs

PURPOSE Computes the eigenvalues of a real, symmetric matrix.

FORMAT *va* **= eigrs(***x***);**

INPUT *x* N×N matrix.

OUTPUT *va* N×1 vector, eigenvalues of *x*.

GLOBAL OUTPUT **_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**

NOW USE **eigh**

eigrs2

PURPOSE Computes eigenvalues and eigenvectors of a real, symmetric matrix.

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*.

GLOBAL OUTPUT **_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**

NOW USE **eighv**

enable

PURPOSE Enables the invalid operation interrupt of the numeric processor. This affects the way missing values are handled in most calculations.

FORMAT **enable;**

PORTABILITY **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.**

files

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 8×3 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.

NOW USE **fileinfo**, **filesa**

graph

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

FORMAT **graph** *x*,*y*;

INPUT

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.

NOW USE **xy**

isSparse

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.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE `sparse.src`

NOW USE **`type`**

line

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.

lpos

PURPOSE Returns the current position of the print head within the printer buffer for the printer.

FORMAT *y* **= lpos;**

OUTPUT *y* scalar, current position of the print head within the printer buffer.

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.

lprint

PURPOSE Controls printing to the line printer.

FORMAT **lprint** [[*/typ*]] [[*/fmted*]] [[*/mf*]] [[*/jnt*]] [[*list of expressions*]] [[**;**]]

REMARKS This function was originally written for line printers. It is still supported for backwards compatibility purposes, but if you're using a page-oriented printer (such as a laser or inkjet printer), it may not give you the results you expect.

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 **print** for information on */typ*, */fmted*, */mf* and */jnt*.

The *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).

EXAMPLE

```
lprint 3*4 5+2;
```

lprint on, lprint off

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.

lpwidth

lpwidth

PURPOSE Specifies the width of the printer.

FORMAT **lpwidth** *n*;

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.

lshow

PURPOSE Prints information about the global symbol table.

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

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, information about all symbols in the symbol table will be printed.

Here is an example listing with an explanation of the columns. Note that lshow does not print the column titles shown here:

```
 Mem Used     Name           Cplx  Type          References        Info
360 bytes     _xweight             MATRIX                          45,1
216 bytes     a                    ARRAY         3 dims             2,3,4
144 bytes     area                 FUNCTION      local refs         1=1
224 bytes     ms                   STRUCT        mystruct           1,1
 88 bytes     sa                   STRING ARRAY                     3,2
432 bytes     sharedrange          PROCEDURE     local refs         2=2
 24 bytes     ttl                  STRING                          16 char
144 bytes     weightedx            PROCEDURE     global refs        1=1
144 bytes     x                C   MATRIX                          3,3
```

The '`Mem Used`' column tells the amount of memory used by each item.

The '`Name`' column tells the name of each symbol.

The '`Cplx`' column contains a '`C`' if the symbol is a complex matrix.

The '`Type`' column specifies the type of each symbol. It can be `ARRAY`, `FUNCTION`, `KEYWORD`, `MATRIX`, `PROCEDURE`, `STRING`, `STRING ARRAY`, or `STRUCT`.

If the symbol is a procedure, keyword or function, the 'References' column will show if it makes any global references. If it makes only local references, the procedure or function can be saved to disk in an .fcg file with the **save** command. If the function or procedure makes any global references, it cannot be saved in an .fcg file.

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 or a string array, then the 'Info' column gives the number of rows and columns. If the symbol is a string, then it gives the number of characters in the string. If the symbol is an N-dimensional array, then it gives the orders of each dimension. As follows:

Rets=Args	if procedure or function
Row,Col	if matrix or string array
Length	if string
OrdN,. . . ,Ord2,Ord1	if array, where N is the slowest moving dimension of the array, and Ord is the order (or size) of a dimension

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

```
lshow -fpg eig*;
```

This command will print information about all functions and procedures that have global references and begin with **eig**.

```
lshow -m;
```

This command will print information about all matrices.

medit

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* ± *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**

nametype

PURPOSE Provides support for programs following the upper/lowercase convention in **GAUSS** data sets. 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 a 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;

     AGE
     PAY
```

```
sex
```

SOURCE **nametype.src**

ndpchk

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 OR'ing 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.

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.

ndpcntrl

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.

plot

PURPOSE Plots the elements of two matrices against each other in text mode.

FORMAT **plot** *x*, *y*;

PORTABILITY **OS/2**, **Windows**

plot writes to the main **GAUSS** 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.

NOW USE **xy**

plotsym

PURPOSE Controls the symbol or character to be plotted by **plot**.

FORMAT **plotsym** *n*;

REMARKS *n* is a scalar containing the ASCII value of the symbol to be plotted. This value must be in the range 0-255.

n may be any legal expression that returns a scalar. Nonintegers will be truncated to an integer before being evaluated by **plotsym**.

The default symbol is an asterisk (*), which has an ASCII value of 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**.

prcsn

PURPOSE Sets the computational precision of some of the matrix operators.

FORMAT **prcsn** *n*;

INPUT *n* scalar, 64 or 80

PORTABILITY **UNIX**, **Windows**

This function has no effect under UNIX or Windows. All computations are done in 64-bit precision (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.

print on, print off

PURPOSE Switches the automatic window 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 to the window. Assignments to local matrices or strings will not be printed.

The name and dimensions of the resulting matrix will also be printed. If the result being assigned is 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**.

rndns, rndus

PURPOSE An alternate way of generating Normal (**rndns**) or uniform (**rndus**) random numbers using a seed value as one of the arguments.

FORMAT *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.

NOW USE **rndKMn**, **rndKMu**, **rndLCn**, **rndLCu**

scroll

PURPOSE Scrolls a section of the window.

FORMAT **scroll** *v*;

INPUT *v* 6×1 vector.

PORTABILITY **Windows** only

REMARKS This command is intended to be used in the DOS compatibility window to support legacy programs.

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 window 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

If the number of lines (element 5) is 0, the entire window will be blanked.

EXAMPLE

```
let v = 1 1 12 80 5 7;
scroll v;
```

This call would scroll an 80-column wide section of the window covering the upper twelve rows of the window. The section would be scrolled up 5 lines and the new lines would be displayed in regular text mode.

setvmode

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 *y* 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.

sparseCols

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 in *x*.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE sparse.src

NOW USE **cols**

sparseEye

PURPOSE Returns a sparse identity matrix.

FORMAT *y* = **sparseEye(***n***);**

INPUT *n* scalar, order of identity matrix.

OUTPUT *y* N×N sparse identity matrix.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

EXAMPLE
```
y = sparseEye(3);
d = denseSubmat(y,0,0);
```

```
      1.0000000  0.0000000  0.0000000
d =   0.0000000  1.0000000  0.0000000
      0.0000000  0.0000000  1.0000000
```

NOW USE **spEye**

sparseFD

PURPOSE Converts a dense matrix to a 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.

This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE
`sparse.src`

NOW USE
`denseToSp`

sparseFP

PURPOSE
Converts a packed matrix to a 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.

This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE `sparse.src`

NOW USE **`spCreate`**

sparseHConcat

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, the result of the concatenation.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE `sparse.src`

NOW USE ~ operator

sparseNZE

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*.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE `sparse.src`

NOW USE **spNumNZE**

sparseOnes

PURPOSE Generates a 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	*r*×*c* sparse matrix of ones.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE sparse.src

NOW USE **spOnes**

sparseRows

PURPOSE Returns the number of rows in a sparse matrix.

sparseScale

FORMAT *r* = **sparseRows**(*x*);

INPUT *x* M×N sparse matrix.

OUTPUT *r* scalar, number of rows in *x*.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE sparse.src

NOW USE **rows**

sparseScale

PURPOSE Scales sparse matrix.

FORMAT { *a r s* } = **sparseScale**(*x*);

INPUT *x* M×N sparse matrix.

OUTPUT *a* M×N scaled sparse matrix.

r M×1 vector, row scale factors.

s N×1 vector, column scale factors.

REMARKS **sparseScale** scales the elements of the matrix by powers of 10 so that they are all within (-10,10).

This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

EXAMPLE

```
x = { 25  -12   0,
       3    0 -11,
       8 -100   0 };

sx = sparseFD(x,.01);

{ sxs, r, s } = sparseScale(sx);

print "x";
print x;
print;
print "scaled matrix";
print denseSubmat(sxs,seqa(1,1,3),seqa(1,1,3));
print;
print "row factors";
print r;
print;
print "column factors";
print s;

x
   25.000000       -12.000000       0.00000000
   3.0000000       0.00000000       -11.000000
```

```
   8.0000000       -100.00000       0.00000000

scaled matrix
   2.5000000       -1.2000000       0.00000000
  0.30000000       0.00000000       -1.1000000
 0.080000000       -1.0000000       0.00000000

row factors
  0.10000000
  0.10000000
 0.010000000

column factors
   1.0000000
   1.0000000
   1.0000000
```

SOURCE sparse.src

NOW USE **spScale**

sparseSet

PURPOSE Resets sparse library global matrices to default values.

FORMAT **sparseSet;**

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of

functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

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

sparseSolve

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.

GLOBAL INPUT

_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 scalar, an estimate of the relative error in B. If zero, **_sparse_Btol** is assumed to be machine precision. Default = 0.

_sparse_CondLimit upper limit on condition of A. Iterations will be terminated if a computed estimate of the condition of A exceeds **_sparse_CondLimit**. If zero, set to 1 / machine precision.

_sparse_NumIters maximum number of iterations.

OUTPUT

x N×1 vector, solution of $Ax = B$.

GLOBAL OUTPUT

_sparse_RetCode scalar, termination condition.

- **0** x is the exact solution, no iterations performed.
- **1** solution is nearly exact with accuracy on the order of **_sparse_Atol** and **_sparse_Btol**.
- **2** solution is not exact and a least squares solution has been found with accuracy on the order of **_sparse_Atol**.
- **3** the estimate of the condition of A has exceeded **_sparse_CondLimit**. The system appears to be ill-conditioned.
- **4** solution is nearly exact with reasonable accuracy.
- **5** solution is not exact and a least squares solution has been found with reasonable accuracy.
- **6** iterations halted due to poor condition given machine precision.
- **7** **_sparse_NumIters** exceeded.

_sparse_Anorm scalar, estimate of Frobenius norm of

$$\begin{bmatrix} A \\ dI \end{bmatrix}$$

_sparse_Acond estimate of condition of A.

_sparse_Rnorm estimate of norm of

$$\begin{bmatrix} A \\ dI \end{bmatrix} x - \begin{bmatrix} B \\ 0 \end{bmatrix}$$

_sparse_ARnorm estimate of norm of

$$\begin{bmatrix} A \\ dI \end{bmatrix}' \begin{bmatrix} A \\ dI \end{bmatrix}$$

_sparse_XAnorm estimate of norm of *x*.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE sparse.src

NOW USE / operator

sparseSubmat

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.

This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These

old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE `sparse.src`

NOW USE **spSubmat**

sparseTD

PURPOSE Multiplies sparse matrix by dense matrix.

FORMAT *z* = **sparseTD**(*s*,*d*);

INPUT
s	M×N sparse matrix.
d	N×L dense matrix.

OUTPUT
z	M×L dense matrix, the result of *s***d*.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE `sparse.src`

NOW USE ***** operator

sparseTranspose

PURPOSE Transposes sparse matrix.

FORMAT *xt* **= sparseTranspose(***x***);**

INPUT *x* M×N sparse matrix.

OUTPUT *xt* M×N transposed sparse matrix.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE `sparse.src`

NOW USE ′ operator

sparseTrTD

PURPOSE Multiplies sparse matrix transposed by dense matrix.

sparseTScalar

FORMAT `z = sparseTrTD(s,d);`

INPUT
- *s* N×M sparse matrix.
- *d* N×L dense matrix.

OUTPUT
- *z* M×L dense matrix, the result of *s′d*.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE `sparse.src`

NOW USE **`spTrTDense`**

sparseTScalar

PURPOSE Multiplies scalar times selected elements of sparse matrix.

FORMAT `y= sparseTScalar(x,z,r,c);`

INPUT
- *x* M×N sparse matrix.
- *z* scalar, multiplicand
- *r* K×1 vector, row indices
- *c* K×1 vector, column indices

OUTPUT *y* M×N sparse matrix.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

EXAMPLE

```
x = { 1  2  0,
      4  0  6,
      7  8  0 };

sx = sparseFD(x,.01);
id = seqa(1,1,2);
sy = sparseTScalar(sx,3,id,id);

id1 = seqa(1,1,3);
print "x";
print x;
print "sparse times scalar";
print denseSubmat(sy,id1,id1);
```

```
x
   1.0000000        2.0000000       0.00000000
   4.0000000       0.00000000        6.0000000
   7.0000000        8.0000000       0.00000000

sparse times scalar
   3.0000000        6.0000000       0.00000000
   12.000000       0.00000000        6.0000000
   7.0000000        8.0000000       0.00000000
```

SOURCE sparse.src

NOW USE **spTScalar**

sparseVConcat

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, the result of the concatenation.

REMARKS This command was created to be used with the old sparse matrices in **GAUSS**, which were not a separate data type, but simply a special type of matrix. These old sparse matrices were not supported in any regular matrix functions or operators.

There is now a separate sparse matrix data type in **GAUSS**, which *is* supported in some regular matrix functions and operators, and there are a new set of functions for creating and manipulating these new sparse matrices. Please see the SPARSE MATRICES chapter in your GAUSS USER GUIDE for more information.

SOURCE sparse.src

NOW USE | operator

vartype

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.

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

NOW USE **vartypef**

Index

M

N

O

P

R

S

U

V

W

Z