GAUSS™

User Guide

Aptech Systems, Inc.— *Mathematical and Statistical System*

Part Number: 2369
Version 4.0
Revised January 25, 2002

Contents

Introduction 1

Product Overview

GAUSS™ is a complete analysis environment suitable for performing quick calculations, complex analysis of millions of data points, or anything in between. Whether you are new to computerized analysis or a seasoned programmer, the GAUSS family of products combine to offer you an easy to learn environment that is powerful and versatile enough for virtually any numerical task.

Since its introduction in 1984, GAUSS has been the standard for serious number crunching and complex modeling of large-scale data. Worldwide acceptance and use in government, industry, and the academic community is a firm testament to its power and versatility.

The GAUSS System can be described several ways: It is an exceptionally efficient number cruncher, a comprehensive programming language, and an interactive analysis environment. GAUSS may be the only numerical tool you will ever need.

Documentation Conventions

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

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

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

`monospace bold`	Refers to a GAUSS programming element within a narrative paragraph.	"...as explained under **`create`**..."

Getting Started 2

Installation Under UNIX/Linux

1. 1.Make a directory to install GAUSS in.
2. **`cd`** to that directory.
3. Unzip the `.gz` file if there is one.
4. Untar the `.tar` file.
5. Put the installation directory in the executable path.
6. Put the installation directory in the shared library search path.
7. Install the license, please refer to Chapter 5: Licensing for instructions.

For last minute information, see README.term.

Installation Under Windows

Machine Requirements

- A 486 computer or higher.

- Operating System and Memory (RAM) requirements

  Windows NT4.0, SP6 IE4.0, 32 MB minimum 256 MB recommended.

  Windows 2000, 64 MB minimum, 256 MB recommended.

  Windows XP, 128 MB minimum, 256 MB recommended.

- Free hard disk space requirements:

  Minimum of 100 MB free hard disk space, more may be needed depending on the size of matrices and the complexity of the program.

- Monthly defragmenting is recommended.

Installation from Download

Downloading

The production release is in `GAUSS_4.0_Win_heavy.zip`.

The GUI has online documentation in HTML format. There are also PDF versions of the Quick Start Guide and GAUSS manuals in this directory that you can download separately if you want them.

```
GAUSS_4.0_Manual.zip
```

Installation

Download `GAUSS_4.0_Win_heavy.zip`, unzip it in a temp directory, and run `setup.exe`.

If you are doing a command line ftp, once logged on you will be placed in the `GAUSS40` directory and will have access to all files contained in that directory and no others. Remember to set the transfer protocol to "bin" before downloading binary files. Use the "get" command to get the files you want off the ftp site.

To install the license, please refer to the Quick Start Guide.

Installation from CD

Insert the GAUSS 4.0 compact disc into the CD-ROM drive, and setup should start automatically. If setup does not start automatically, click Start, then click Run. Type `D:\setup.exe` in the dialog box (where D is the drive letter of the CD-ROM drive).

You can use this procedure for the initial installation of GAUSS, and for additions or modifications to GAUSS components.

To install the license, please refer to the Quick Start Guide.

Using the Command Line Interface 3

TGAUSS is the command line version of GAUSS. The executable file, `tgauss` is located in the GAUSS installation directory.

The format for using TGAUSS is:

`tgauss` *flag(s) program program...*

`-b`	Execute file in batch mode and then exit. You can execute multiple files by separating file names with spaces.
`-l` *logfile*	Set the name of the batch mode log file when using the **`-b`** argument. The default is `tmp/gauss###.log`, where `###` is the proccess ID.
`-e` *expression*	Executes a GAUSS expression. This command is not logged when GAUSS is in batch mode.
`-o`	Suppresses the sign-on banner (output only).
`-T`	Turns the dataloop translator on.
`-t`	Turns the dataloop translator off.

Viewing Graphics

GAUSS generates `.tkf` files for graphical output. The default output for graphics is `graphic.tkf`. Under Windows, you can use **`vwr`** to view the results of a file. Under Windows and UNIX, two functions are available to convert `.tkf` files to PostScript for printing and viewing with external viewers: the **`tkf2ps`** function will convert `.tkf` files to PostScript (`.ps`) files, and the **`tkf2eps`** function will convert `.tkf` files to encapsulated PostScript (`.eps`) files. For example, to convert the file `graphic.tkf` to a postscript file named `graphic.ps` use:

```
ret = tkf2ps("filename.tkf", "filename.ps")
```

If the function is successful it returns **`0`**.

Interactive Commands

quit

The **`quit`** command will exit TGAUSS.

The format for **`quit`** is:

```
quit
```

You can also use the **`system`** command to exit TGAUSS from either the command line or a program (see **`system`** in the GAUSS Language Reference).

The format for **`system`** is:

```
system
```

ed

The **`ed`** command will open an input file in an external text editor, see **`ed`** in the GAUSS Language Reference.

The format for **`ed`** is:

```
ed filename
```

browse

The **`browse`** command allows you to search for specific symbols in a file and open the file in the default editor. You can use wildcards to extend search capabilities of the browse command

The format for **`browse`** is:

```
browse symbol
```

config

The config command gives you access to the configuration menu allowing you to change the way GAUSS runs and compiles files.

The format for **`config`** is:

```
config
```

Run Menu

Translator	Toggles on/off the translation of a file using **`dataloop`**. The translator is not necessary for GAUSS program files not using **`dataloop`**.
Translator line number tracking	Toggles on/off execution time line number tracking of the original file before translation.
Line number tracking	Toggles on/off the execution time line number tracking. If the translator is on, the line numbers refer to the translated file.

Compile Menu

Autoload	Toggles on/off the autoloader.	
Autodelete	Toggles on/off autodelete.	
GAUSS Library	Toggles on/off the GAUSS library functions.	
User Library	Toggles on/off the user library functions.	
Declare Warnings	Toggles on/off the **`declare`** warning messages during compiling.	
Compiler Trace	Off	Turns off the compiler trace function.
	File	Traces program file openings and closings.
	Line	Traces compilation by line.
	Symbol	Creates a report of procedures and the local and global symbols they reference.

Debugging

The **debug** command runs a program under the source level debugger.

The format for **debug** is:

`debug` *filename*

General Functions

?	Displays a list of available commands.
q/Esc	Exits the debugger and return to the GAUSS command line.
+/-	Disables the last command repeat function.

Listing Functions

l *number*	Displays a specified number of lines of source code in the current file.
lc	Displays source code in the current file starting with the current line.
ll *file line*	Displays source code in the named file starting with the specified line.
ll *file*	Displays source code in the named file starting with the first line.
ll *line*	Displays source code starting with the specified line. File does not change.
ll	Displays the next page of source code.
lp	Displays the previous page of source code.

Execution Functions

s *number*	Executes the specified number of lines, stepping over procedures.
i *number*	Executes the specified number of lines, stepping into procedures.
x *number*	Executes code from the beginning of the program to the specified line count, or until a breakpoint is hit.
g ⟦ *args* ⟧	Executes from the current line to the end of the program, stopping at breakpoints. The optional arguments specify other stopping points. The syntax for each optional arguments is:

filename line cycle	The debugger will stop every *cycle* times it reaches the specified *line* in the named file.
filename line	The debugger will stop when it reaches the specified *line* in the named file.
filename ,, cycle	The debugger will stop every *cycle* times it reaches any line in the named file.
line cycle	The debugger will stop every *cycle* times it reaches the specified *line* in the current file.
filename	The debugger will stop at every line in the named file.
line	The debugger will stop when it reaches the specified *line* in the current file.
procedure cycle	The debugger will stop every *cycle* times it reaches the first line in a called procedure.

procedure	The debugger will stop every time it reaches the first line in a called procedure.

j ⟦*args*⟧	Executes code to a specified line, procedure, or cycle in the file without stopping at breakpoints. The optional arguments are the same as **g**, listed above.
jx *number*	Executes code to the execution count specified (*number)* without stopping at breakpoints.
o	Executes the remainder of the current procedure (or to a breakpoint) and stops at the next line in the calling procedure.

View Commands

v ⟦*vars*⟧	Searches for (a local variable, then a global variable) and displays the value of a specified variable.
v$ ⟦*vars*⟧	Searches for (a local variable, then a global variable) and displays the specified character matrix.

The display properties of matrices and string arrays can be set using the following commands.

r	Specifies the number of rows to be shown.
c	Specifies the number of columns to be shown.
number,*number*	Specifies the indices of the upper left corner of the block to be shown.
w	Specifies the width of the columns to be shown.
p	Specifies the precision shown.
f	Specifies the format of the numbers as decimal, scientific, or auto format.
q	Quits the matrix viewer.

Breakpoint Commands

lb — Shows all the breakpoints currently defined.

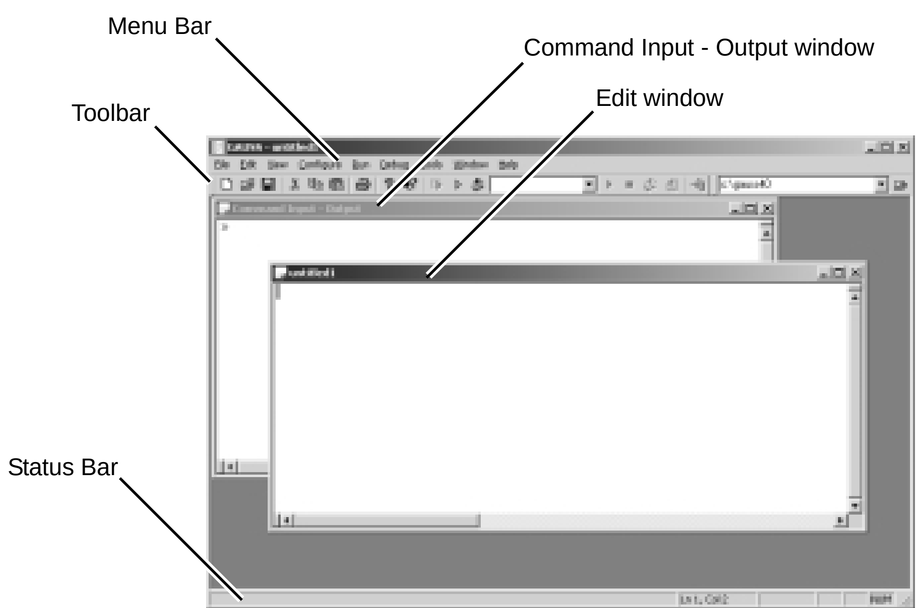
b ⟦ *args* ⟧ — Sets a breakpoint in the code. The syntax for each optional argument is:

filename line cycle	The debugger will stop every *cycle* times it reaches the specified *line* in the named file.
filename line	The debugger will stop when it reaches the specified *line* in the named file.
filename ,, cycle	The debugger will stop every *cycle* times it reaches any line in the named file.
line cycle	The debugger will stop every *cycle* times it reaches the specified *line* in the current file.
filename	The debugger will stop at every line in the named file.
line	The debugger will stop when it reaches the specified *line* in the current file.
procedure cycle	The debugger will stop every *cycle* times it reaches the first line in a called procedure.
procedure	The debugger will stop every time it reaches the first line in a called procedure.

d ⟦ *args* ⟧ — Removes a previously specified breakpoint. The optional arguments are the same arguments as **b**, listed above.

Introduction to the Windows Interface 4

The GAUSS graphical user interface is a multiple document interface. The interface consists of the Menu Bar, the Toolbar, edit windows, the Command Input - Output window, and the Status bar.



GAUSS Menus

You can view the commands on a menu by either clicking the menu name or pressing ALT+*n*, where *n* is the underlined letter in the menu name. For example, to display the File menu, you can either click File or press ALT+F.

File Menu

The File menu lets you access the file, printer setup, and exit commands. Some of these actions can also be executed from the toolbar. The File menu contains the following commands:

New	Opens a new, untitled document in an Edit window. *Note: New, unsaved documents are not automatically backed up until you save them, giving them a file name.*
Open	Opens an existing file for viewing or editing.
Save	Saves your changes to the file in the active window. If the file is untitled, you are prompted for a path and filename.
Save As	Saves your changes to the file in the active window using a new or different path or file name.
Close	Closes the document in the active window. You are prompted to save the file if it has been modified since you last saved it.
Close All	Closes all open files. You are prompted to save any file that has been modified since you last saved it.
Insert File	Opens an existing text file and copies the contents into the active document. This is similar to pasting text from the Windows clipboard.
Print	Prints the active file or selected text from the active window.
Print Preview	Shows a preview of your print job to view before printing.
Print Setup	Specifies the printer you want to use. Other printer options, such as page orientation and paper tray, are also accessed with this command.
Change Working Directory	Changes the directory where GAUSS looks for the files it uses for normal operation. This command does not affect the Open or Save As paths.

Clear Working Directory List	Clears the working directory list.
Exit	Closes all open files and exits GAUSS. You are prompted to save any file that has been modified since it was last saved.
Recent Files	GAUSS maintains a list of the ten most recent files you opened, at the end of the File menu. If the file you want to open is on this list, click it and GAUSS opens it in an Edit window.

Edit Menu

The Edit menu lets you access the set of editing commands. Some of these actions can also be executed from the toolbar. The Edit menu contains the following commands:

Undo	Restores your last changes in the active window.
Redo	Restores changes in the active window that you removed using the Undo Edit command.
Cut	Removes selected text from the active window and places it on the Windows clipboard.
Copy	Copies selected text from the active window to the Windows clipboard.
Paste	Copies text from the Windows clipboard to the active window at the cursor position.
Select All	Selects all text in the active window.
Find	Finds the specified text in the active window. The search starts at the cursor position and continues to the end of the text in the active window. The search can be case sensitive or case insensitive. You may also limit the search to regular expressions
Find Again	Resumes the search for the next occurrence of the text you specified in the previous Find action. Subsequent searches for the same text can also be performed by pressing F3.

Replace	Locates the specified text in the active window and replaces it with the text you entered in the "Replace with" field in the Search dialog box. The search starts at the cursor position and continues to the end of the text in the active window. The search can be case sensitive or case insensitive, and the replacement can be unique or global.
Insert Time/ Date	Inserts the current time and date at the cursor position. GAUSS uses the time and date that appears in the Microsoft Windows Date/Time Properties window.
Go To Line	Moves the cursor to the specified line number.
Go To Next Bookmark	Moves to the next bookmark in the program.
Toggle Bookmark	Sets or clears existing bookmarks from the program.
Edit Bookmarks	Opens the Edit Bookmarks window. From the Edit Bookmarks window you can add, remove, or go to any set bookmark in a program.
Record Macro	Places a series of keystrokes into memory so that they can be called at a later date. For more information about recording macros see "Using Keystroke Macros," page 5-2.

View Menu

The View menu lets you toggle the Main Toolbar, the Status Bar, the Working Directory Toolbar, or the Debug Toolbar on or off.

Main Toolbar	Toggles the Main toolbar on or off. For more information about the Main toolbar, see "Main Toolbar," page 4-9.
Status Bar	The Status Bar is located along the bottom of the GAUSS window. For more information about the status bar, see "Status Bar," page 4-12.
Working Directory Toolbar	Toggles the Working Directory toolbar on or off. For more information about the working directory toolbar, see "Working Directory Toolbar," page 4-11

Debug Toolbar	Toggles the Debug toolbar on or off. For more information about the Debug toolbar, see "Working Directory Toolbar," page 4-11.

Configure Menu

The Configure menu lets you customize the GAUSS environment.

Preferences	Opens the General Preferences window. From the General Preferences window you can define Run options, Compile options, DOS window options, and Autosave options. For more information on configuring GAUSS General Preferences, see "Preferences Dialog Box," page 5-7.
Editor Properties	Opens the Editor Properties window. From the Editor Properties window you can define colors and fonts, the language syntax, tabs, or general editor properties. For more information on configuring editor properties, see "Editor Properties," page 5-9.

Run Menu

The Run menu lets you run the code you have entered, a block of code you selected, or the active file, depending on the operating mode.

Insert GAUSS Prompt	Manually adds the GAUSS prompt at the cursor position. The GAUSS prompt (») is automatically displayed following the execution of GAUSS code.
Insert Last Cmd	Re-enters the last command written to the Input buffer.
Run Selected Text	Runs any text selected from the editor or the Command Input - Output window.
Run Active File	Runs the active file. The file then becomes the main file.
Test Compile Active File	Compiles the currently selected file. During compilation, any errors are displayed in the Output window. *Note: This command is different than the GAUSS Compile command, which compiles a program and saves the pseudocode as a file.*

Run Main File	Runs the file specified in the Main File list.
Compile Main File	Compiles the main file. During compilation, any errors are displayed in the Output window. *Note: This command is different than the GAUSS Compile command, which compiles a program and saves the pseudocode as a file.*
Edit Main File	Opens the specified main file in an edit window.
Stop Program	Stops the program currently running and returns control to the editor.
Build GCG File	Creates GAUSS pseudocode file that can be run over and over with no compile time.
Set Main File	Makes the active file the main file.
Clear Main File List	Removes all entries in the Main File list on the Main toolbar.
Translate Dataloop Cmds	Toggles translate dataloop command on and off. For more information see "Data Transformations," page 19-1.

Debug Menu

The Debug menu lets you access the commands used to debug your active file or main file.

The Debug menu contains the following Commands:

Debug Main File	Runs the main file in the debugger.
Debug Active File	Runs the active file in the debugger.
Set/Clear Breakpoint	Enables or disables a breakpoint at the cursor in the active file.
Edit Breakpoints	Opens a list of all breakpoints in your program. The breakpoints are listed by line number. Any procedure breakpoints are also listed.

Clear All Breakpoints	Removes all line and procedure breakpoints from the active file.
Go	Starts the debugger.
Stop	Stops the debugger.
Step Into	Runs the next executable line of code in the application and steps into procedures.
Step Over	Runs the next executable line of code in the application but does not step into procedures.
Step Out	Runs the remainder of the current procedure and stops at the next line in the calling procedure. Step Out returns if a breakpoint is encountered.
Set Watch	Opens the Matrix Editor for watching changing variable data. For more information about viewing variables see “Viewing Variables,” page 6-1.

Tools Menu

The Tools menu lets you open GAUSS tools windows. The following commands can be used:

Matrix Editor	Lets you create or edit data in a matrix (or grid). A cell can be edited by typing in a new value and pressing Enter. For more information see “Matrix Editor,” page 6-1.
Source Browser	Searches source files for string patterns. For more information see “GAUSS Source Browser,” page 8-1.
Library Tool	Lets you manage the contents of libraries. For more information see “Library Tool,” page 7-1.
DOS Compatibility Window	Runs programs that expect an 80x25 window that understands ANSI escape characters.

Window Menu

The Window menu commands let you manage your workspace. You can toggle the focus between all open windows using Ctrl+Tab, or clicking in the window you want active. All open windows are listed at the end of the Window menu. The following commands can be used:

Cmd Window	Makes the Command Input - Output window the active window.
Output Window	Splits the output from the Command Input - Output window.
Debug Window	Starts the debugger on the current file.
Dual Horizontal	Horizontally tiles the program source and execution windows within the main window, and minimizes all other windows.
Dual Vertical	Vertically tiles the program source and execution windows within the main window, and minimizes all other windows.
Cascade	Arranges all open windows on the screen, overlapping each, with the active window on top.
Tile Horizontal	Arranges all open windows horizontally on the screen without any overlap.
Tile Vertical	Arranges all open windows vertically on the screen without any overlap.
Arrange Icons	Arranges all minimized windows across the bottom of the main GAUSS window.
Split Horizontally	Splits the active window into two horizontal panes. This allows you to view two different areas of the same document to facilitate split-window editing. *Note: You can move the splitter bar by dragging it with the mouse. You can remove the splitter bar from the window by dragging it to the end of the window.*
Split Vertically	Splits the active window into two vertical panes. This allows you to view two different areas of the same document to facilitate split-window editing. *Note: You can move the splitter bar by dragging it with the mouse. You can remove the splitter bar from the window by dragging it to the end of the window.*
Open Window List	GAUSS maintains a list of all the windows you have opened at the end of the Window menu. If the window you want to view is on this list, click it and it becomes the active window.

Help Menu

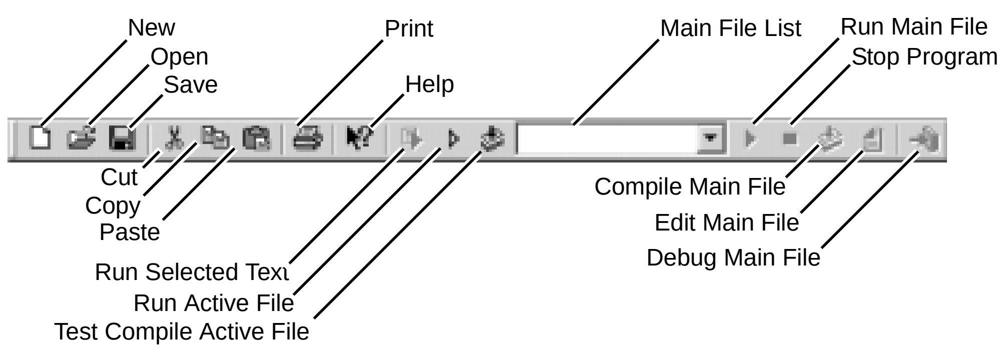
The Help menu lets you access information in the GAUSS Help system. The GAUSS Help menu contains the following Commands:

Help Topics	Starts the GAUSS Help system.
Contents	Starts the GAUSS Help system.
Keyboard	Accesses the list of keystrokes you can use for cursor movement, editing, and text selection.
GAUSS Reference	Accesses the online GAUSS Language Reference guide. The Guide contains the syntax for each GAUSS command.
About GAUSS	Provides information about your version of GAUSS, your license type and ID, as well as copyright information.

GAUSS Toolbars

The toolbar buttons let you have fast access to the most commonly used commands. Place the mouse pointer over the button to display a description of the command.

Main Toolbar

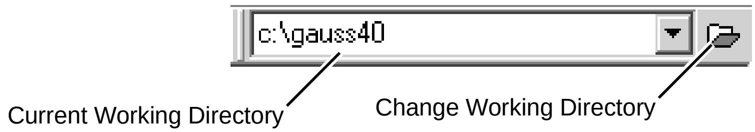


New	Opens a new, untitled document in an Edit window. *Note: New, unsaved documents are not automatically backed up until you save them, giving them a file name.*
Open	Opens an existing file for viewing or editing.

Save	Saves your changes to the file in the active window. If the file is untitled, you are prompted for a path and filename.
Cut	Removes selected text from the active window and places it on the Windows clipboard.
Copy	Copies selected text from the active window to the Windows clipboard.
Paste	Copies text from the Windows clipboard to the active window at the cursor position.
Print	Prints the active file or selected text from the active window.
Help	Accesses the GAUSS help system.
Run Selected Text	Runs any text selected from the editor or the Command Input - Output window.
Run Active File	Runs the active file. The file then becomes the main file.
Test Compile Active File	Compiles the currently selected file. During compilation, any errors are displayed in the Output window. *Note: This command is different than the GAUSS Compile command, which compiles a program and saves the pseudocode as a file.*
Main File List	Displays the name of the main file and lets you quickly change the main file to one of the files listed.
Run Main File	Runs the file specified in the Main File list.
Stop Program	Stops the program currently running and returns control to the editor.
Test Compile Main File	Compiles the main file. During compilation, any errors are displayed in the Output window. *Note: This command is different than the GAUSS Compile command, which compiles a program and saves the pseudocode as a file.*
Edit Main File	Opens the specified main file in an edit window.
Debug Main File	Runs the main file in the debugger.

Working Directory Toolbar

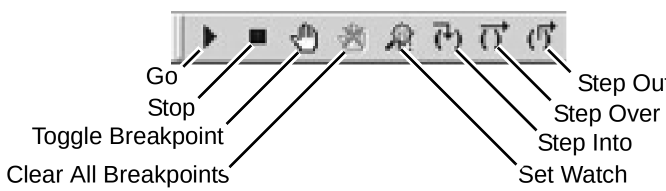
You can use the Working Directory toolbar to quickly change your working directory.



Current Working Directory List	Displays the name of the current working directory and lets you quickly change the working directory to one of the directories listed.
Change Working Directory	Browses to a new directory.

Debug Toolbar

You can use the Debug toolbar for quick access to commands while debugging a file.



Go	Starts the debugger.
Stop	Stops the debugger.
Toggle Breakpoint	Enables or disables a breakpoint at the cursor in the active file.
Clear All Breakpoints	Removes all line and procedure breakpoints from the active file.
Set Watch	Opens the Matrix Editor for watching changing variable data. For more information about viewing variables see "Viewing Variables," page 6-1.
Step Into	Runs the next executable line of code in the application and steps into procedures.

Step Over Runs the next executable line of code in the application but does not step into procedures.

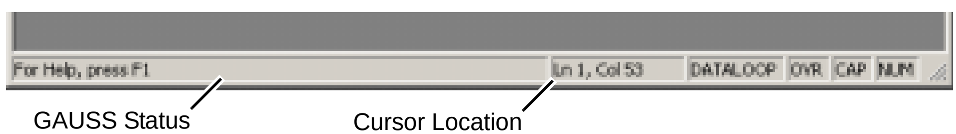
Step Out Runs the remainder of the current procedure and stops at the next line in the calling procedure. Step Out returns if a breakpoint is encountered.

Status Bar

The status bar is located along the bottom of the GAUSS window. The status of the windows and processes are shown on the status bar.

GAUSS Status

The first section of the status bar shows the current GAUSS status. From time to time you are alerted to the task GAUSS is performing by new messages appearing in the status bar.



Cursor Location The line number and column number where the cursor is located appear on the status bar for the active window. When a block of text is selected, the values indicate the first position of the selected text.

DATALOOP DATALOOP appears on the status bar to indicate the Dataloop Tranlator is turned on.

OVR OVR appears on the status bar when typing replaces the existing text with text you enter. When OVR does not appear on the status bar, typing inserts text without deleting the existing text. Press the Insert key to toggle between the two conditions.

CAP CAP appears on the status bar to indicate the Caps Lock key has been pressed and all text you enter will appear in upper case.

NUM NUM appears on the status bar to indicate the Num Lock key has been pressed and the keypad numbers are active.

Using the Windows Interface 5

The GAUSS graphical user interface is a multiple document interface. The interface consists of edit windows and the Command Input - Output window. Integrated into GAUSS is a full debugger with breakpoints and watch variables. The GAUSS graphical user interface also incorporates the Matrix Editor on page 6-1, The Library Tool on page 7-1, and Source Browser on page 8-1, as well as a context-sensitive HTML Help system on page 9-1.

Using the GAUSS Edit Windows

The GAUSS edit windows provide syntax color coding and auto-formatting as well as easy access to the Matrix Editor and Library Tool, and include an integrated context-sensitive help system accessible through the F1 key.

The edit windows provide standard text editing features like drag and drop text editing, and find and replace. The editor also lets you set bookmarks, define keystroke macros, find and replace using regular expressions, and run selected text from the editor.

Editing Programs

To begin editing, open an edit window by browsing to the source file, or by typing **`edit`** and the filename in the Command Input - Output window. If more than one file is open, the last file opened or run becomes the active window.

Using Bookmarks

Bookmarks are efficient placeholders used to identify particular sections or lines of code. To add or remove bookmarks, place the cursor in the line you want to bookmark and then press CTRL+F2, or click Toggle Bookmark on the Edit menu. You can jump to the next bookmark by pressing F2, or go to the previous bookmark by pressing SHIFT+F2.

To edit a list of all currently defined bookmarks, click Edit Bookmarks on the Edit menu. The Edit Bookmarks window allows you to add, remove, name or select the bookmark to which you wish to jump.

Changing the Editor Properties

You can customize the formatting of your code and text by changing font colors, fonts, adding line indentations, and adding line numbering to your programs. To access these properties, on the Configure menu click Editor Properties, or right-click on an edit window and click Properties on the context menu.For more information about the Editor Properties see “Editor Properties,” page 5-9.

Using Keystroke Macros

GAUSS will save up to 10 separate keystroke macros.

To record a keystroke macro, press CTRL+SHIFT+R, or click Record Macro on the Edit menu. When you start recording the macro, a stop button will appear in the GAUSS window.

You create a macro by clicking Record Macro and pressing the keystrokes you want recorded. Once you have completed recording the macro, you can stop recording with the stop button. Once you have finished recording the macro, you can select one of ten macro names for it.

Use the following guidelines when creating and using your macro:

- Only keystrokes in the active window are recorded, not keystrokes in a dialog box.
- Only keystrokes are recorded, not mouse movements.

Macros are not saved when you close GAUSS.

If your macro is lengthy, consider creating a separate file and copying the information from the file into the active window, rather than using a macro to enter the information.

Using Margin Functions

The margin of the edit window can be used to show currently set bookmarks, currently set breakpoints, and line numbers. You can also select an entire line of text with a single click in the Selection Margin.

You can turn on or off the margin in the Misc tab of the Editor Properties dialog box.

Editing with Split Views

Using split views, you can edit two parts of the same program in the same buffer. To open split views, click Split Horizontally or Split Vertically on the Window menu.

Finding and Replacing Text

Along with a standard find and replace function, you can use the edit window to find and replace regular expressions. To find regular expressions, open the Find dialog box and select the checkbox for regular expressions.

Running Selected Text

There are three ways you can run selected text. First, highlight the text you want to run, then either press CTRL+R, drag and drop the selected text into the Command Input - Output window, or click “Run Selected Text” on the Run menu.

Using The Command Input - Output Window

The Command Input - Output window lets you input interactive commands and view the results. The Command Input - Output window can be split into two separate windows, one for input and one for output, by clicking Output Window on the Window menu.

Output will be written at the insertion point in the Command Input - Output window or the Output window, when it is a separate window. GAUSS commands cannot be executed from this window.

From the Command Input - Output window, you can run saved programs. You can view or edit the data of any variable in the active workspace with the Matrix Editor. You can also open files for editing or to debug.

The GAUSS Command Input - Output window has many of the same features that the GAUSS text editor has. You can cut and paste text. You can search the buffer of the Command Input - Output window. You can also save the contents of the Command Input - Output window to a text file.

Running Commands

The GAUSS interface allows you to run programs that consist of single commands or blocks of commands executed interactively, as well as large-scale programs that may consist of commands in one or more files. The file that is run to execute the command is the main file (the file name displayed in the Main File list).

When you run commands interactively, the actual code being processed is called the “active block.” The active block is all code between the GAUSS prompt (») and the end of the current line. Thus, the active block can be one or more lines of code.

Interactive commands can be entered at the “»” prompt in the Command Input - Output window or selected using the mouse and clicking the Run Selected Text button on the Main toolbar.

A block of code can be executed by selecting the block with the mouse and then running that block using the Run Selected Text function.

Note: The GAUSS prompt (») at the beginning of the selected text is ignored.

You can enter multi-line commands into the Command Input - Output window by pressing CTRL+Enter at the end of each line. At the end of the final line in a multi-line command, press Enter. The Command Input - Output window will automatically place a semicolon at the end of a single-line command before it is interpreted. For multi-line commands, you must enter a semicolon at the end of each line.

You can also run multi-line commands by pasting the text of a file at the GAUSS prompt, or selecting multiple lines of code from the Command Input - Output window and pressing CTRL+R.

You can repeat any of the last 20 lines entered into the command buffer by pressing CTRL+L to cycle through the last command buffer.

Running Programs in Files

You can execute the active file by clicking Run Active File on the Run menu, or by clicking the Run Currently Active File button on the Main toolbar.

You can execute the file displayed in the Main File list (the main file) by clicking Run Main file on the Run menu, or by clicking the Run Main File button on the Main toolbar.

Using The Debugger

The debugger greatly simplifies program development. With all of the features of a dedicated debugging system, the debugger can help you to quickly identify and solve logic errors at run-time.

The debugger is integrated into the multiple document interface of GAUSS; it uses the interface tools, such as the edit windows, the Matrix Editor, and the Command Input - Output window for debugging. So while using the debugger, you still have all the features of the edit windows and Matrix Editor, along with GAUSS's suite of debugging tools.

You use the debugger to watch the program code as it runs. Prior to running the debugger, breakpoints and watch variables can be set to stop the program at points you set and provide additional data as the code is run.

Starting and Stopping the Debugger

You can start the debugger by clicking Go on the Debug menu or the Debug toolbar.

When starting the debugger, you can choose to debug the active file or to debug the main file of a program. If you are debugging a single file and already have the file open, you can use the menu or toolbar to start the debugger on the file, or simply type **debug** and the filename in the Command Input - Output window.

When you start the debugger, the debugger automatically highlights the first line of code to be run. Any breakpoints are shown in the left margin of the window.

You can stop the debugger at any time by clicking Stop on the Debug menu or the Debug toolbar.

Using Breakpoints

Breakpoints stop code execution where you have inserted them. Breakpoints are normally set prior to running the debugger, but can also be set or cleared during debugging by clicking the Set/Clear Breakpoint command on the Debug menu.

The debugger supports two types of breakpoints: procedure breakpoints and line number breakpoints. Procedure breakpoints pause execution when the specified procedure or function is reached. 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. The debugger also allows you to specify a certain cycle of execution for a line number or procedure where you want the execution to be paused. The cycle count is for the occurrence of the line number or procedure, not the number of times a line is to be skipped.

Setting and Clearing Breakpoints

You can set or clear a line breakpoint in the highlighted line of code by clicking Set/ Clear Breakpoint on the Debug menu or by pressing the F9 key.

To set breakpoints in any part of the file not currently being executed, just click the line where you want the breakpoint to be, then click Toggle Breakpoint.

To clear breakpoints in the file, click a line of code that has a breakpoint set and then click Set/Clear Breakpoint. You can also clear all breakpoints from the active file by clicking Clear All Breakpoints.

Using the Breakpoint Editor to Set and Clear Breakpoints

The Breakpoint Editor allows you to set or clear both line and procedure breakpoints. It also lets you specify cycles of execution for breakpoints. With the Breakpoint Editor, you can set or clear breakpoints in any program currently in your working directory.

Stepping Through a Program

GAUSS's debugger includes the ability to step into, step out of, and step over code during debugging.

Use Step Into to execute the line of code currently highlighted by the debugger.

Use Step Out to execute to the end of the current function without pause and return to the calling function.

Use Step Over to execute the line of code currently highlighted by the debugger without entering the functions that are called.

Viewing and Editing Variables

GAUSS allows you to view and edit the values of variables during debugging.

Viewing Variable Values During Debugging

Once the debugger is started, the editor window uses floatover variable windows for viewing variable data. Floatover variable windows give a quick view of the value a variable currently holds by simply moving your mouse over the variable name in the edit window.

The floatover variable window is only intended to give a quick view of the data, so it may not show all data held by the variable. If the variable data is incomplete, the floatover variable window will display an arrow to show that there is more data. If you need to view more data, open the Matrix Editor by highlighting the variable name and pressing CTRL+E.

Editing Variable Values During Debugging

The debugger integrates the Matrix Editor to edit values of loaded variables, or to use as a watch window to view the changing values of variables as you step through a program.

To edit a variable value, highlight the variable in the edit window, or the Command Input - Output window and then open the Matrix Editor. You can use the menu or toolbar to start the Matrix Editor, or simply type CTRL+E.

Making a Watch Window

You can make the Matrix Editor a Watch window, allowing you to watch the changing value of a variable as the lines of the program are executed. You can activate the Watch window by clicking Set Watch on the Debug menu, or by highlighting a variable name in the debugger window and pressing CTRL+E.

You use a Watch window to see how variables change in value during debugging. Watch variables can be specified prior to running the debugger or during a debugging session.

The debugger searches for a watch variable using the following order:

1. A local variable within a currently active procedure.
2. A global variable.

A watch variable can be the name of a matrix, a scalar, a string array, or a string. For a matrix or a string array, the first element is displayed. If a matrix element is clicked, the Matrix Editor is loaded with the matrix. The matrix elements can be changed during the debugging session.

Customizing GAUSS

Preferences Dialog Box

The Preferences dialog box lets you specify how GAUSS operates. To open the Preferences dialog box, click Preferences... on the Configure menu. The changes you make in the Preferences dialog box remain set between sessions.

Run Options

Dataloop Translator	Specifies whether GAUSS will translate data loops into procedures.
Translate Line Number Tracking	Specifies whether GAUSS will preserve the line numbers of data loops after being translated to procedures.
Line Number Tracking	Specifies whether GAUSS will preserve line numbers of a file being compiled for the interpreter.

Sound at End of Job Determines whether or not a sound is played at the end of the execution of GAUSS code. The sound can be selected using the Select button and played using the Test button.
The default is OFF.

Compile Options

The Compile tab contains options that let you control how GAUSS compiles a program before it is run.

Autoload Specifies whether the autoloader will automatically resolve references in your code. If Autoload is off, you must define all symbols used in your program.

Autodelete Use Autodelete in conjunction with Autoload to control the handling of references to unknown symbols.

GAUSS Library Specifies whether the autoloader will use the standard GAUSS library in compiling your code.

User Library Specifies whether the autoloader will use the User Libraries in compiling your code.

Declare Warnings Specifies whether the GAUSS compiler will display declare warnings in the Command Input - Output window. For more information on declare warnings see “Using dec Files,” page 14-12.

Compiler Trace Specifies whether you would like to trace the file compilation by file opening and closing, specific lines, or whether you would like to trace by local and global symbols.

Files

The Files tab contains options that let you control how GAUSS auto-saves your work

Autosave on Execute Specifies whether open files will automatically be saved when a file is run. If the file you are running is loaded, it will be saved prior to execution, regardless of how it is executed (Run file, command line, main file, or active file). All open editor files, including the active file, are saved before execution.
Note: New, unsaved documents are not automatically backed up until you save them, giving them a file name. After you save the new file, it will be automatically backed up with all other open files.

Autosave	Specifies whether you want GAUSS to automatically save your files at a set interval of time.

DOS Window

The DOS Window tab lets you control the fonts used in the DOS window.

Font Options	Specifies what font the DOS window will use.

Editor Properties

You can customize the formatting of your code and text by changing font colors, fonts, adding line indentations, and adding line numbering to your programs. To access these properties, on the Configure menu click Editor Properties.

Color/Font

Color	Specifies the way syntax coloring works in the editor.
Font	Specifies what font the edit window will use.

Language/Tabs

Auto Indentation Style	Specifies how the autoindenter will indent your code.
Tabs	Specifies how many spaces a tab has.
Language	Specifies what syntax the GAUSS editor will recognize for syntax coloring.
Fixup Text Case While Typing Language Keywords	Specifies whether the editor will automatically change the case of GAUSS keywords when they use the wrong case.

Misc

Smooth Scrolling	Enables or disables smooth scrolling when the window is scrolled up/down by one line or left/right by one character.
Show Left Margin	Enables or disables the editor’s margin. The margin is used for showing breakpoints, bookmarks, or line numbers.

Line Tooltips on Scroll	Shows the first line number on screen as a tooltip as you scroll up and down the file.
Allow Drag and Drop	Enables or disables drag and drop functionality.
Allow Column Selection	Lets you select and manipulate columns of text.
Confine Caret to Text	Tells the GAUSS editor to interpret carets as text only rather than as substitution symbols or text.
Color Syntax Highlighting	Toggles on or off color syntax highlighting.
Show Horizontal Scrollbar	Toggles on or off the horizontal scrollbar.
Show Vertical Scrollbar	Toggles on or off the vertical scrollbar.
Allow Horizontal Splitting	Toggles on or off the ability to split editor panes horizontally.
Allow Vertical Splitting	Toggles on or off the ability to split editor panes vertically.
Line Numbering	Specifies the style and starting digit for line numbering.
Max Undoable Actions	Sets the number of actions that you can undo.

Using GAUSS Keyboard Assignments

Cursor Movement Keys

UP ARROW	Up one line
DOWN ARROW	Down one line

LEFT ARROW	Left one character
RIGHT ARROW	Right one character
CTRL+LEFT ARROW	Left one word
CTRL+RIGHT ARROW	Right one word
HOME	Beginning of line
END	End of line
PAGE UP	Next screen up
PAGE DOWN	Next screen down
CTRL+PAGE UP	Scroll window right
CTRL+PAGE DOWN	Scroll window left
CTRL+HOME	Beginning of document
CTRL+END	End of document

Edit Keys

BACKSPACE	Delete character to left of cursor, or delete selected text
DEL	Delete character to right of cursor, or delete selected text
CTRL+INS or CTRL+C	Copy selected text to Windows clipboard
SHIFT+DEL or CTRL+X	Delete selected text and place it onto Windows clipboard
SHIFT+INS or CTRL+V	Paste text from Windows clipboard at the cursor position
CTRL+Z	Undo last editing action

Text Selection Keys

SHIFT+UP ARROW	Select one line of text up
SHIFT+DOWN ARROW	Select one line of text down
SHIFT+LEFT ARROW	Select one character to the left
SHIFT+RIGHT ARROW	Select one character to the right

SHIFT+CTRL+LEFT ARROW	Select one word to the left
SHIFT+CTRL+RIGHT ARROW	Select one word to the right
SHIFT+HOME	Select to beginning of the line
SHIFT+END	Select to end of the line
SHIFT+PAGE UP	Select up one screen
SHIFT+PAGE DOWN	Select down one screen
SHIFT+CTRL+HOME	Select text to beginning of document
SHIFT+CTRL+END	Select text to end of document

Command Keys

CTRL+A	Redo
CTRL+C	Copy selection to Windows clipboard
CTRL+D	Open Debug window
CTRL+E	Open Matrix Editor
CTRL+F	Find/Replace text
CTRL+G	Go to specified line number
CTRL+I	Insert GAUSS prompt
CTRL+L	Insert last
CTRL+N	Make next window active
CTRL+O	Open Output window
CTRL+P	Print current window, or selected text
CTRL+Q	Exit GAUSS
CTRL+R	Run selected text
CTRL+S	Save window to file
CTRL+W	Open Command window
CTRL+V	Paste contents of Windows clipboard
CTRL+X	Cut selection to Windows clipboard

CTRL+Z	Undo

Function Keys

F1	Open GAUSS Help system or context-sensitive Help
F2	Go to next bookmark
F3	Find again
F4	Go to next search item in Source Browser
F5	Run Main File
F6	Run Active File
F7	Edit Main File
F8	Step Into
F9	Set/Clear breakpoint
F10	Step Over
ALT+F4	Exit GAUSS
ALT+F5	Debug Main File
CTRL+F1	Searches the active libraries for the source code of a function.
CTRL+F2	Toggle bookmark
CTRL+F4	Close active window
CTRL+F5	Compile Main File
CTRL+F6	Compile Active File
CTRL+F10	Step Out
ESC	Unmark marked text

Menu Keys

ALT+C	Configure menu
ALT+D	Debug menu
ALT+E	Edit menu

ALT+F	File menu
ALT+H	Help menu
ALT+R	Run menu
ALT+T	Tools menu
ALT+W	Window menu
ALT+V	View menu

Matrix Editor 6

Using the Matrix Editor

The Matrix Editor lets you view and edit matrix data in your current workspace. You can open the Matrix Editor from either the Command Input - Output window or a GAUSS edit window by highlighting a matrix variable name and typing Ctrl+E. You can view multiple matrices at the same time by opening more than one Matrix Editor.

Editing Matrices

The Matrix Editor will allow you to format matrices in decimal, scientific, Hexadecimal, or as text characters.

Just like a spreadsheet, when using the Matrix Editor, you can use your keyboard's arrow keys to quickly move between matrix positions. To edit a scalar value, select a cell and press Enter. You can use the Home and End keys to move to the beginning or end of a scalar. When finished editing, press Enter again.

Viewing Variables

All variables are treated as matrices in GAUSS. A scalar is simply a 1x1 matrix. A vector is a (Nx1) or (1xN) matrix. So you can use the Matrix Editor to view and monitor the value of any variable. You can update the value of a variable at any time by using the Reload function. When using the Matrix Editor to view, edit or monitor smaller matrices, you can minimize space it occupies on the screen by selecting Minimal View from the View menu.

By using the Auto-reload function, GAUSS will automatically update the values of variables in the Matrix Editor. Using Auto-reload you can create a watch window.

Setting Watch Variables

Watch Variables allow you to see how variables change in value while debugging a program. A watch variable can be the name of a matrix, a scalar, a string array, or a string.

The debugger searches for a watch variable in the following order:

- a local variable within a currently active procedure
- a global variable

Matrix Editor Menu Bar

Matrix Menu

The Matrix menu lets you control the data of the Matrix in the Matrix Editor as an entire set.

Load	Clears any existing grid and loads any named matrix from the GAUSS workspace to the grid.
Reload	Reloads the existing matrix with the name shown on the Title bar.
Auto-Reload	Automatically updates the data shown in the Matrix Editor, creating a watch window.
Save	Saves the grid as a matrix in the GAUSS workspace. If a matrix of the same name already exists in the workspace, it is overwritten.

Format Menu

The Format menu lets you control the way the data is presented in the Matrix Editor.

Edit Menu

The Edit menu gives you tools to control the data in the Matrix Editor.

Clear All	Clears the grid of all values but keep the row and column order.

Preferences	Sets several matrix options, including the number of digits to the right of the decimal point, cell height and width, and whether pressing the Enter key moves the cursor down or over one cell. These options, along with screen position and window state, are saved between sessions.

View Menu

The View menu lets you control the Matrix Editor window. The View menu also lets you control your view of imaginary numbers.

Real Parts	Specifies that you want the real parts of imaginary numbers to be displayed in the Matrix Editor.
Imaginary Parts	Specifies that you want the imaginary parts of numbers to be displayed in the Matrix Editor.
Minimal View	Minimizes the amount of screen space occupied by the Matrix Editor. This is especially useful for creating watch windows for single variables.
Stay on Top	Forces the Matrix Editor window to remain visible on the screen even when the interface focus has shifted to another window.

Library Tool 7

Using the Library Tool

The Library Tool lets you quickly manage your libraries. You can add and remove libraries and you can add and remove files within the libraries.

Managing Libraries

Using the New Library button, you can create a new library for organizing your code. You can remove a library by selecting the Delete Library button.

Managing the Library Index

To add absolute path names to the library index, use the Add Paths button. To only use file names for searching libraries, use the Strip Paths button. Use Rebuild to recompile all the files used in the library, and rebuild the library index file. Use the Revert to Original button to revert to the configuration the library was in when the Library Tool was opened.

Managing Library Files

You can add files to a library with the Add button. You can remove files from a library with the Remove button. After changing source files referred to in a library, select the files in the file list and update the library index with the Update button. To remove multiple files from a library, select the files in the file selection window, and use the Clear Selection button.

For more information about libraries, see “Libraries,” page 14-1.

GAUSS Source Browser 8

The GAUSS Source Browser lets users quickly find, view, and if necessary, modify the source code for external procedures in active libraries.

In GAUSS for Windows, put the cursor on a global symbol name and press Ctrl-F1. The Source Browser will open an Edit window containing the source file holding the symbol definition. The browser searches the active library files (**`.lcg`** files) for the symbol name.

If **`**> symbolname`** is found in the file at the beginning of a line, that line will be displayed at the top of the window. Users may make use of these features in their own code.

In TGAUSS, type **`browse`** followed by a command name or string containing wildcards. The source code for the command will be opened in the default editor. Wildcards may also be used. You select the desired command from the list by number.

GAUSS Help 9

Context-Sensitive Help

GAUSS integrates a context-sensitive Help system to help you use the GAUSS environment and the GAUSS language. Context-sensitive means you can get help on the GAUSS interface or the GAUSS language without having to go through the Help menu. For example, to get Help on any keyword in the GAUSS language, place the insertion point on that keyword in a GAUSS edit window or the Command Input - Output window and press F1.

You can press F1 from any context-sensitive part of the GAUSS interface to display Help information about that part. The context-sensitive parts are:

- GAUSS windows
- Toolbar buttons
- GAUSS menus
- The GAUSS language

For all intrinsic commands and functions, the GAUSS Command Reference for the command is displayed. For all other external procedures in active libraries, an HTML window is opened containing a source code file. You may then scroll through the file to find the desired symbol.

CTRL+F1 Support

You can search through all active libraries for any global symbol by placing the cursor on the name of the symbol and pressing CTRL+F1.

GAUSS searches through all active libraries for the file that the symbol is defined in. If found, the file containing the source code is opened in an edit window. If the file contains **`**>`** *symbol_name* at the beginning of a line of commented code, the cursor will be placed at the beginning of that line. If not found, the cursor will be placed at the beginning of the file.

To properly implement this functionality in your own source code, place

`**>` *symbol_name*

at the beginning of a line in a comment block.

ToolTips

A ToolTip is a small label that is displayed when the mouse pointer is held over a GAUSS button. The ToolTip will give a brief description of the button's function.

SHIFT+F1 Help Support

If you press SHIFT+F1 or click on the Help toolbar button, GAUSS goes into Help mode and the pointer changes to a Help pointer (arrow + ?). The next thing you click with the Help pointer opens in Windows Help.

Help Menu

From the Help menu, you can directly access either the online Language Reference Manual or the online User Guide. The Help menu also gives quick access to GAUSS default keyboard mappings.

Other Help

GAUSS also includes a full online version of the GAUSS Language Reference and GAUSS User Guide in PDF format. These manuals are located on the GAUSS CD-ROM.

The Gaussians mail list is an e-mail list providing users of GAUSS an easy way to reach other GAUSS users. Gaussians provides a forum for information exchange, tips and experiences using GAUSS. For more information about the Gaussians mail list, see *http://www.aptech.com/s2_gaussians.html*. You can also e-mail *support@aptech.com*.

Language Fundamentals 10

GAUSS is a compiled language. GAUSS is also an interpreter. A compiled language, because GAUSS scans the entire program once and translates it into a binary code before it starts to execute the program. 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.

How can GAUSS be so fast if it is an interpreter? Two reasons. First, GAUSS has a fast interpreter, and the binary compiled code is compact and efficient. Second, and most significantly, 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.

This chapter will enable you to understand the distinction between "compile time" and "execution time," two very different stages in the life of a GAUSS program.

Expressions

An expression is a matrix, string, constant, function reference, 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 '**=**'.

Statements

A statement is a complete expression or a 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;
```

or

```
x*3;
```

Here is an example of a statement that is a command rather than an expression:

```
output on;
```

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.

Executable Statements

Executable statements are statements that can be "executed" over and over during the execution phase of a GAUSS program (execution time). 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 the 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 };
```

Nonexecutable Statements

Nonexecutable statements are statements that have an effect only when the program is compiled (compile time). They generate no executable code at the current location of the instruction pointer.

Here are two 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);
```

There are two executable statements in the example above: the first line and the last line. In the binary code that is generated, the last line will follow immediately after the first line. 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.

Programs

A program is any set of statements that are run together at one time. There are two sections within a program.

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 must always be a main section even if it consists only of a call to the one and only procedure 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.

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.

#include statements are allowed 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;
```

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 earlier 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.

#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
```

and then write your program using **hpage** and **vpage**. GAUSS will replace them with **9.0** and **6.855** when it compiles the program. This makes a program 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.

For more information on line number tracking, see "Debugging," page 15-2 and see "Debugging Data Loops," page 19-2. See also **#lineson** in the *GAUSS Language Reference.*

The syntax for **#srcfile** and **#srcline** is different than for the other directives that take arguments. Typically, directives do not take arguments in parentheses; that is, 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(c:\gauss\test.e)
```

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 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.

For details on defining and calling procedures, see “Procedures and Keywords,” page 12-1.

Data Types

There are two basic data types in GAUSS: matrices and strings. It is not necessary to declare the type of a variable, but it is good programming practice to respect the types of variables whenever possible. The data type and size can change in the course of a program.

The **`declare`** statement, used for compile-time initialization, enforces type checking.

Short strings of up to 8 bytes can be entered into elements of matrices, to form character matrices. (For details, see “Character Matrices,” page 10-19.)

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
```

These will be stored as double precision (15-16 significant digits). The range is the same as for matrices. (For details, see “Matrices,” page 10-8.)

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
```

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] [1,2] [1,3] [2,1] [2,2] [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	$4.19 \times 10^{-307} \leq \lvert X \rvert \leq 1.67 \times 10^{+308}$

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 two 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 matrix or data set 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.

Following are some examples of matrix definition statements.

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. The statement

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

or

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

will result in

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

The statement

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

will result in

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

The statement

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

will result in

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

The statement

```
let x[3,3];
```

will result in

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

The statement

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

will result in

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

Complex constants can be entered in a **`let`** statement. In the following example, the + or - is not a mathematical operator, but connects the two parts of a complex number. There should 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’. The statement

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

will result in

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

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 and braces:

```
x = {};
```

Empty matrices are currently supported only by the **rows** and **cols** functions and the concatenation operators (**~** and **|**):

```
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.

The **~** is the horizontal concatenation operator and the **|** is the vertical concatenation operator. The statement

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

will be evaluated as

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

and will result in a 2x2 matrix because horizontal concatenation has precedence over vertical concatenation:

```
1 2
3 4
```

The statement

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

will be evaluated as

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

and will result in a 2x2 matrix because the arithmetic operators have precedence over concatenation:

```
2 4
1 3
```

For more information, see “Operator Precedence,” page 10-22.

The **let** command is used to initialize matrices with constant values:

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

Unlike the concatenation operators, it cannot be used to define matrices in terms of expressions such as

```
y = x1-x2~x2|x3*3~x4;
```

The statement

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

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

The statement

```
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).

The statement

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

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

The statement

```
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.

The statement

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

will cause a ? to be printed in the window, and will prompt the user until six numbers have been entered from the keyboard.

The statement

```
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.)

The statement

```
load x;
```

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

The statement

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

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

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:

`$+`	Combine two strings into one long string.
`^`	Interpret following name as a variable, not a literal.
`chrs`	Convert vector of ASCII codes to character string.
`dttostr`	Converts a matrix containing dates in DT scalar format to a string array.
`ftocv`	Character representation of numbers in NxK matrix.
`ftos`	Character representation of numbers in 1x1 matrix.
`ftostrC`	Converts a matrix to a string array using a C language format specification.
`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.

`strsplit` splits an Nx1 string vector to an NxK string array of the individual tokens.

`strsplitPad` Splits a string vector into a string array of the individual tokens. Pads on the right with the null strings.

`strtodt` Converts a string array of dates to a matrix in DT scalar format.

`strtof` converts a string array to a numeric matrix.

`strtofcplx` converts a string array to complex numeric matrix.

`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:

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

or

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

The backslash is used as the escape character inside double quotes to enter special characters:

"\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 character elements of matrices is with the substitution operator (**^**) .

In the command

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

by default, GAUSS will interpret the **olsdat** as a literal; that is, the literal name of the GAUSS data file 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 substitution 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, use

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

The substitution operator (**^**) works with **load** and **save**, also:

```
lpath="c:\gauss\procs";
```

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

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;
save ^name=x;
save path=^spath;
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	Initialize variables at compile time.
delete	Delete specified global symbols.

fgetsa	Read multiple lines of text from a file.
fgetsat	Read multiple lines of text from a file, discarding newlines.
format	Define output format for matrices, string arrays, and strings.
fputs	Write strings to a file.
fputst	Write strings to a file, appending newlines.
let	Initialize matrices, strings, and string arrays.
loads	Load a string or string array file (`.fst` file).
lprint	Print expressions to the printer.
lshow	Print global symbol table to the printer.
print	Print expressions in window and/or auxiliary output.
reshape	Reshape a matrix or string array to new dimensions.
save	Save matrix, string array, string, procedure, function, or keyword to disk and give the disk file either a `.fmt`, `.fst`, or `.fcg` extension.
show	Display global symbol table.
sortcc	Quick-sort rows of matrix or string array based on character column.
type	Indicate whether variable passed as argument is matrix, string, or string array.
typecv	Indicate whether variables named in argument are strings, string arrays, matrices, procedures, functions, or keywords.
varget	Access the global variable named by a string array.
varput	Assign the global variable named by a string array.
vec	Stack columns of a matrix or string array to form a column vector.
vecr	Stack 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

Character Matrices

Matrices can have either numeric or character elements. For convenience, a matrix containing character elements is referred to as a character matrix.

A character matrix is not a separate data type, but 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 entire 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 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.

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, also. 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 you to specify which type of data it contains when printing the contents of the matrix. (For details, see **`print`** and **`printfm`** in the *GAUSS Language Reference.*)

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.

Date and Time Formats

DT Scalar Format

The DT scalar format is a double precision representation of the date and time. In the DT scalar format, the number

```
20010421183207
```

represents 18:32:07 or 6:32:07 PM on April 21, 2001.

DTV Vector Format

The DTV vector is a `1x8` vector. The format for the DTV vector is:

[1]	Year
[2]	Month, 1-12
[3]	Day of month, 1-31
[4]	Hour of day, 0-23
[5]	Minute of hour, 0-59
[6]	Second of minute, 0-59
[7]	Day of week, 0-6 where 0 is Sunday
[8]	Day since beginning of year, 0-365

UTC Scalar Format

The UTC scalar format is the number of seconds since January 1, 1970, Greenwich Mean Time.

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 meaningful results.

NaN

There are many floating point encodings that 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. Subsequent calculations will not deal well with an infinity; it usually signals an error in your program. The result of an operation involving an infinity is most often 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.

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 earlier. 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 this 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;
```

The 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.

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 that contain these 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. (For a listing of the precedence of all operators, see "Operator Precedence," page 11-18.)

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 as

$(2^3)^7$

In the expression

```
f1(x)*f2(y)
```

`f1` is evaluated before **`f2`**.

Here are some 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!`	$2*(3!)$

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.

Looping

do loop

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

```
do while scalar_expression; /* loop if expression is true */
    .
    .
    statements
    .
    .
endo;
```

also

```
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;
```

This will print:

```
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 *GAUSS Language Reference*.)

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*, the first list of statements is executed. If it is *FALSE*, 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*, 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.

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 good 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 procedures. The variables 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 the subroutine is in. (For details, see **gosub** in the *GAUSS Language Reference*.)

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);
```

Rules of Syntax

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

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 with one exception: 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:

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

Case

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

Comments

```
/*  this kind of comment can be nested  */
@  this kind of comment cannot be nested  @
```

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);
```

Symbol Names

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

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;
```

Assignment Statements

The assignment operator is the equal sign '**=**' :

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

Multiple assignments must be enclosed in braces '**{ }**':

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

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

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

Function Arguments

The arguments to functions are enclosed in parentheses '**( )**':

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

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 one or two 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, all rows will be used; if **cv** is a scalar 0, all columns will be used. If a vector is used in an index expression, it is illegal to use the space operator or the colon operator on the same side of the comma as the vector.

Arrays of Matrices and Strings

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

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**:

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

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 *ith* 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 3rd 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 3rd matrix listed in **mvec** and return a 1 if successful or a 0 if it fails.

Arrays of Procedures

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

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 *ith* procedure, evaluated at **x**:

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

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

fi(x,2) will return **f2(x)**. The ampersand is used to return the pointers to the procedures. **nms** is a numeric vector that contains a set of pointers. The **local** statement is used twice. The first tells the compiler that **f** is a local matrix. The *ith* pointer, which is just a number, is assigned to **f**. The second **local** statement tells the compiler to treat **f** as a procedure from this point on; thus the subsequent statement **f(x)** is interpreted as a procedure call.

Operators 11

Element-by-Element Operators

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

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, 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*:

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

$$y = 2$$

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

- If *x* is an NxI column vector and *y* is an NxK matrix, or vice versa, the vector is swept “across” the matrix:

vector		matrix		
1	→	2	4	3
4	→	3	1	4
3	→	6	1	2

result		
3	5	4
7	5	8
9	4	5

- If *x* is a 1xK column vector and *y* is an NxK matrix, or vice versa, 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, the matrices are not conformable to these operations and an error message will be generated.

Matrix Operators

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

Numeric Operators

For details on how matrix conformability is defined for element-by-element operators, see “Element-by-Element Operators,” page 11-1.

+ 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.

Linear equation solution is performed in the following cases:

$Ax = b$

- If *A* is a square matrix and has the same number of rows as *b*, 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*, 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.

% 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* is noninteger, it 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), the "outer product" or "table" of the two will be computed. (For conformability rules, see "Element-by-Element Operators," page 11-1.)

./ 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;
```

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

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

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

***~** Horizontal direct product

z **= *x* *~ *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.

Other Matrix Operators

' Transpose operator

y **= *x*';**

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|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~y;
```

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

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

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

Relational Operators

For details on how matrix conformability is defined for element-by-element operators, see “Element-by-Element Operators,” page 11-1.

Each of these operators has two equivalent representations. Either can be used (for example, **<** 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 '**.**', 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: "if every element of x is 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 '**.**', 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***;**

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 tables:

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*;

Other Operators

Assignment Operator

Assignments are done with one equal sign:

```
y = 3;
```

Comma

Commas are used to delimit lists:

```
clear x,y,z;
```

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);
```

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. (For more information, see “Indexing Procedures,” page 12-9.)

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. Here are some 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 use the same *k* to create *z* as follows:

```
z = “dog” $+ k[1,1];
```

the result 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 such as

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

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

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;
```

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 currently 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;

save ^name = x;

save path = ^spath;

run ^prog;

msym ^mstring;
```

Using Dot Operators with Constants

When you use those operators preceded by a '`.`' (dot operators) with a scalar integer constant, insert a space between the constant and any following dot operator. Otherwise, the dot will be interpreted as part of the scalar; that is, the decimal point. 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;
```

and not as

```
x = 2   .< y;
```

Be careful when using the dot relational operators (**.<**, **.**≤, **.==**, **./=**, **.>**, **.**≥). The same problem can occur with other dot operators, also. 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;

and not as

y = (x ./ 2) ./ x;

The second division, then, is handled as a matrix division rather than an element-by-element division.

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 that 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	Operator	Precedence
.'	90	**.$≥**	65	**$≥**	55
'	90	**./=**	65	**/=**	55
!	89	**.<**	65	**<**	55
.^	85	**.≤**	65	**≤**	55
^	85	**.==**	65	**==**	55
(unary -)	83	**.>**	65	**>**	55
*	80	**.≥**	65	**≥**	55
*~	80	**.eq**	65	**eq**	55
.*	80	**.ge**	65	**ge**	55
.*.	80	**.gt**	65	**gt**	55
./	80	**.le**	65	**le**	55
/	80	**.lt**	65	**lt**	55
%	75	**.ne**	65	**ne**	55
$+	70	**.not**	64	**not**	49
+	70	**.and**	63	**and**	48
-	70	**.or**	62	**or**	47
~	68	**.xor**	61	**xor**	46
|	67	**.eqv**	60	**eqv**	45
.$/=	65	**$/=**	55	**(space)**	35
.$<	65	**$<**	55	**:**	35
.$≤	65	**$≤**	55	**=**	10
.$==	65	**$==**	55		
.$>	65	**$>**	55		

Procedures and Keywords 12

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 keep the details of their operation separate from the other parts of the program. 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. A procedure can be as small and simple or as large and complicated as necessary to perform a particular task. Procedures allow you to build on your 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 in the global symbol table 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.

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:

```
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. For example:

```
{ b,sd,t } = regress(x,y);
```

Following is a discussion of the five parts of a procedure definition.

Procedure Declaration

The **proc** statement is the procedure declaration statement. The format is:

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 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 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.

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. (For more information, see “Passing Procedures to Procedures,” page 12-8.)

Variables that are global to the system (that is, variables listed in the global symbol table that can be shown with the **show** command) can be accessed by any procedure without any redundant declaration inside the procedure. If you want to create variables 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 the procedure will be to the local rather than to a global having the same name. (See **clearg**, **varget**, and **varput** in the *GAUSS Language Reference* 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 message.

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, there are 30 active locals whenever **`cat`** is called.

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`**. 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 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. (For more information, see “Indexing Procedures,” page 12-9.)

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`** error message or a **`Procedure calls too deep`** error message, depending on the space necessary to store the locals for each separate invocation of the procedure.

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, the **`retp`** statement is optional. The **`endp`** statement that 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.

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, executing this implicit return will result in a **Wrong number of returns** error message and the program will terminate.

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, 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.

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.

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 — 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 the keyword for the argument that is passed to the keyword when it is called. There is always one 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. (For more information, see "Defining a Procedure," page 12-2.) Keywords always return nothing. Any **retp** statements, if used, should be empty. For example:

```
keyword add(s)
   local tok, sum;

   if s $== “”;
      print “The argument is a null string”;
      retp;
   endif;
```

```
    print "The argument is: '"s"'";

    sum = 0;
    do until s $== "";
       { tok, s } = token(s);
       sum = sum + stof(tok);
    endo;
    format /rd 1,2;
    print "The sum is:     " sum;
endp;
```

The keyword defined above will print the string argument passed to it. The argument will be printed enclosed in single quotes.

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. For example, if you type

```
add 1 2 3 4 5;
```

the keyword will respond

```
The sum is: 15.00
```

Here is another example:

```
add;
```

the keyword will respond

```
The argument is a null string
```

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 four arguments. The first is a procedure **f1** that has two 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 **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 **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.

Indexing Procedures

This example assumes there are a set of procedures named **f1-f5** that 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 that will return the value of the *ith* 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;
```

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 *ith* pointer, **f** is declared to be a procedure and is called as a procedure in the **retp** statement.

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)));
```

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, that is, 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 a `.fcg` file with the **save** command, and loaded later with the **loadp** command whenever it is needed. This will usually be faster than recompiling. For example:

```
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 a `.fcg` extension. (For details, see **loadp** and **save** in the *GAUSS Language Reference*.)

All compiled procedures should be saved in the same subdirectory so 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, use

```
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 be used only 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.

Structures 13

A structure is a collection of one or more variables grouped under a single name. The variables in a structure can be of different types or can themselves be structures.

Defining and Creating Structures

Structures are defined using the **`struct`** keyword.

```
struct dset {
    string array sdata;
    string array sdatadesc;
    matrix ndata;
    string array ndatadesc;
};

struct company {
    scalar id;
    string name;
```

```
        string contact;
        string phone;
        string fax;
        struct dset data;
    };
```

These statements create structure definitions that persist until the workspace is cleared. They do not create structures, only structure type definitions. To create a variable that is a structure, use:

```
struct company c;
```

This will create a variable **c** which is a structure of type *company*.

Initializing Structures

Currently, no compile time structure initialization is supported. To initialize the structure above, the following statements could be used:

```
c.id = 49;
c.name = "XYZ Corp.";
c.contact = "John Doe";
c.phone = "(206) 555-1212";
c.fax = "(206) 555-1313";
c.data.sdata = sa;
c.data.sdatadesc = sadesc;
c.data.ndata = x;
c.data.ndatadesc = xdesc;
```

The assumption is that **x** is an existing matrix and **sa**, **sadesc** and **xdesc** are existing string arrays.

Passing Structures to Procedures

Structures or members of structures can be passed to procedures. When a structure is passed as an argument to a procedure, it is passed by value. The structure becomes a local copy of the structure that was passed. The data in the structure is not duplicated unless the local copy of the structure has a new value assigned to one of its members.

Structure arguments must be declared in the procedure definition.

```
struct rectangle {
        matrix ulx;
        matrix uly;
        matrix lrx;
        matrix lry;
    };
proc area(struct rectangle rect);
        retp( (rect.lrx - rect.ulx) .* (rect.uly -
   rect.lry) );
    endp;
```

Local structures are defined using a **struct** statement inside the procedure definition.

```
proc center(struct rectangle rect);
    struct rectangle cent;

    cent.lrx = (rect.lrx - rect.ulx) ./ 2;
    cent.ulx = -cent.lrx;
    cent.uly = (rect.uly - rect.lry) ./ 2;
    cent.lry = -cent.uly;

    retp(cent);
endp;
```

Libraries 14

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.

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.

Forward References

When the compiler encounters a symbol that has not previously been defined, it 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 such as:

```
x = 5;
```

Left-hand side references, since they are assignments, are assumed to be matrices. In the previous statement, **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 of these statements:

```
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 such as:

```
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.

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 deleted automatically.

If the autoloader is ON, GAUSS searches for unresolved symbol references during compilation using a specific search path. If the autoloader is OFF, an **`Undefined symbol`** error message 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 message 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 not listed in an active library are not allowed. For example:

```
x = rndn(10,10);
y = sym(x);      /* forward reference to symbol */

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 want 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 to contain your desired procedure definitions. By convention, this file is usually named with a `.src` extension, but you can use any name and any file extension. In this file, put all the definitions of related procedures you wish to use. Here is an example of such a file, 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 one-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. (For details, see **lib** in the *GAUSS Language 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 of the following library files are valid:

Example 1

```
/*
**  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
```

Example 2

```
/*
**  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, combine related procedures in the same file in order to minimize the compiling of procedures not needed by your program. Do not combine unrelated functions in one `.src` file, because if one function in a `.src` file is needed, the whole file will be compiled.

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:

```
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.

For details of the parameters available with the **`lib`** command, see the *GAUSS Language Reference*.

.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` in one of the subdirectories listed in your **`src_path`**. If autodelete is OFF, the `.g` file must be listed in an active library; for example, in the user library.

Global Declaration Files

If your application makes use of several global variables, create a file containing **`declare`** statements. Use files with the extension `.dec` to assign default values to global matrices and strings with **`declare`** statements. A file with a `.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.

Here is an example that illustrates the way in which `.dec`, `.ext`, `.lcg`, and `.src` files work together. Always begin the names of global matrices or strings with '_' to distinguish them from procedures:

`.src` File

```
/*
**  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;
```

`.dec` File

```
/*
**  fcomp.dec – global declaration file for fuzzy
**  comparisons.
*/

declare matrix _fcmptol != 1e-14;
```

`.ext` File

```
/*
**  fcomp.ext – external declaration file for fuzzy
**  comparisons.
*/

external matrix _fcmptol;
```

`.lcg` File

```
/*
**  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 can 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;
```

Troubleshooting

Below is a partial list of errors you may encounter in using the library system, followed by the most probable cause.

(4) : error G0290 : 'c:\gauss\lib\prt.lcg' : Library not found

The autoloader is looking for a library file called `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 cannot 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 c:\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 not have activated the library that contains your symbol definition. Do so in a **library** statement.

c:\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.

```
c:\gauss\lib\prt.lcg(5)  : error G0301 : 'prt.dec' :
Syntax error in library
Undefined symbols:
__VNAMES c:\gauss\src\prt.src(6)
```

Check your library to see that all filenames are flush left and all symbols defined in that file are indented by at least one space.

Using dec Files

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 during 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 a global in more than one file.
- To avoid meaningless redefinition errors and **declare** warnings, never declare a global more than once in any one file. 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.

By following these guidelines, any **declare** warnings and redefinition errors you get will be meaningful. By knowing that such warnings and errors are significant, you will be able to debug your programs more efficiently.

Compiler 15

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. GAUSS is not a native code compiler; rather, it compiles to a form of pseudocode. The file will have a `.gcg` extension.

The **`compile`** command will compile an entire program to a compiled file. 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 that 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.

Compiling Programs

Programs are compiled with the **`compile`** command.

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 must be present and active during the compile, but not when the program is run. If the program uses the **dlibrary** command, the `.dll` files must be present when the program is run and the dlibrary path must be set to the correct subdirectory. This will be handled automatically in your configuration file. If the program is run on a different computer than it was compiled on, the `.dll` files must be present in the correct location. **sysstate** (case 24) can be used to set the **dlibrary** path at run-time.

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;
```

List the procedures you will be using in an **external** statement and follow it with a **saveall** statement. It is not necessary to list procedures you do not explicitly call, but are called from another procedure, because the autoloader will automatically find them before the **saveall** command is executed. Nor is it necessary to list every procedure you will be calling, unless the source will not be available when the compiled file is **use**'d.

Remember, 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.

Debugging

If you are using compiled code in a development situation where debugging is important, 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.

File I/O 16

The 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 information of files matching a specification.
`files`	Returns a directory listing as a character matrix.

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	Returns type of data set (bytes per element).
writer	Write data to a data set.

ASCII Files

GAUSS has facilities for reading and writing ASCII files. Since most software can read and write ASCII files, this provides a way of sharing data between GAUSS and many other kinds of programs.

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 see "ATOG," page 21-1. 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 resulting GAUSS matrix must be no larger than the limit for a single matrix.

For example,

```
load x[] = dat1.asc;
```

will load the data in the file `dat1.asc` into an Nx1 matrix **x**. This method is preferred 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, 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 automatically be reshaped to 100x5.

Writing

To write data to an ASCII file, the **print** or **printfm** command is 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. The **print** or **printfm** command is used to control what is sent to the output file.

The window can be turned on and off using **screen**. When printing a large amount of data to the auxiliary output, the window can be turned off using the command

```
screen off;
```

This will make the process much faster, especially if the auxiliary output is a disk file.

It is easy to forget to turn the window on again. 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	Specify 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 window on and off.

This 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 follows. This program will write the contents of the GAUSS file `mydata.dat` into an ASCII file called `mydata.asc`. If there is an existing file by the name of `mydata.asc`, it will 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 window is turned off to speed up the process. The GAUSS data file `mydata.dat` is opened for reading, and 200 rows will be read per iteration until the end of the file is reached. The data read will be printed to the auxiliary output `mydata.asc` only, because the window is off.

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 following example 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;
```

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.

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 limited by RAM. 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 are:

Bytes	Data Type	Significant Digits	Range
2	integer	4	$-32768 \le X \le 32767$
4	single	6-7	$8.43E-37 \le \lvert X\rvert \le 3.37E+38$
8	double	15-16	$4.19E-307 \le \lvert X\rvert \le 1.67E+308$

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. (For details, see **`create`** in the *GAUSS Language Reference*.)

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 entire data set. If you must, explicitly convert the rows of data first, using the **`real`** and **`imag`** functions (see the *GAUSS Language Reference*), and 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.

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 in the example 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 example 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 the previous 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 that will be equal to the larger of the two data types of the input files. This will ensure 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 files.

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 Applications programs do. When creating a data set, it is important 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 distinguishes 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 of 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, use **vartypef**:

```
open f = mydata for read;
vn = getnamef(f);
vt = vartypef(f);
print vn';
print vt';
```

```
   Sex  Age  Pay
    0    1    1
```

The function **getnamef** in the previous example returns a string array rather than a character vector, so you can print it without the '**$**' prefix.

Using the Uppercase/Lowercase Convention (v89 Data Sets)

This is obsolete, use **vartypef** and v96 data sets to be compatible with future versions.

The following method for distinguishing character/numeric data will soon be obsolete; use the Type Vectors method described earlier.

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; if the variable name 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**.

GAUSS does not make this distinction internally. 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" in 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 writing or creating a data set, the case of the variable names is important. This is especially true if the GAUSS applications programs will be used on the data set.

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** command, 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 is useful when only a subset of rows is needed at any time. This procedure will save memory and be much faster than **load**'ing the entire matrix into memory.

File Formats

This section discusses the GAUSS binary file formats.

There are four currently supported matrix file formats.

Version	Extension	Support
Small Matrix v89	`.fmt`	Obsolete, use v96.

Extended Matrix v89	`.fmt`	Obsolete, use v96.
Matrix v92	`.fmt`	Obsolete, use v96.
Universal Matrix v96	`.fmt`	Supported for read/write.

There are four currently supported string file formats:

Version	Extension	Support
Small String v89	`.fst`	Obsolete, use v96.
Extended String v89	`.fst`	Obsolete, use v96.
String v92	`.fst`	Obsolete, use v96.
Universal String v96	`.fst`	Supported for read/write.

There are four currently supported data set formats:

Version	Extension	Support
Small Data Set v89	`.dat`, `.dht`	Obsolete, use v96.
Extended Data Set v89	`.dat`, `.dht`	Obsolete, use v96.
Data Set v92	`.dat`	Obsolete, use v96.
Universal Data Set v96	`.dat`	Supported for read/write.

Small Matrix v89 (Obsolete)

Matrix files are binary files, and cannot be read with a text editor. They are created with **`save`**. Matrix files with up to 8190 elements 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]

Extended Matrix v89 (Obsolete)

Matrices with more than 8190 elements are saved in an extended format. These 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 body is the same as the small matrix v89 file.

Small String v89 (Obsolete)

String files are created with **save**. String files with up to 65519 characters have a 16-byte header formatted as follows:

Offset	Description

0-1	DFDF hex, identification flag
2-3	1, unsigned 2-byte integer

Offset	Description
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.

Extended String v89 (Obsolete)

Strings with more than 65519 characters are saved in an extended format. These 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.

Small Data Set v89 (Obsolete)

All data sets are created with **`create`**. v89 data sets consist of two files; one (`.dht`) contains the header information; the second (`.dat`) contains the binary data. The data will be 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. Data sets with up to 8175 columns have a `.dht` file formatted 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, 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 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]
```

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:

```
[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
```

Extended Data Set v89 (Obsolete)

Data sets with more than 8175 columns are saved in an extended format. These 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.

Matrix v92 (Obsolete)

Offset	Description
0-3	always 0
4-7	always 0xEECDCDCD
8-11	reserved

Offset	Description
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 + 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 the imaginary parts.

String v92 (Obsolete)

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 8 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.

Data Set v92 (Obsolete)

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 - real data set, 1 - 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.

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 - real matrix, 1 - complex matrix
56-59	1 - imaginary part of matrix follows real part (standard GAUSS style) 1 - 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 - row major ordering of elements, 2 - 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.

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 - signed 2-byte integer 1004 - single precision floating point 1008 - double precision float
48-51	2, 4, or 8, the size of an element in bytes
52-55	0 - real matrix, 1 - 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

Offset	Description
64-67	1 - row major ordering of elements, 2 - 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.

Foreign Language Interface 17

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 libraries (DLLs, also known as shared libraries or shared objects) and linked in at run-time as needed. The FLI 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 **`$(GAUSSDIR)/ dlib`** as the default directory.

Creating Dynamic Libraries

Assume you want to build a dynamic library called `myfuncs.dll`, 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. The compiler command is first, followed by the linker command, followed by remarks regarding that platform.

For explanations of the various flags used, see the documentation for your compiler and linker. One flag is common to both 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 **(**`.obj`**)** files, then link the object files in a separate phase into a dynamic library.

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

```
cl -c $(CCOPTS) -DWIN32 -D_WIN32 -D_MT -c -W3 -
   Dtry=__try \
   -Dexcept=__except -Dleave=__leave -
   Dfinally=__finally \
   -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -D_X86_=1 -DSTRICT
   -LD \
   -Zp1 myfunc1.c myfunc2.c
link -DLL -def:ntgauss.def -out:myfuncs.dll myfunc1.obj
   \
   myfunc2.obj fp10.obj libcmt.lib oldnames.lib
   kernel32.lib \
   advapi32.lib user32.lib gdi32.lib comdlg32.lib
   winspool.lib
```

These commands are written for the Microsoft Visual C/C++ compiler, ver. 2.0.

The Visual C/C++ linker allows you to specify a module definition file, which is a text file that describes the dynamic library to be created. In this example, the module definition file is `myfuncs.def`. It includes information on how the library is to be initialized and terminated, how to handle its data segment, etc. It also needs to list the symbols that will be exported, i.e., made callable by other processes, from the dynamic library. Assume that `myfunc1.c` and `myfunc2.c` contain the FLI functions **func1()**, **func2()**, and **func3()**, and a static funciton **func4()** that is called by the others, but never directly from GAUSS. Then `myfuncs.def` would look like this:

```
LIBRARY myfuncs
EXPORTS
   func1
   func2
   func3
```

As you can see, creating dynamic libraries from the command line can be quite an arcane process. For this reason, we recommend that you create dynamic libraries from inside the Visual C/C++ workbench environment, rather than from the command line.

Writing FLI Functions

Your FLI functions should be written to the following specifications:

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

   This does not mean you cannot 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 *2, etc.]]) ;
3. **int func**(double *arg[ ]) ;

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...]]) ;

For details on calling your function, passing arguments to it, getting data back, and what the return value means, see **dllcall** in the *GAUSS Language Reference*.

Data Exchange 18

Data Exchange procedures are used to move data between GAUSS and other software tools. Two procedures export GAUSS matrices and data sets to formats that can be read by a variety of spreadsheets and databases. Two more procedures import data files from these formats into GAUSS matrices and data sets.

Formats Supported

The table below lists the spreadsheet and database formats supported, and their usual file extensions. The default file type is determined from the file extension. This default can be overwritten so that other file extensions can be used. For more information see “Global Variables,” page 18-3.

File Type	File Extension	
	Export	Import
Lotus v1–v5	`.wks`	`.wks .wk1–.wk5`
Excel v2.1–v7.0	`.xls`	`.xls`
Quatro v1–v6	`.wq1`	`.wq1 .wq2 .wb1`
Symphony v1.0–1.1	`.wrk`	`.wrk`

File Type	File Extension	
	Export	Import
dBase II	`.db2`	`.db2`
dBase III/IV	`.dbf`	`.dbf`
Paradox, FoxPro, Clipper	`.db`	`.db`
ASCII – character delimited	`.csv .txt .asc`	`.csv .txt .asc`
ASCII – formatted	`.prn`	`.prn`
GAUSS data set	`.dat`	`.dat`

The newer releases of spreadsheets and databases can read their older formats. Specifically, the procedures will read data in all versions listed and will write data out in the most compatible, or earliest, format. For example, the Lotus driver will read all versions up to version 5 and will write data out to a generic version 1.0 `.wks` file.

You can import and export ASCII files.

If you have a problem importing data, save it in an earlier spreadsheet or database format. The same suffix does not necessarily mean the same data format.

Importing sheet 0 or sheet 1 will import the first sheet of a spreadsheet. (Most multi-sheet spreadsheets call this sheet 1.)

On export, elements that are missing values will be exported to spreadsheets as blank cells, and to ASCII files as the value **`_dxmiss`**. On import, spreadsheet cells that are **#ERR** or **#N/A** will be imported as GAUSS missing values. Elements from any format that have the value **`_dxmiss`** will be imported as GAUSS missing values.

Data Exchange Procedures

These four procedures are used for exchanging data with databases and spreadsheets:

export	Exports a GAUSS matrix to a specified file format.
exportf	Exports a GAUSS data set to a specified file format.
import	Imports a spreadsheet or database file to a GAUSS matrix.
importf	Imports a spreadsheet or database file to a GAUSS data set.

For details, see the *GAUSS Language Reference*.

Global Variables

The following global variables can be used to modify the operation of the Data Exchange procedures:

`_dxftype` (string)	Overrides the file extension to define the type of file to import or export. For example, after setting **`_dxftype = "xls"`**, files exported or imported will be Excel format, independent of the actual file extension. Use **`_dxftype = ""`** (empty string) to return to default operation (file extension defining file type).
`_dxtype` (matrix)	Scalar or Kx1 vector of 1's and 0's defining the data types of columns. 1's indicate numeric columns, 0's indicate character columns. A scalar can be used if all columns are of the same type. Default is scalar 1 (all numeric).
`_dxwidth` (matrix)	Scalar or Kx1 vector of integers giving the width of spreadsheet columns in characters. A scalar can be used if all columns have the same width. Default is 12. (**`_dxwidth`** does not always control the column width correctly when exporting to an Excel (`.xls`) datasheet or databook. Adjust this parameter within Excel after loading the file.)
`_dxprcn` (matrix)	Scalar or Kx1 vector defining the number of digits of precision in the columns of spreadsheets. A scalar can be used if all columns are to have the same precision. Default is 4. (**`_dxprcn`** does not always control the number of digits after the decimal correctly when exporting to an Excel (`.xls`) datasheet or databook. Adjust this parameter within Excel after loading the file.)
`_dxtxdlim` (scalar)	ASCII value of character that delimits fields in ASCII files (tab = 9, comma = 44, space = 32). Default is space (32).
`_dxaschdr`	Scalar (0 or 1) determining if column headers are written to/from ASCII files. If **`_dxaschdr = 1`**, headers are written. If **`_dxaschdr = 0`**, no headers are written. Default is 0.
`_dxwkshdr`	Scalar (0 or 1) determining if column headers are written to/from spreadsheet files. If **`_dxwkshdr = 1`**, headers are written. If **`_dxwkshdr = 0`**, no headers are written. Default is 0.
`_dxmiss`	Scalar that defines the missing value representation. Default is the normal GAUSS representation (the indefinite NaN).

`_dxprint`	Scalar (0 or 1) determining if progress messages are to be printed to the window. **`_dxprint = 1`** prints messages. **`_dxprint = 0`** suppresses the print. Default is 1.

Data Transformations 19

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 performs 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.

Using Data Loop Statements

A data loop begins with a **dataloop** statement and ends with an **endata** statement. Inside 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 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.

Using Other Statements

All program statements in the main file and not inside 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 user familiar with the code generated in the temporary file can use this to do out-of-the-ordinary operations inside the data loop.

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.

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 data loop procedures are placed at the end of the temporary file.

Depending on 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.

Compilation Phase

In the compilation phase, the temporary file is compiled. Depending on 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.

Execution Phase

In the execution phase, the compiled program is executed. Depending on the status of line number tracking, error messages will include line number references from both the main file and the temporary file.

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.

Publication Quality Graphics 20

GAUSS Publication Quality Graphics (PQG) is a set of routines built on the graphics functions in GraphiC by Scientific Endeavors Corporation.

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 window with several equally sized graphs, or place several overlapping graphs in the window. Graphic panel size and location are all completely under the user's control.

General Design

GAUSS PQG consists of a set of main graphing procedures and several additional procedures and global variables for customizing the output.

All of the actual output to the window happens during the call to these main routines:

`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.

Using Publication Quality Graphics

Getting Started

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.

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. For example:

```
library mylib, pgraph;

graphset;
```

Data Setup

The data to be graphed must be in matrices. For example:

```
x = seqa(1,1,50);

y = sin(x);
```

Graphics Format Setup

Most of the graphics elements contain defaults that 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.

Calling Graphics Routines

The graphics routines take as input the user data and global variables that have previously been set. It is in these routines where the graphics file is created and displayed.

Following are three PQG 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. (For a detailed description of these variables, see "Global Control Variables," page 20-13.)

Example 1 The routine being called here is a simple XY plot. The entire window 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); 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= “Sine wave 1.\0”\   /* 4 lines legend text */
          “Sine wave .8\0”\
          “Sine wave .6\0”\
          “Sine 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 graphic panels are drawn. The first 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 window:

```
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 */
                            /* graphic panels */
makewind(9,6.855,0,0,0);    /* first graphic panel */
                            /*full size */
makewind(9/2,6.855/2,1,1,0);/* second graphic panel */
                           /* inset to first */

setwind(1);                 /* activate first graphic */
                            /* panel */
   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 graphic */
                          /* panel */
   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 graphic panels */
```

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 graphic panels.

As Example 3 illustrates, the code between graphic panel functions (that is, **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.

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 PQG system. The example programs are located on the **examples** subdirectory.

Graphics Coordinate System

PQG uses a 4190x3120 pixel grid on a 9.0x6.855-inch printable area. There are three 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 Graphic Panels,” page 20-9.)

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.

Graphics Graphic Panels

Multiple graphic panels for graphics are supported. These graphic panels allow the user to display multiple graphs on one window or page.

A graphic panel is any rectangular subsection of the window or page. Graphic panels may be any size and position on the window and may be tiled or overlapping, transparent or nontransparent.

Tiled Graphic Panels

Tiled graphic panels do not overlap. The window can easily be divided into any number of tiled graphic panels with the **window** command. **window** takes three parameters: number of rows, number of columns, and graphic panel attribute (1=transparent, 0=nontransparent).

This example will divide the window into six equally sized graphic panels. There will be two rows of three graphic panels — three graphic panels in the upper half of the window and three in the lower half. The attribute value of 0 is arbitrary since there are no other graphic panels beneath them:

```
window(nrows,ncols,attr);

window(2,3,0);
```

Overlapping Graphic Panels

Overlapping graphic panels 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 graphic panel is created with the **`makewind`** command.

In this example, **`makewind`** will create an overlapping graphic panel 4 inches horizontally by 2.5 inches vertically, positioned 1 inch from the left edge of the page and 1.5 inches from the bottom of the page. It will be nontransparent:

```
makewind(hsize,vsize,hpos,vpos,attr);


window(2,3,0);

makewind(4,2.5,1,1.5,0);
```

Nontransparent Graphic Panels

A nontransparent graphic panel is one that is blanked before graphics information is written to it. Therefore, information in any previously drawn graphic panels that lie under it will not be visible.

Transparent Graphic Panels

A transparent graphic panel is one that is not blanked, allowing the graphic panel beneath it to “show through.” Lines, symbols, arrows, error bars, and other graphics objects may extend from one graphic panel to the next by using transparent graphic panels. First, create the desired graphic panel configuration. Then create a full-window, transparent graphic panel using the **`makewind`** or **`window`** command. Set the appropriate global variables to position the desired object on the transparent graphic panel. Use the **`draw`** procedure to draw it. This graphic panel will act as a transparent “overlay” on top of the other graphic panels. Transparent graphic panels can be used to add text or to superimpose one graphic panel on top of another.

Using Graphic Panel Functions

The following is a summary of the graphic panel functions:

begwind	Graphic panel initialization procedure.
endwind	End graphic panel manipulations, display graphs.
window	Partition window into tiled graphic panels.
makewind	Create graphic panel with specified size and position.
setwind	Set to specified graphic panel number.
nextwind	Set to next available graphic panel number.
getwind	Get current graphic panel number.
savewind	Save graphic panel configuration to a file.
loadwind	Load graphic panel configuration from a file.

This example creates four tiled graphic panels and one graphic panel that overlaps the other four:

```
library pgraph;
graphset;
begwind;

window(2,2,0);/* Create four tiled graphic panels */
              /* (2 rows, 2 columns) */

xsize = 9/2;  /* Create graphic panel that overlaps */
              /*the tiled graphic panels */
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 graphic panel processing, */
              /* display graph */
```

Inch Units in Graphics Graphic Panels

Some global variables allow coordinates to be input in inches. If a coordinate value is in inches and is being used in a graphic panel, that value will be scaled to **`window inches`** and positioned relative to the lower left corner of the graphic panel. A graphic panel inch is a true inch in size only if the graphic panel is scaled to the full window; otherwise, **`X`** coordinates will be scaled relative to the **`horizontal`** graphic panel size and **`Y`** coordinates will be scaled relative to the **`vertical`** graphic panel size.

Saving Graphic Panel Configurations

The functions **`savewind`** and **`loadwind`** allow the user to save graphic panel configurations. Once graphic panels are created (using **`makewind`** and **`window`**), **`savewind`** may be called. This will save to disk the global variables containing information about the current graphic panel configuration. To load this configuration again, call **`loadwind`**. (See **`loadwind`** in the *GAUSS Language Reference*.)

Graphics Text Elements

Graphics text elements, such as titles, messages, axes labels, axes numbering, and legends, can be modified and enhanced by changing fonts and by adding superscripting, subscripting, and special mathematical symbols.

To make these modifications and enhancements, the user can embed "escape codes" in the text strings that 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.
\ 20*n*	Select font number *n* (see "Selecting Fonts," following).

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 generate **[** and **]** in an X axis label:

```
xlabel("Data used for x is: data@[.,1 2 3@]");
```

Selecting Fonts

Four fonts are supplied with the Publication Quality Graphics system. They are Simplex, Complex, Simgrma, and Microb. (For the characters available in each font, 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 can be used in text strings. A font can then be selected by embedding an escape code of the form "\ 20*n*" in the string that 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 the fonts were loaded as in the previous example, the escape characters for each would be:

```
\201  Simplex
\202  Complex
\203  Microb
\204  Simgrma
```

The example then for selecting a font for each string to be written would be:

```
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.

Greek and Mathematical Symbols

The following examples illustrate the use of the Simgrma font; they assume that Simgrma was the fourth font loaded. (For the available Simgrma characters and their numbers, see Appendix A.) The Simgrma characters are specified by either:

1. The character number, preceded by a "\".
2. The regular text character with the same number.

For example, to get an integral sign "**ò**" in Simgrma, embed either a "\ 044" or a "," in the string that has been currently set to use Simgrma font.

To produce the title $f(\mathrm{x}) = sin^2(\pi x)$, use the following title string:

```
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 π/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

$$\sqrt{1/2\pi} \int e^{-\mu^2/2} d\mu$$

Colors

0	Black	8	Dark Grey
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow

7 Grey 15 White

Global Control Variables

The following global variables are used to control various graphics elements. Default values are provided. Any or all of these variables can be set before calling one of the main graphing routines. The default values can be modified by changing the declarations in `pgraph.dec` and the statements in the procedure **graphset** in `pgraph.src`. **graphset** can be called whenever the user wants 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 0. If this is 0, the graph will be centered on the output page. Default is 0.

Note: Used internally. (For the same functionality, see **axmargin** in the *GAUSS Language Reference*.) This is used by the graphics graphic panel routines. The user must not set this when using the graphic panel 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. Default is 0.

Note: Used internally. (For the same functionality, see **axmargin** in the *GAUSS Language Reference*.) This is used by the graphics graphic panel routines. The user must not set this when using the graphic panel 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 type 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, see “Colors,” page 20-12.

[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.

To create two single-headed arrows, located using inches, use

```
_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 half its 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 type 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, see “Colors,” page 20-12.

[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.

To create two single-headed arrows, located using plot coordinates, use

```
_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 is 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, see “Colors,” page 20-12.

_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 is 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 astwo vertical lines representing the quartile ranges with a filled circle representing the 50^{th} percentile.

[2] box color, see “Colors,” page 20-12. 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:

intqrange	=	$75^{th} - 25^{th}$
min	=	$25^{th} - 1.5intqrange$
max	=	$75^{th} + 1.5intqrange$

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 is 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, see "Colors," page 20-12. (**_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**.

This example will crop main curves, and lines and circles drawn by **_pline**:

```
_pcrop = { 1 1 0 1 0 };
```

_pcross scalar. If 1, the axes will intersect at the (0,0) X-Y location if it is visible. 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 (the first character is a font selection escape code):

```
_pdate = "\201GAUSS  ";
```

If this is set to a null string, no date will be printed. (For more information on using fonts within strings, see "Graphics Text Elements," page 20-9.)

_perrbar Mx9 matrix, draws one error bar per row of the input matrix. If scalar 0, no error bars will be drawn. Location values are in plot coordinates.

[M,1] x location.

[M,2] left end of error bar.

[M,3] right end of error bar.

[M,4] y location.

[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, see "Colors," page 20-12.

[M,9] line thickness. This value may be zero or greater. A value of zero is normal line width.

To create one error bar using solid lines, use

```
_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 is 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.

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:

```
_plctrl = { 0, 10, -5 };
```

_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 ≤ size ≤ 9. Default is 5.

[3] x coordinate of lower left corner of legend box.

[4] y coordinate of lower left corner of legend box.

This example puts a legend in the lower right corner:

```
_plegctl = 1;
```

This example creates a smaller legend and positions it 2.5 inches from the left and 1 inch from the bottom:

```
_plegctl = { 2 3 2.5 1 };
```

_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 is 0. (See **contour** in the *GAUSS Language Reference*.)

_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. 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.
- **[M,4]** y starting point.
- **[M,5]** x ending point.
- **[M,6]** y ending point.
- **[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.
- **[M,4]** y starting point.
- **[M,5]** x ending point.
- **[M,6]** y ending point.
- **[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.
- **[M,4]** y starting point.
- **[M,5]** x ending point.
- **[M,6]** y ending point.
- **[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.
- **[M,4]** y center of circle.
- **[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.

[M,4] y center of circle.

[M,5] radius.

[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.

[M,4] y center of circle.

[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.

[M,4] y center of circle.

[M,5] beginning point of radius, 0 is the center of the circle

[M,6] ending point of radius.

[M,7] angle in radians.

[M,8] color, see “Colors,” page 20-12.

[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, see “Colors,” page 20-12.

[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] x distance.

[2] y distance.

If scalar 0, there will be no shift. Default is 0.

Note: Used internally. (For the same functionality, see **axmargin** in the *GAUSS Language Reference*.) This is used by the graphics panel routines. The user must not set this when using the graphic panel procedures.

_plotsiz 2x1 vector, size of the axes area in inches. If scalar 0, the maximum size will be used.

Note: Used internally. (For the same functionality, see **axmargin** in the *GAUSS Language Reference*.) This is used by the graphics panel routines. The user must not set this when using the graphic panel 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. 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 is 0.

_pmcolor 9x1 vector, color values to use for plot, see "Colors," page 20-12.

[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, see "Colors," page 20-12.
[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:

```
_pmsgstr = "Message one.\000Message two.";
```

_pnotify scalar, controls window output during the creation of the graph. Default is 1.

0	no activity to the window while writing `.tkf` file.
1	display progress as fonts are loaded and `.tkf` file is being generated.

_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 0.13 inch will be used.

_protate scalar. If 0, no rotation, if 1, plot will be rotated 90 degrees. Default is 0.

_pscreen scalar. If 1, display graph in window, if 0, do not display graph in window. Default is 1.

_psilent scalar. If 0, a beep will sound when the graph is finished drawing to the window. Default is 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 (x)	**14**	solid star (x)

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 is 0.
- **[2]** color for base (default 7), see "Colors," page 20-12. 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**, earlier.)
- **[M,4]** symbol height. If this is 0, a default height of 5.0 will be used.
- **[M,5]** symbol color, see "Colors," page 20-12.
- **[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**, earlier.)
- **[M,5]** symbol height. If this is 0, a default height of 5.0 will be used.
- **[M,6]** symbol color, see "Colors," page 20-12.
- **[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 is 0.

_ptitlht scalar, the height of the title characters in inches. If this is 0, a default height of approx. 0.13 inch 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 is 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 is 4.

_pypmax scalar, the maximum number of places to the right of the decimal point for the Y axis numbers. Default is 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 is 4.

_pzclr scalar, row vector, or Kx2 matrix, Z level color control for procedures **surface** and **contour**. (See **surface** in the *GAUSS Language Reference*.)

_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 window 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 is 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 is 4.

Utilities 21

ATOG

ATOG is a stand-alone conversion utility that converts ASCII files into GAUSS data sets. ATOG can convert delimited and packed ASCII files into GAUSS data sets. ATOG can be run from a batch file or the command line.

The syntax is:

`atog` *cmdfile*

cmdfile is the name of the command file. If no extension is given, `.cmd` will be assumed. If no command file is specified, a command summary will be displayed.

Command Summary

The following commands are supported in ATOG:

`append`	Append data to an existing file.
`complex`	Treat data as complex variables.
`input`	The name of the ASCII input file.
`invar`	Input file variables (column names).
`msym`	Specify missing value character.
`nocheck`	Do not check data type or record length.

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.

preservecase Preserves the case of variable names in output file.

The principal commands for converting an ASCII file that is delimited with spaces or commas are given in the following example.

```
input agex.asc;

output agex;

invar $ race # age pay $ sex region;

outvar region age sex pay;

outtyp d;
```

From this example, a delimited ASCII file `agex.asc` is converted to a double precision GAUSS data file `agex.dat`. The input file has five variables. The file will be interpreted as having five 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 four columns since the first column of the input file (race) is not included in the output variables. The columns of the output file will be:

column	name	data type
1	region	character
2	AGE	numeric
3	sex	character
4	PAY	numeric

The variable names are saved in the file header. Unless **`preservecase`** has been specified, the names of character variables will be saved in lower case, and the names of numeric variables will be saved in upper case. 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.

Commands

A detailed explanation of each of the ATOG commands follows.

append

Instructs ATOG to append the converted data to an existing data set:

```
append;
```

No assumptions are made regarding the format of the existing file. Make certain the number, order, and type of data converted match the existing file. ATOG creates v96 format data files, so will only append to v96 format data files.

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.

input

Specifies the file name of the ASCII file 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 c:\research\data\myfile.asc;
```

specifies a file to be located in the **`c:\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. For example:

```
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.

This example:

```
invar delimit(, N) $ name # var[5];
```

will 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,
```

This example:

```
invar delimit(,) $ name # var[5];
```

or

```
invar delimit $ name # var[5];
```

will 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 here:

```
456.4,  345.2,  533.2,
     ,  624.3,  639.5,
602.3,  333.4,
```

If the **invar** statement had specified **3** variables and N-1 delimiters, 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
   .    624.3  639.5
602.3   333.4     .
```

If N delimiters had been specified, this file would be interpreted as having two rows, and a final incomplete row that is dropped:

```
456.4   345.2  533.2
   .    624.3  639.5
```

The spaces were 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**. This example:

```
invar delimit(\r) $ name # var[5];
```

will 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.

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

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** 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 can be interpreted in different ways using different **invar** statements:

```
          ABCDEFGHIJ12345678901234567890<CR><LF>
          |        |         |         | |  |
position  1        10        20        30 31 32
```

This example:

```
invar record=32 $(1,3) group dept #(11,4.2) x[3] (*,5)
             y;
```

will result in:

variable	value	type
group	**ABC**	**character**
dept	**DEF**	**character**
X1	**12.34**	**numeric**
X2	**56.78**	**numeric**
X3	**90.12**	**numeric**
Y	**34567**	**numeric**

This example:

```
invar record=32 $ dept (*,2) id # (*,5) wage (*,2) area
```

will result in:

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.

This example:

```
msym &;
```

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.

output

The name of the GAUSS data set. A file will be created with the extension `.dat.` For example:

```
output c:\gauss\dat\test;
```

creates the file `test.dat` on the **`c:\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	$-32768 \le X \le 32767$
4	single precision	6-7	$8.43 x 10^{-37} \le \lvert X \rvert \le 3.37 x 10^{+38}$
8	double precision	15-16	$4.19 \times 10^{-307} \le \lvert X \rvert \le 1.67 \times 10^{+308}$

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, 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

Selects 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. They can be in any order. For example:

```
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 **nopreservecase**, which will force variable names for numeric variables to upper case and character variables to lower case.

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 is `agex1.cmd`:

```
input c:\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 called `xlod.asc`.

The file contains 32-character records:

```
         AEGDRFCSTy02345678960631567890<CR><LF>
         EDJTAJPSTn12395863998064839561<CR><LF>
         GWDNADMSTy19827845659725234451<CR><LF>
         |        |       |          |  |   |
position 1        10      20         30 31  32
```

The **atog** command file is `xlod.cmd`:

```
input c:\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 is `cplx.cmd`:

```
input c:\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.

Error Messages

`atog - Can't find input file`

The ASCII input file could not be opened.

`atog - Can't open output file`

The output file could not be opened.

`atog - Can't open temporary file`

Notify Aptech Systems.

`atog - Can't read temporary file`

Notify Aptech Systems.

`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 - Illegal command`

An unrecognizable command was found in a command file.

atog - Internal error

Notify Aptech Systems.

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 may be 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 may be 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.

atog - read error

A read error has occurred while converting a packed ASCII file.

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.

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. `.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 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.

F use page breaks and form feed characters.

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 `.lcg` files in the **`lib_path`** subdirectory. `gauss.lcg`, if it exists, will be listed last.

Report Format

The listing produced will go to the standard output. The order the symbols will be listed in is the same order 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; that is, you have listed them all in a **`library`** statement. `gauss.lcg` will be listed last.

Here is an exampe of a listing:

```
   Symbol      Type      File             Library      Path
=====================================================================
 1. autoreg    ------    autoreg   .src   auto.lcg     c:\gauss\src
 2. autoprt    ------    autoreg   .src   auto.lcg     c:\gauss\src
 3. autoset    ------    autoreg   .src   auto.lcg     c:\gauss\src
 4. _pticout   matrix    pgraph    .dec   pgraph.lcg   c:\gauss\src
 5. _pzlabel   string    pgraph    .dec   pgraph.lcg   c:\gauss\src
 6. _pzpmax    matrix    pgraph    .dec   pgraph.lcg   c:\gauss\src
 7. asclabel   proc      pgraph    .src   pgraph.lcg   c:\gauss\src
 8. fonts      proc      pgraph    .src   pgraph.lcg   c:\gauss\src
 9. graphset   proc      pgraph    .src   pgraph.lcg   c:\gauss\src
10. _svdtol    matrix    svd       .dec   gauss.lcg    c:\gauss\src
12. _maxvec    matrix    system    .dec   gauss.lcg    c:\gauss\src
13. besselj    proc      bessel    .src   gauss.lcg    c:\gauss\src
```

```
14. bessely    proc     bessel   .src  gauss.lcg    c:\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 ***`**.

Using LIBLIST

LIBLIST is executed with

`liblist` -*flags lib1 lib2 ... libn*

To put the listing in a file called `lib.lst`, use

`liblist -l > lib.lst`

To convert all your `.g` files and list them in `srcfile.lst`, use

`liblist -c`

The numbers are there so that you can sort the listing 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 do not inadvertently use the wrong one.

Error Messages 22

The following is a list of error messages intrinsic to the GAUSS programming language. 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 into smaller pieces. Notify Aptech Systems.

G0005 File is already compiled

G0006 Statement too long

Statement longer than 4000 characters.

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. Notify Aptech Systems.

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 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 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 been given a definition.

G0026 Too many symbols

The global symbol table is full. (To set the limit, see **new** in the *GAUSS Language Reference*.)

G0027 Invalid directory

G0028 Can't open configuration file

GAUSS cannot find the 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. (To make the main program space smaller and reclaim enough space to continue, see **new** in the *GAUSS Language Reference*.)

G0031 Execution stack too deep - expression too complex

An expression is too complex. Break it into smaller pieces. Notify Aptech Systems.

G0032 fn function too large

G0033 Missing right index bracket

G0034 Missing arguments

G0035 Argument too large

G0036 Matrices are not conformable

For a description of the function or operator being used and conformability rules, see “Matrix Operators,” page 11-4, or the *GAUSS Language Reference*.

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. (To increase the main program space, see **`new`** in the *GAUSS Language Reference*.)

G0051 Program too large

The main program area is full. (To increase the main program space, see **`new`** in the *GAUSS Language Reference*.)

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 a procedure definition.

G0056 Too many active locals

The execution stack is full. There are too many local variables active. Restructure your program. Notify Aptech Systems.

G0057 Procedure stack overflow - expression too complex

The execution stack is full. There are too many nested levels of procedure calls. Restructure your program. Notify Aptech Systems.

`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 intervening operand.

`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.

`G0068 Program compiled under GAUSS-386 real version`

`G0069 Program compiled under GAUSS-386i complex version`

`G0070 Procedure calls too deep`

You may have 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. **`delete`** the symbol and try again. If this happens in the context of a single program, you have a programming error. If this is a conflict between different programs, 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 cannot **`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 cannot 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 8 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`

`parse` encountered a token that was too long.

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 ###

Notify Aptech Systems.

G0101 Argument cannot be zero

The argument to **ln** or **log** cannot be zero.

G0102 Subroutine calls too deep

Too many levels of **gosub**. Restructure your program.

G0103 return without gosub

You have encountered a subroutine without executing a **gosub**.

G0104 Argument must be positive

G0105 Bad expression or missing arguments

Check the expression in question, or you forgot an argument.

G0106 Factorial overflow

G0107 Nesting too deep

Break the expression into smaller statements.

G0108 Missing left bracket [

G0109 Not enough data items

You omitted 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 attempting to turn a function into a matrix or string. If this is a name conflict, **delete** the function.

G0116 sysstate: invalid case

G0117 Invalid argument

G0118 Argument must be integer

File handles must be integral.

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 attempting 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

For a description of the function or operator being used and conformability rules, see "Matrix Operators," page 11-4, or the *GAUSS Language Reference*.

G0150 Matrix not square

G0151 Sort failure

G0152 Variable not initialized

You have referenced a variable that has not been initialized to any value.

G0153 Unsuccessful close on auxiliary output

The disk may be 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 attempting to turn a procedure into a matrix or string. If this is a name conflict, **delete** the procedure.

G0157 Illegal redefinition of matrix

G0158 endp without proc

You are attempting 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

Remove the global references or leave this in source code form for the autoloader to handle. (See **library** in the *GAUSS Language Reference*.)

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

G0274 Invalid parameter for conv

G0275 Parameter is NaN (Not A Number)

The argument is a NaN (see “Special Data Types,” page 10-20).

G0276 Illegal use of reserved word

G0277 Null string illegal here

G0278 proc without endp

You must terminate a procedure definition with an **endp** statement.

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. These functions are obsolete, please use **rndlcn** and **rndlcu**

G0288 Found break not in do loop

G0289 Found continue not in do loop

G0290 Library not found

The specified library cannot be found on the **lib_path** path. Make sure installation was correct.

G0291 Compiler pass out of memory

Notify Aptech Systems.

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.

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 may be full.

G0296 Can't raise negative number to nonintegral power

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	Cannot 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

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`

`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`

Notify Aptech Systems.

`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 allowed only 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

G0421 compile command not supported in Run-Time Module

G0428 Cannot use debug command inside program

G0429 Invalid number of subdiagonals

G0431 Error closing dynamic library

G0432 Error opening dynamic library

G0433 Cannot find DLL function

G0435 Invalid mode

G0436 Matrix is empty

G0437 loadexe not supported; use dlibrary instead

G0438 callexe not supported; use dllcall instead

G0439 File has wrong bit number

G0441 Type vector malloc failed

G0442 No type vector in gfblock

G0445 Illegal left-hand side reference in procedure

G0447 vfor called with illegal loop level

G0454 Failure opening printer for output

G0456 Failure buffering output for printer

G0457 Can't take log of a negative number

G0458 Attempt to index proc/fn/keyword as a matrix

G0459 Missing right brace }

G0460 Unexpected end of statement

G0461 Too many data items

G0462 Negative trim value

G0463 Failure generating graph

Maximizing Performance 23

These hints will help you maximize the performance of your new GAUSS System.

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 RAM disk, the temporary files will be placed on the RAM disk.

For example:

```
set tmp=f:\tmp
```

`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 the programs are rerun.

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 **do** 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 use:

```
c = sumc(abs(x) .> 1);
print c;
```

The **do** loop takes over 40 times longer.

Virtual Memory

The following are hints for making the best use of virtual memory in GAUSS.

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 entire data set 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, the fastest time occurred when the data was read from disk in small enough sections to allow the **`ols`** procedure to execute entirely in RAM. 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 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.

Hard Disk Maintenance

The hard disk used for the swap file should be optimized occasionally with a disk optimizer. Use a disk maintenance program to ensure that the disk media is in good shape.

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 computer.

This affects the choice of algorithms used for matrix multiply functions.

This will not change the results you get, but it can radically affect performance for large matrices.

Fonts Appendix A

There are four fonts available in the Publication Quality Graphics System:

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. (For details on selecting fonts for your graph, see “Selecting Fonts,” page 20-10.

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	≠	89	Ψ	117	υ
34	(	62	≧	90	≳	118	)
35	≡	63	≃	91	[	119	ω
36	≈	64	∪	92	∂	120	ξ
37	↑	65	½	93	]	121	ψ
38	√	66	⅓	94	∩	122	ζ
39	′	67	H	95	↓	123	{
40	⊂	68	Δ	96	″	124	∫
41	⊃	69	⅛	97	α	125	}
42	×	70	Φ	98	β	126	∝
43	±	71	Γ	99	η		
44	∫	72	X	100	δ		
45	∓	73	⅔	101	ε		
46	∘	74	⊥	102	φ		
47	÷	75	⅜	103	γ		
48	∇	76	Λ	104	χ		
49	√	77	⅝	105	ι		
50	∮	78	⅞	106	t		
51	<	79	¼	107	κ		
52	>	80	Π	108	λ		
53	/	81	Θ	109	μ		
54	∃	82	P	110	ν		
55	‖	83	Σ	111	ο		
56	∞	84	≲	112	π		
57	⊙	85	ϒ	113	ϑ		
58	→	86	↔	114	ρ		
59	←	87	Ω	115	σ		
60	≦	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		

Reserved Words Appendix B

The following words are used for GAUSS intrinsic functions. You cannot use these names for variables or procedures in your programs:

a

abs	and	atan	atan2

b

balance	bandcholsol	bandsolpd	break
band	bandltsol	besselj	
bandchol	bandrv	bessely	

c

call	cdfchic	cdfnc	cdftvn
callexe	cdffc	cdfni	cdir
cdfbeta	cdfgam	cdftc	ceil
cdfbvn	cdfn	cdftci	cfft

cffti	clearg	complex	countwts
ChangeDir	close	con	create
chol	closeall	conj	crout
choldn	cls	cons	croutp
cholsol	color	continue	csrcol
cholup	cols	conv	csrlin
chrs	colsf	coreleft	csrtype
cint	comlog	cos	cvtos
clear	compile	counts	

d

date	debug	diag	dos
dbcommit	declare	diagrv	dtvnormal
dbconnect	delete	disable	dtvtoutc
dbdisconnect	det	dlibrary	
dbopen	detl	dllcall	
dbstrerror	dfree	do	

e

ed	else	endp	error
edit	elseif	envget	errorlog
editm	enable	eof	exec
eig	end	eq	exp
eigh	endfor	eqv	external
eighv	endif	erf	eye
eigv	endo	erfc	

f

fcheckerr	fgetsat	fn	fputs
fclearerr	fgetst	font	fputst
fflush	fileinfo	fontload	fseek
fft	files	fontunload	fstrerror
ffti	filesa	fontunloadall	ftell
fftn	fix	fopen	ftocv
fgets	floor	for	ftos
fgetsa	fmod	format	

g

gamma	getname	goto	gt
ge	getnamef	graph	
getf	gosub	graphsev3	

h

hasimag	hess	hsec

i

if	indexcat	inv	iscplx
imag	indnv	invpd	iscplxf
indcv	int	invswp	ismiss

k

key	keyw	keyword

l

le	loadexe	locate	lt
let	loadf	log	ltrisol
lib	loadk	lower	lu

library	loadm	lpos	lusol
line	loadp	lprint	
ln	loads	lpwidth	
load	local	lshow	

m

matrix	meanc	miss	msym
maxc	minc	missrv	
maxindc	minindc	moment	

n

ndpchk	ndpcntrl	new
ndpclex	ne	not

o

oldfft	ones	openpqg	output
oldffti	open	or	outwidth

p

packr	plot	prcsn	proc
parse	plotsym	print	prodc
pdfn	pop	printdos	push
pi	pqgwin	printfm	

r

rankindx	return	rndcon	rotater
rcondl	rev	rndmod	round
readr	rfft	rndmult	rows
real	rffti	rndn	rowsf
recserar	rfftip	rndns	run

recsercp
reshape
retp
rfftn
rfftnp
rfftp
rndseed
rndu
rndus

s

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

t

tab
tan
tempname
time
timeutc
trace
trap
trapchk
trim
trimr
trunc
type
typecv
typef

u

union
uniqindx
unique
until
upper
use
utctodtv
utrisol

v

vals	varput	vec	vfor
varget	varputl	vech	
vargetl	vartypef	vecr	

w

while	wingetcolorcells	winrefresh	winsetforeground
winclear	wingetcursor	winrefresharea	winsetrefresh
winclreararea	winmove	winresize	winsettextwrap
winclearttylog	winopenpqg	winsetactive	winwrite
winclose	winopentext	winsetbackground	winzoompqg
wincloseall	winopentty	winsetcolor	writer
winconvertpqg	winpan	winsetcolorcells	
wingetactive	winprint	winsetcolormap	
wingetattributes	winprintpqg	winsetcursor	

x

xor	xpnd

z

zeros

Singularity Tolerance Appendix C

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.

Reading and Setting the Tolerance

The tolerance value may be read or set using the **sysstate** function, cases 13 and 14.

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.)

Index

Operators and Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R