GAUSS 17™

User Guide

GAUSS

Expanding Probabilities

Windows is a registered trademark of Microsoft Corporation.

Other trademarks are the property of their respective owners.

The Java API for the GAUSS Engine uses the JNA library. The JNA library is covered under the LGPL license version 3.0 or later at the discretion of the user. A full copy of this license and the JNA source code have been included with the distribution.

Version 17
Revision Date: 12/7/2016

Table of Contents

1 Introduction

1.1 Product Overview

The **GAUSS Mathematical and Statistical System (GAUSS)**^{TM} 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.

1.2 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."**
italics	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`	filename, path, etc.	"...is located in the `examples` subdirectory..."
`monospace bold`	reference to a **GAUSS** command or other programming element within a narrative paragraph	"...as explained under **`plotScatter`**..."
Bold Text	reference to section of the manual	"...see **Operator Precedence**, Section 10.7 ..."

2 Getting Started with GAUSS

This section will guide you through the following tasks:

- [Entering Interactive Commands](#)
- [Configuring the User Interface](#)
- [Running an Example Program](#)

2.1 Entering Interactive Commands

Open **GAUSS** and click on the *Command* tab on the left side of the **GAUSS** user interface. This will take you to the *Command Page*, which contains a command history window and a large area for the *Program Input/Output Window*.
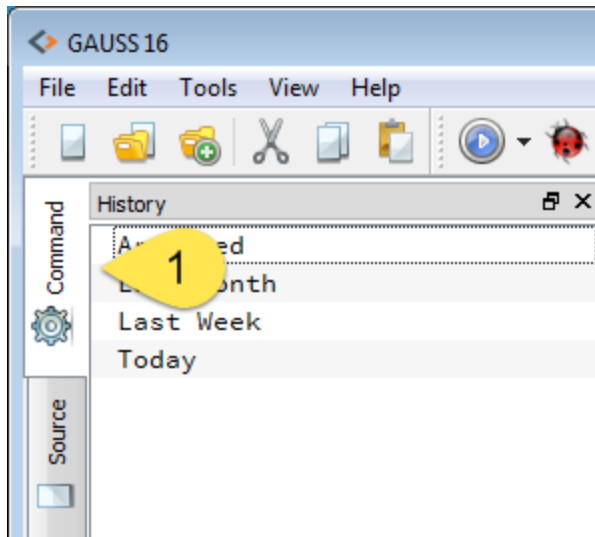
Command Tab

The *Program Input/Output Window* allows you to enter interactive commands and is the location to which your program files will also send their printed output.
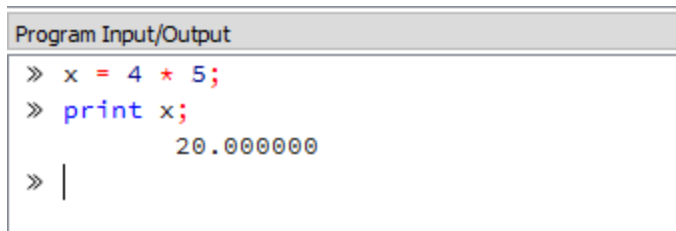
Let's start by entering some simple commands. First type:

x = 4 * 5;

and then press the 'Enter' key to execute the command. The above command will create a 1x1 matrix that is equal to 20. We can print out the contents of x with the print command like this:

```
print x;
```

You should now see that **GAUSS** printed out the value of x which should be 20.

Semi-colon. If you were paying close attention, you may have noticed that both statements above were ended with a semi-colon. In **GAUSS**, a semi-colon is what separates statements. If you are entering one command at a time, semi-colons are not strictly required. However, if you would like to enter multiple commands, or run a program file, you will need to separate each statement with a semi-colon.

2.2 Configuring the User Interface

We will now move to the **GAUSS** *Source Page*. The *Source Page* is where you will likely spend most of your time in **GAUSS**. It allows you to open, edit and run **GAUSS** program files as well as keep track of different folders and libraries. Click the "*Source*" tab on the left side of the **GAUSS** user interface to go to the *Source Page*.
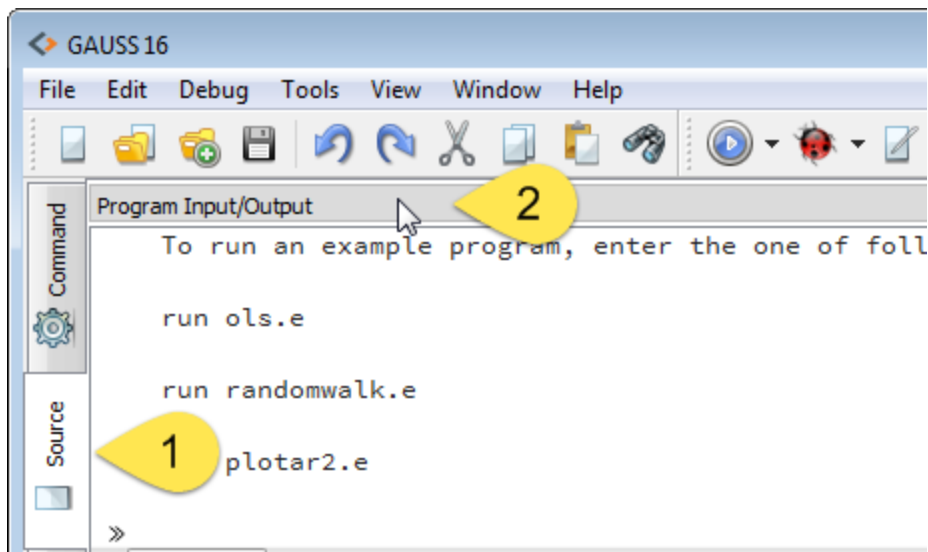
The first thing that we will do is to move *the Program Input/Output Window*. Most of the windows in the **GAUSS** user interface are '*Dock Widgets*'. *Dock Widgets* are windows that can be dragged to different locations in the main application.

We will move the *Program Input/Output Window* from the top of **GAUSS** to the bottom by:

1. Clicking and dragging the *Program Input/Output Window* title bar toward the bottom of the screen.
2. Waiting until a new area with a blue background opens up for the *Program Input/Output Window* to land.
3. Letting go of the mouse button to allow the *Program Input/Output Window* to land in the newly opened area.
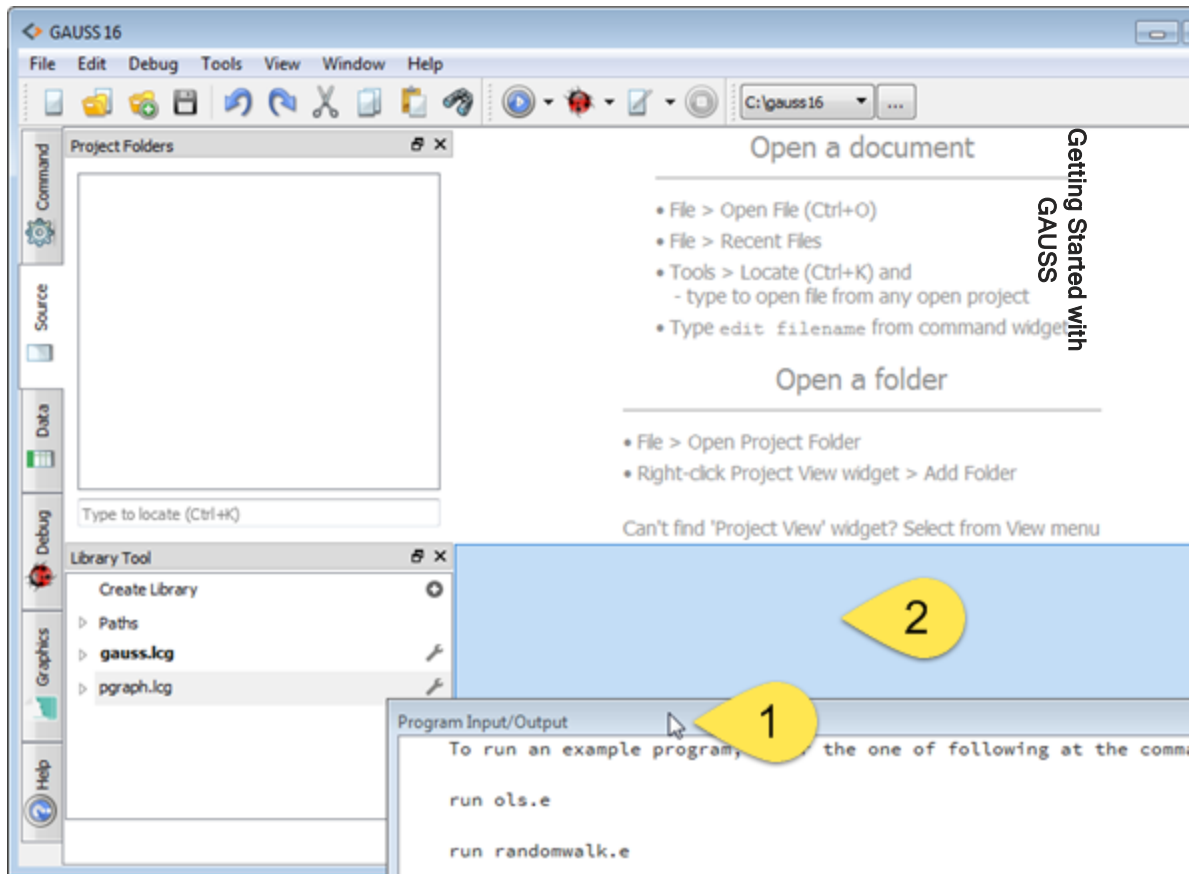
If you have trouble with this, the key is to move slowly and patiently, rather than making quick jerky motions. Do not let go of the window title bar until you are sure that the correct location has opened for you.

2.2.1 Helpful Hints:

1. You can close any window in the **GAUSS** user interface by clicking the 'x' icon in that window's title bar.

2. You may reopen any window that you have closed by selecting it from the '*View*' menu located at the top of GAUSS in the main menu bar.

2.3 Running an Example Program

We will stay on the **GAUSS** Source Page for this section. Our first task will be to open the **GAUSS** examples folder in the *Project Folders Window*. If the *Project Folders Window* is not open, click the '*View*' menu at the top of the GAUSS user interface and select '*Project Folders*' to open the *Project Folders Window*. Once it is open, right-click in an empty portion of the *Project Folders Window* and select '*Add Folder*'.
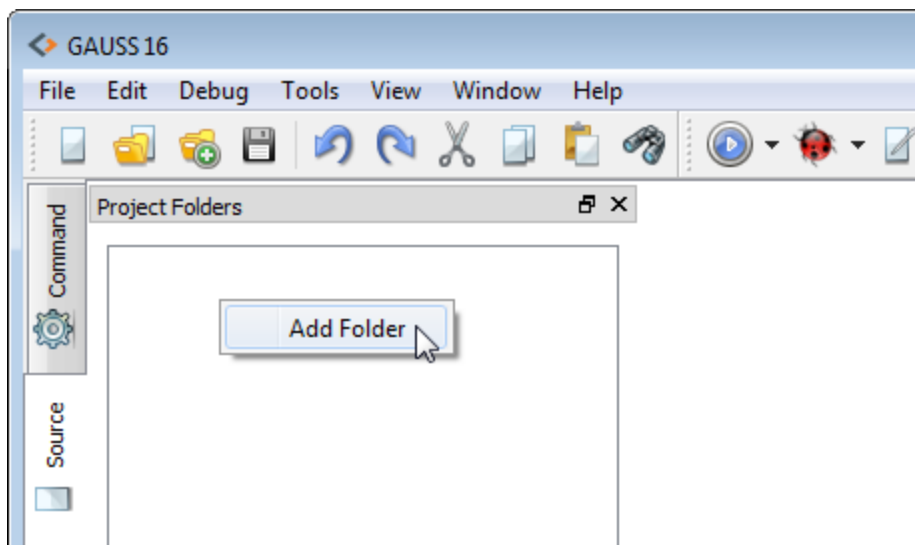
This will open a system file browser (Windows Explorer on Windows, or Finder on Mac). Once this is open, browse to the folder in which you installed **GAUSS**, click the 'examples' folder and then select the '*Open*' button.
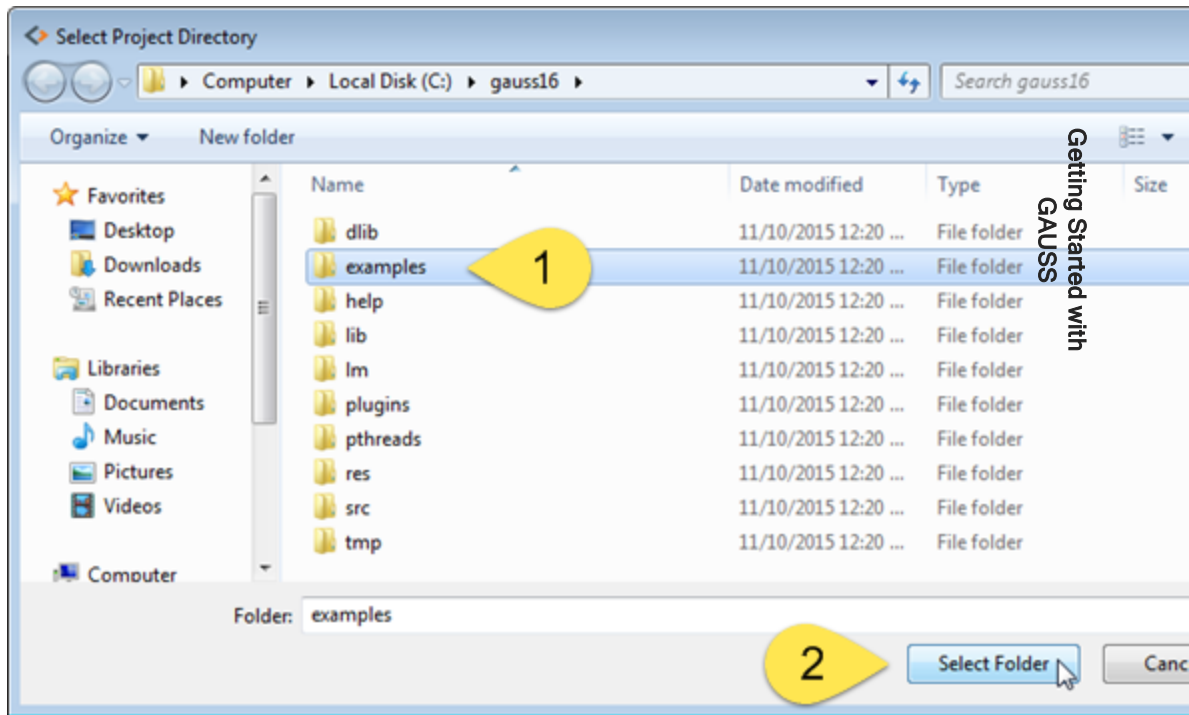
Our next step will be to set our **GAUSS** working directory to the examples directory. Right-click over 'examples' in the *Project Folders Window* and select '*Set to Working Directory*'. Notice that this directory is now listed in the **GAUSS** toolbar.
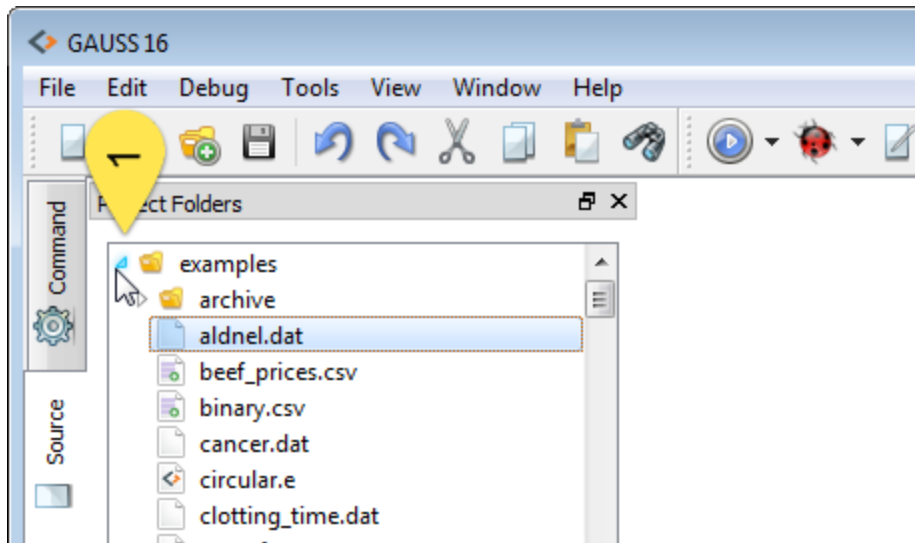
View the contents of this folder by expanding the 'examples' node (click on the triangle to the left of 'examples'). Next scroll down until you see 'glmgamma1.e' and double-click it to open the file.
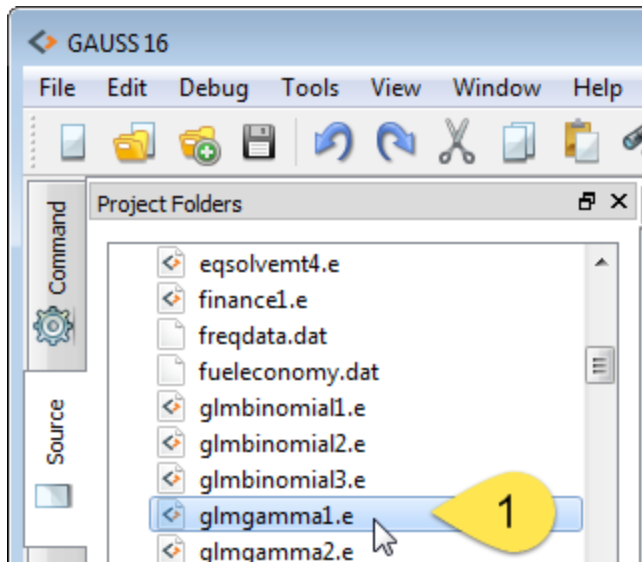
Once the file is open, we can run it by clicking the downward pointing triangle next to the '*Run*' button on the toolbar and selecting '*Run current file*'.
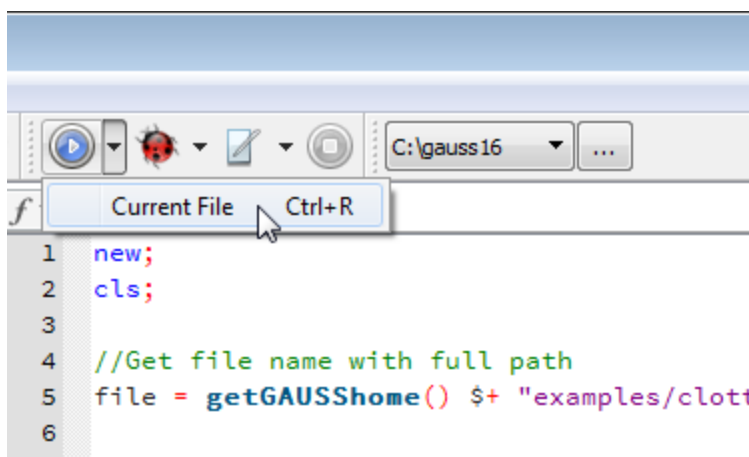
After this step, you should see a results table in the *Program Input/Output Window*. Please try and run other example files. The names of all example files that draw graphs start with 'plot'.
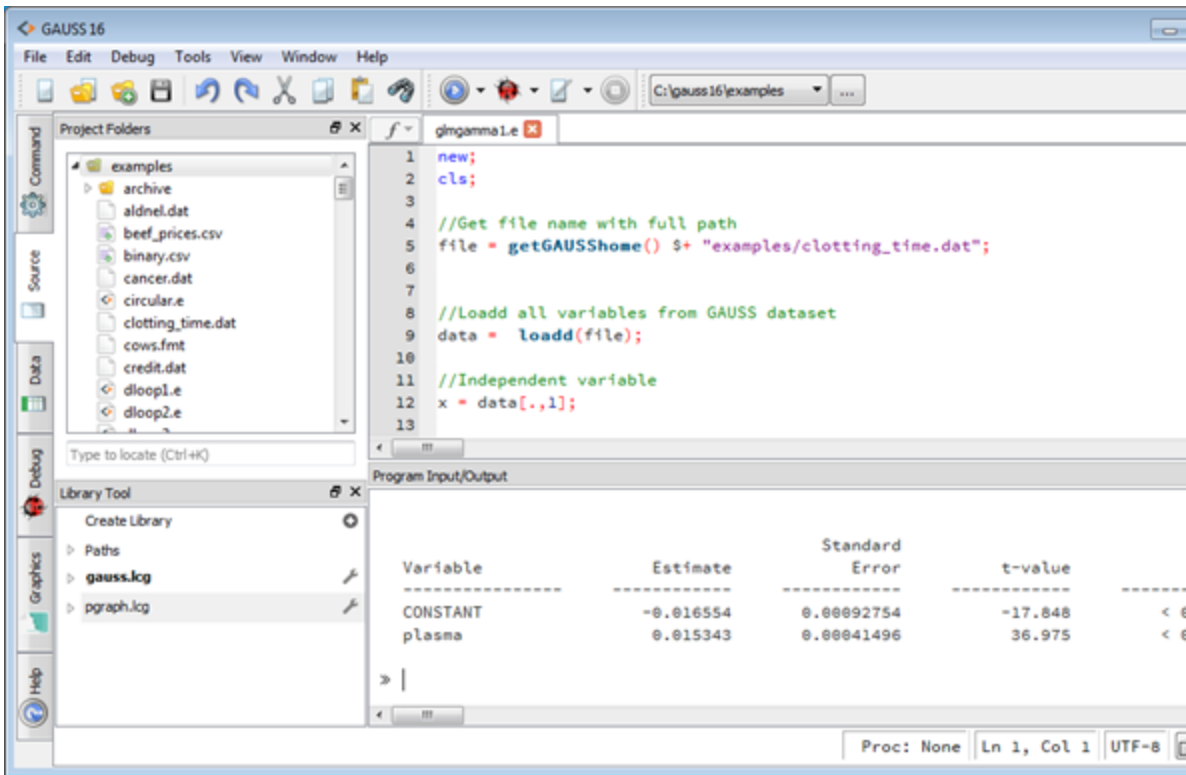
3 Introduction to the GAUSS Graphical User Interface

3.1 Page Organization Concept

GAUSS graphical user interface is organized into six separate "pages," each designed to facilitate the performance of one of the common tasks performed in **GAUSS**:

Command Page	For executing interactive commands.
Source Page	For editing program files.
Data Page	For examining and editing **GAUSS** matrices and other data.
Debug Page	For interactively debugging your programs.
Graphics Page	For viewing and interacting with graphs.
Help Page	For accessing the **GAUSS** help system.

Pages are separate, customizable, main windows with their own set of widgets. Each page is shown as a tab on the left-hand side of the main application, allowing you to instantly access a window custom-configured for the task you wish to perform.
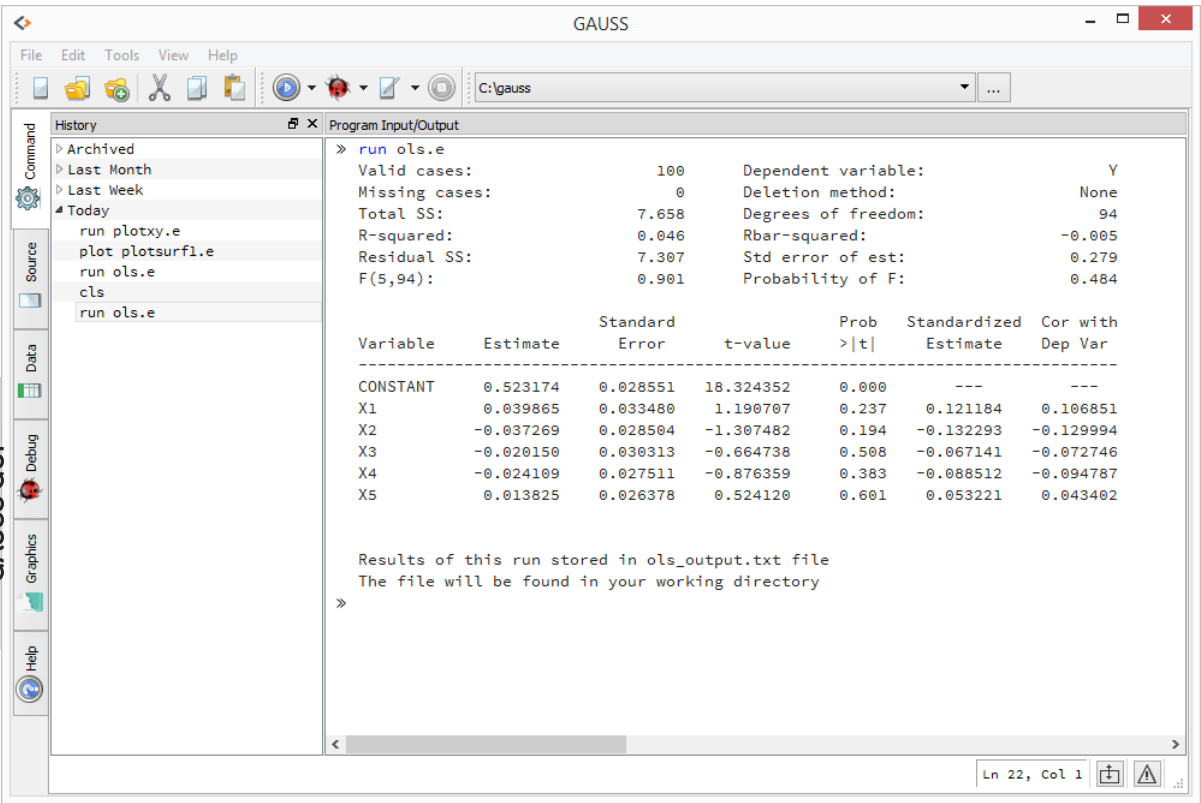
Figure 3.1: **GAUSS** page tabs along the left side of the application

Pages may be undocked from the main application and redocked by toggling the Dock button on the right side of the status bar. Navigation between an undocked page and the main application may be accomplished with ALT+TAB and ALT+SHIFT+TAB. To navigate between pages, use CTRL+TAB to cycle forward and CTRL+SHIFT+TAB to cycle backward.

Each page has its own toolbars and menus, which perform desired functions and facilitate intuitive navigation through the GUI. For example, clicking the " New" toolbar but-

ton from any page in the GUI will bring the Source Page to the top of the window stack with a new file open and ready for editing.

3.2 Command Page

The Command Page is designed for entering interactive commands and viewing results.
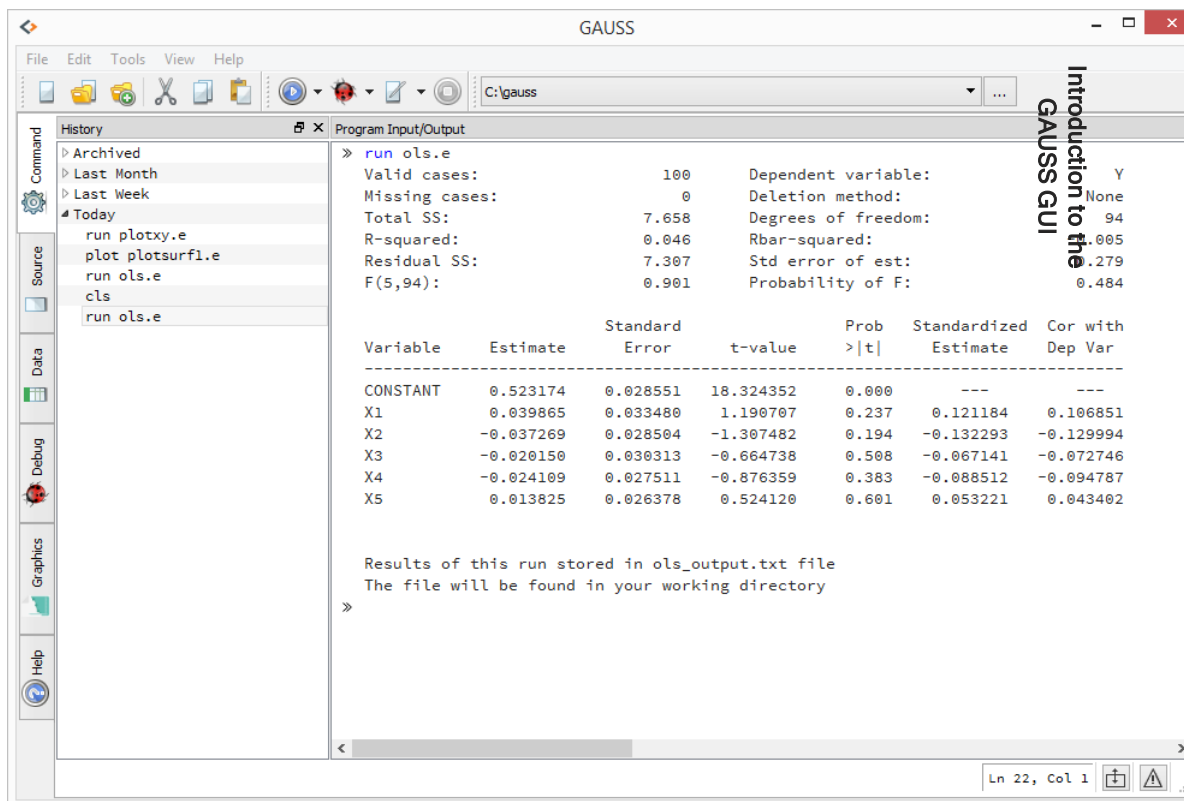
Figure 3.2: Command Page

3.2.1 Command Page Menus

File Menu

New	Creates a new, untitled file in a programming editor on the Source Page.
Open File	Opens an existing file in a programming editor on the Source Page.
Open Project Folder	Adds a folder and its contents to the **GAUSS** project window on the Source Page.
Open Graph	Opens a previously created **GAUSS** graph (.plot file).
Print	Prints selected text.
Recent Files	Contains a dropdown list of recently edited files.
Exit	Exits a **GAUSS** session.

Edit Menu

Undo	Restores your last unsaved change.
Redo	Re-inserts changes removed with undo.
Cut	Removes selected text and copies it to the clipboard.
Copy	Copies selected text to the clipboard.
Paste	Pastes the clipboard contents at the cursor position.
Search	Opens a search pane for the Program Input/Output window.
Advanced Search	Opens the Source Browser.

Tools Menu

Preferences	Allows you to configure the **GAUSS** user environment.
Install Application	Opens an installation wizard to install a **GAUSS** application package.
Change Font	Allows you to specify a new font. Aptech recommends using a monospaced font such as Courier.
Clear Action List	Clears the recent files from the action list menus.
Change Working Directory	Allows you to browse for a new working directory.
Clear Working Directory History	Deletes the contents of your recent working directory history.
Recent Working Directories	Contains a dropdown list of your most recent working directories.

View Menu

The View menu lets you toggle on or off the windows on the current page.

Help Menu

Goto Help	Takes you to the Help Page.
About GAUSS	Provides information regarding your version of **GAUSS** .

3.2.2 Command Page Toolbar

Figure 3.3: Command Page Toolbar

New	Opens a new, untitled document in a programming editor on the Source Page and brings you to the Source Page.
Open	Opens an existing file for editing.
Open Folder	Adds a folder and its contents to the GAUSS project window on the Source Page.
Cut	Removes selected text and places it on the clipboard.
Copy	Copies selected text to the clipboard.
Paste	Copies the clipboard contents to the cursor position.
Run	Runs the file at the top of the Action List.
Debug	Debugs the file at the top of the Action List.
Edit	Opens the file at the top of the Action List.
Stop Program	Stops a running **GAUSS** program.

3.2.3 Working Directory Toolbar
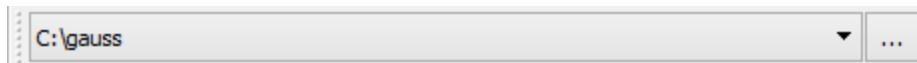
Figure 3.4: Working Directory Toolbar

The Working Directory Toolbar contains a dropdown list that shows your current working directory and a history of recent directories. The Change Working Directory button allows you to browse for and select a new working directory.

3.2.4 *Action List* Toolbar

Figure 3.5: *Action List* Toolbar

Run	Executes a file from the *Action List*.
Debug	Debugs a file from the *Action List*.
Edit	Edits a file from the *Action List*.
Stop	Stops a program that is currently running or being debugged.

The *Action List* is a dropdown list of the files you have most recently performed an action on. Clicking on the *Run*, *Debug*, or *Edit* button will perform the specified action on the file at the top of the *Action List*. Placing your mouse over the button produces a

tooltip with the name of that file. To perform an action on a different file in the *Action List*, click on the downward-pointing triangle immediately to the right of the desired button.

All of the buttons share the same *Action List*. You may add a file to the *Action List* by running it from the command line. Or while editing a file, you can either select **Current File** from the dropdown menu next to the *Run* or *Debug* button or **Add to List** from the *Edit* dropdown menu.

3.2.5 Layout and Usage

The Command Page contains three main widgets: the Program Input/Output Window, the Command History Window, and the Error Output Window.
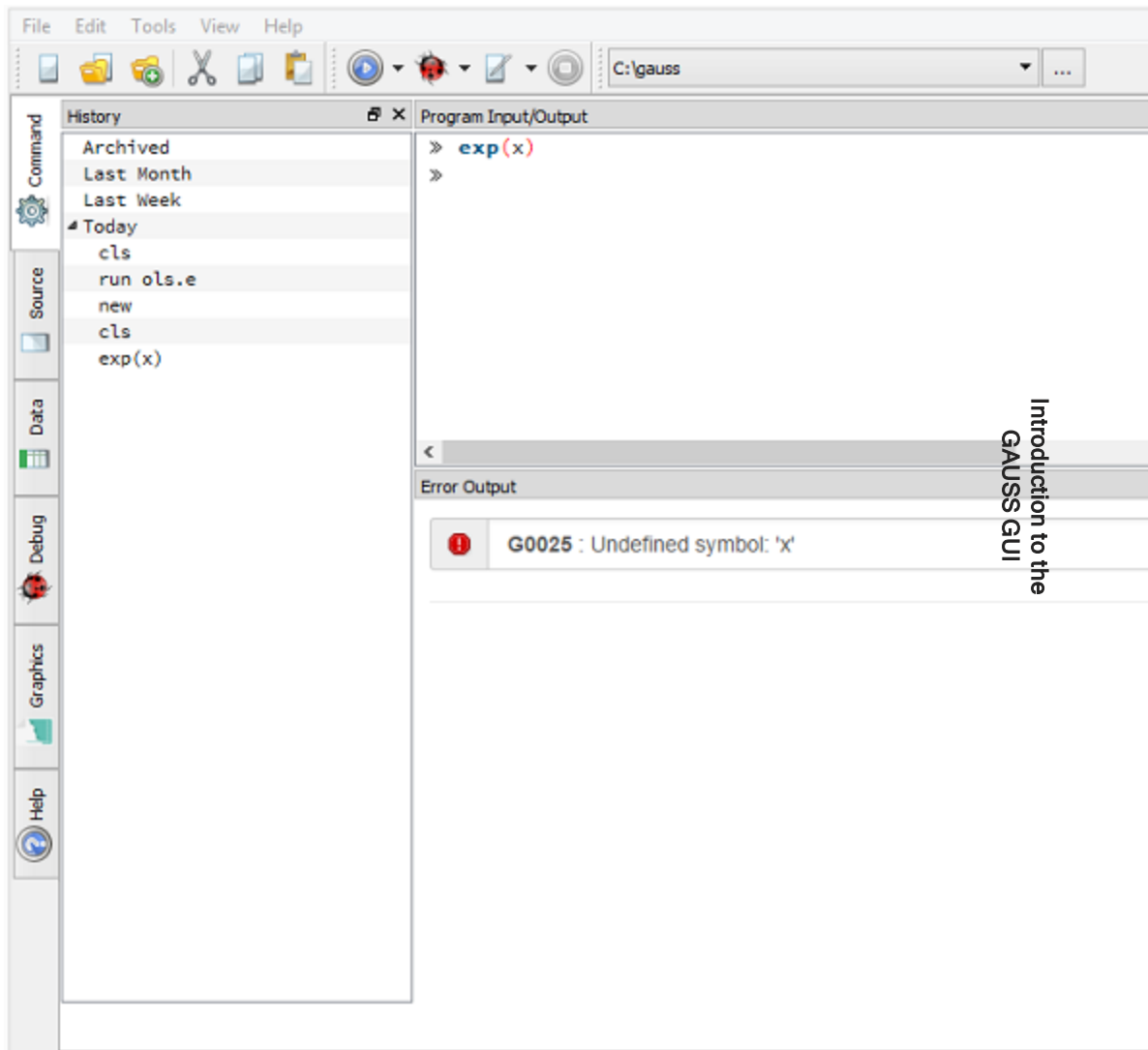
Figure 3.6: Command Page Widgets

3.2.6 Command History Window

The Command History Window contains a list of recently-executed commands organized by date or by **GAUSS** session. To re-run a command, double-click on it or select the command and then press the Enter key.

Context Menu
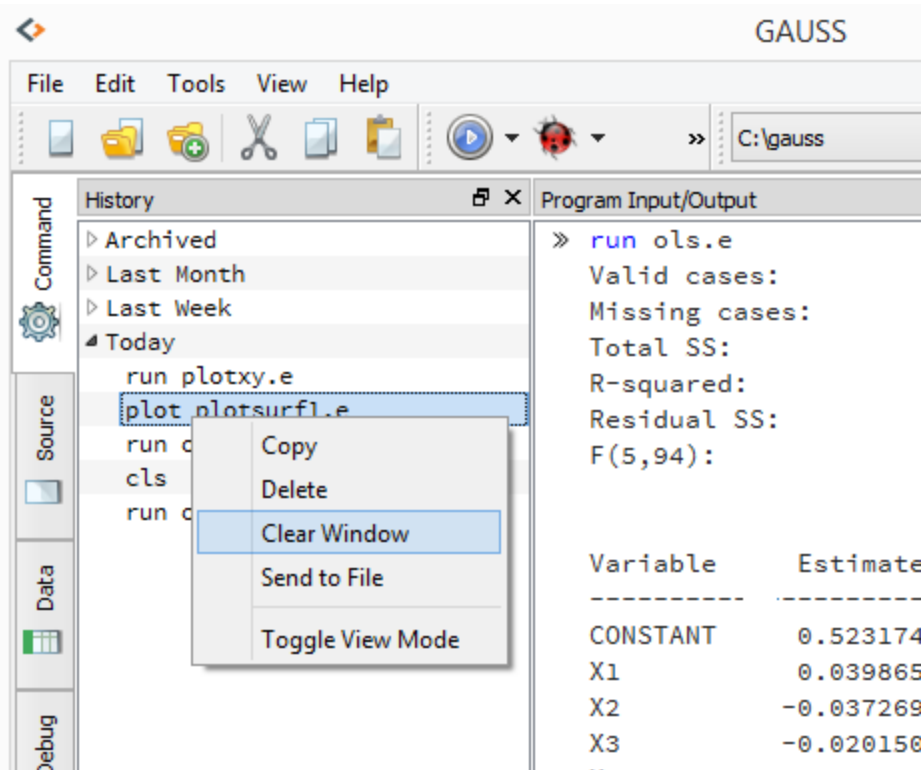
Figure 3.7: Command History

Right-click in the Command History Window to bring up the context menu with the following options:

Copy	Copies the highlighted entry to the clipboard.
Delete	Deletes the highlighted entry.
Clear Window	Deletes all Command History entries.
Send to File	Creates a new source file containing the highlighted commands.
Toggle View Mode	Toggles between date mode and session mode.

3.2.7 Command Line History and Command Line Editing

When you run a command at the **GAUSS** prompt, it is added to your command line history. The last 1,000 commands executed at the **GAUSS** command line are stored. The following keystrokes are supported for movement and editing at the command line and for retrieving the command line history:

Left Arrow	Moves cursor left one character.
Right Arrow	Moves cursor right one character.
HOME	Moves cursor to beginning of line.
END	Moves cursor to end of line.
CTRL+Left Arrow	Moves cursor left one word.
CTRL+Right Arrow	Moves cursor right one word.
Up Arrow	Search up through command history.
Down Arrow	Search down through command history.
CTRL+Space	Activate autocomplete function drop-down list or tooltips.

NOTE: Use CTRL+PAGE-UP or CTRL+PAGE-DOWN to scroll through your program
output in the program input/output window.

3.2.8 Error Output Window

The Error Output Window shows errors messages from program runs or interactive com-
mands and may be viewed from any page by clicking the Error Output button on the right
side of the status bar. However, it will automatically open when an error occurs and will
close and empty its contents on the next program run or interactive command.

The line number shown next to the error message is a clickable link (indicated by blue
text), which will take you to the file and line containing the error.
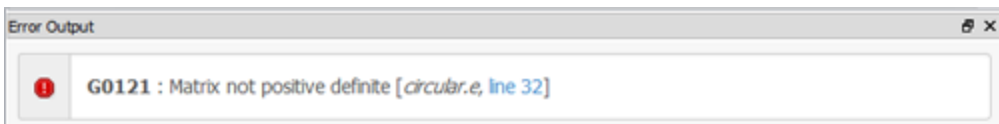
Figure 3.8: Error Output Window

3.3 Source Page

The Source Page is designed for creating and editing programs and procedures.

3.3.1 Source Page Menus

In addition to the menus common to each page (see Section 3.2.1) and the Debug Menu, the Source Page contains the following additional menu options.

File Menu

Close	Closes the selected file.
Close All	Closes all open files.
Save	Saves the active file.
Save As	Saves the active file with a different file or path name.

Window Menu

Split File Buffer Horizontally	Horizontally splits the active file into two views.
Split File Buffer Vertically	Vertically splits the active file into two views.
Remove Buffer Split	Removes a buffer split in the active file.
Split Tab Stack Horizontally	Tiles any open programming editors horizontally.
Split Tab Stack Vertically	Tiles any open programming editors vertically.
Remove Tab Split	Removes any editor window tiling.
Go Back	Takes you to your last cursor position, which could

	be in a different file.
Go Forward	After using Go Back, takes you to the next cursor position, which could be in a different file.
Previous Open Document	Cycles backwards through the list of open tabs.
Next Open Document	Cycles forward through the list of open tabs.

3.3.2 Layout and Usage

The Source Page contains seven separate widgets: the Programming Editor, the Error Output Window, the Program Input/Output Window, the Projects List, the Bookmarks List, the Library Tool, and the Source Browser.
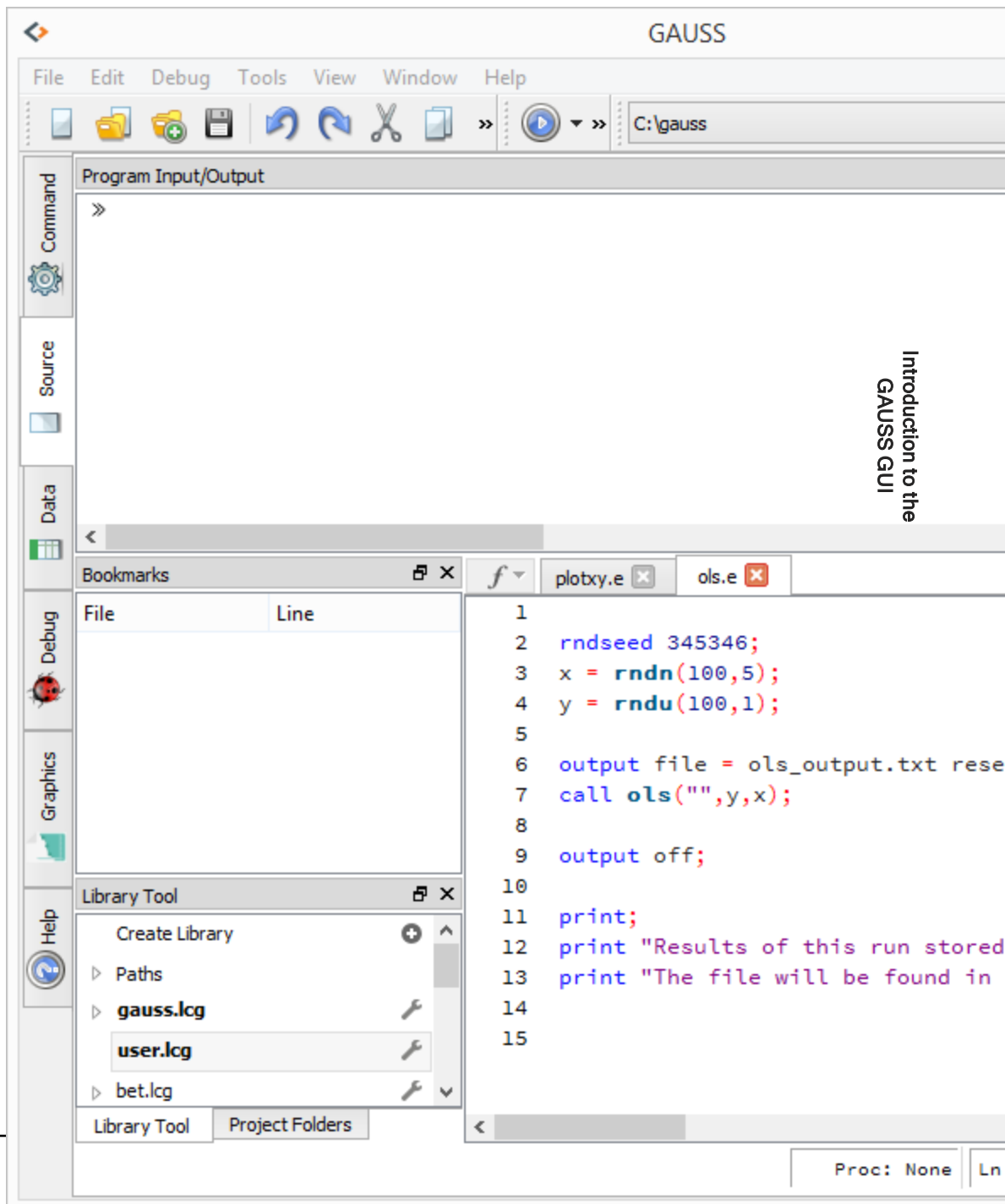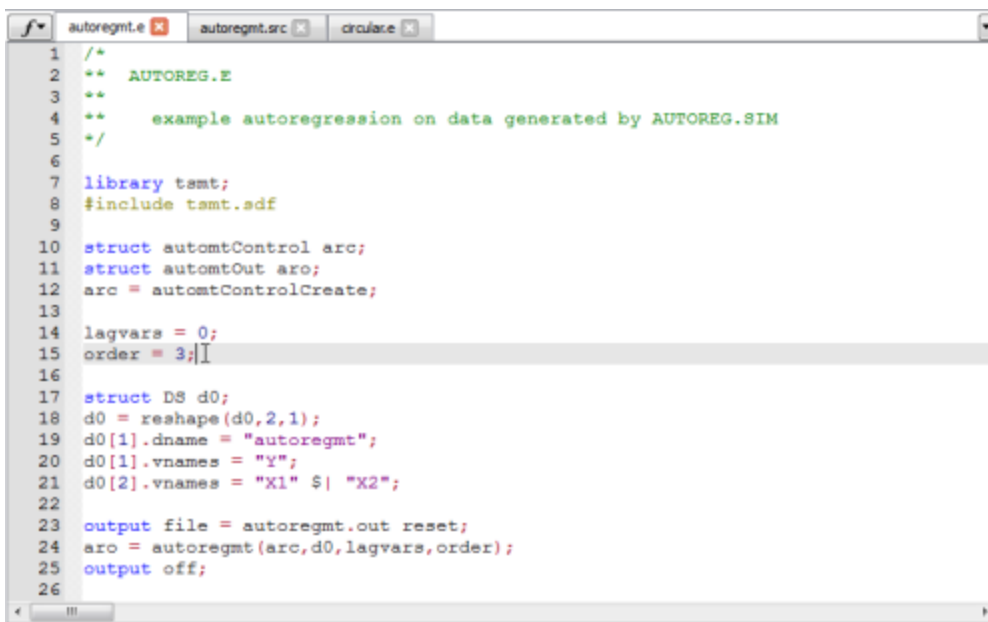
Figure 3.9: Source Page

Programming Editor

Individual programming editors are opened in the editor docking area. The editor docking area allows tabbing of multiple open files, with the option to tile editors with a horizontal or vertical split. Select **Window->Split Horizontally** or **Window->Split Vertically** to tile open editor windows.


```
 1   /*
 2   **   AUTOREG.E
 3   **
 4   **     example autoregression on data generated by AUTOREG.SIM
 5   */
 6
 7   library tsmt;
 8   #include tsmt.sdf
 9
10   struct automtControl arc;
11   struct automtOut aro;
12   arc = automtControlCreate;
13
14   lagvars = 0;
15   order = 3;
16
17   struct DS d0;
18   d0 = reshape(d0,2,1);
19   d0[1].dname = "autoregmt";
20   d0[1].vnames = "Y";
21   d0[2].vnames = "X1" $| "X2";
22
23   output file = autoregmt.out reset;
24   aro = autoregmt(arc,d0,lagvars,order);
25   output off;
26
```

Figure 3.10: Programming Editor

Programming editor features:

1. **Syntax highlighting:** The **GAUSS** programming editor will provide syntax highlighting for **GAUSS**, C/C++, Java, Fortran, R and many other languages.

2. **Autocompletion:** Autocompletion is available in the **GAUSS** programming editor for **GAUSS** functions and for structure members of any structure in an active GAUSS library.
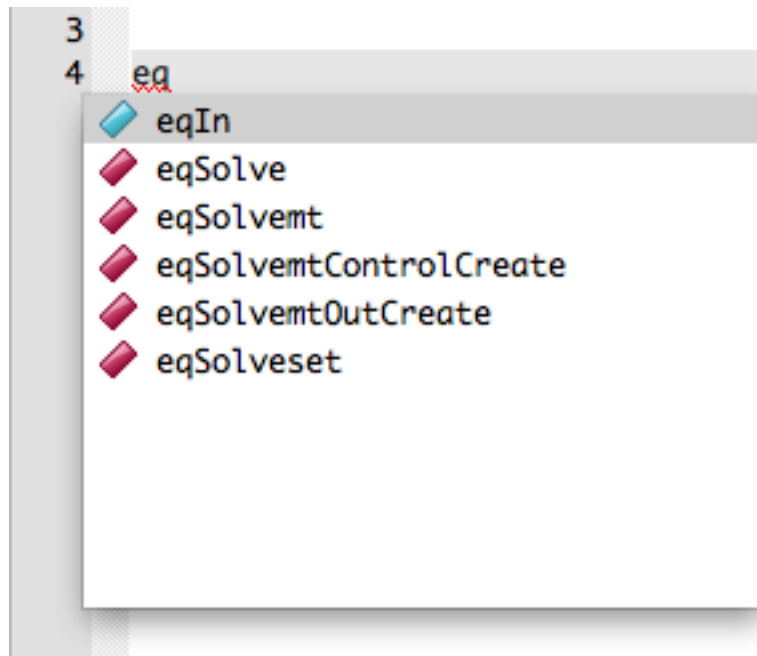
Figure 3.11: Autocomplete

Using autocomplete: if the characters you enter match items in the autocomplete list, a dropdown box will appear containing those functions. To navigate the dropdown list, press the down arrow or continue typing until only one selection

remains. Once the desired command is highlighted, press the ENTER key to insert the remainder of the word.

The editor will provide autocomplete for the members of any structures that are in a GAUSS library if: the library is loaded, the file contains a statement to load the library, or the structure definition is in the current file.

3. **Function Call Tooltips:**After a **GAUSS** command and an opening parenthesis has been entered, a tooltip will appear with the argument list for the function.
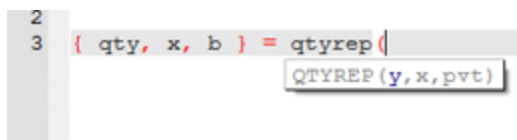
Figure 3.12: Tooltips

4. **Function Browser:**Dropdown list lists all procedures defined in your file and lists their arguments. Select a procedure from the list to navigate to its definition.
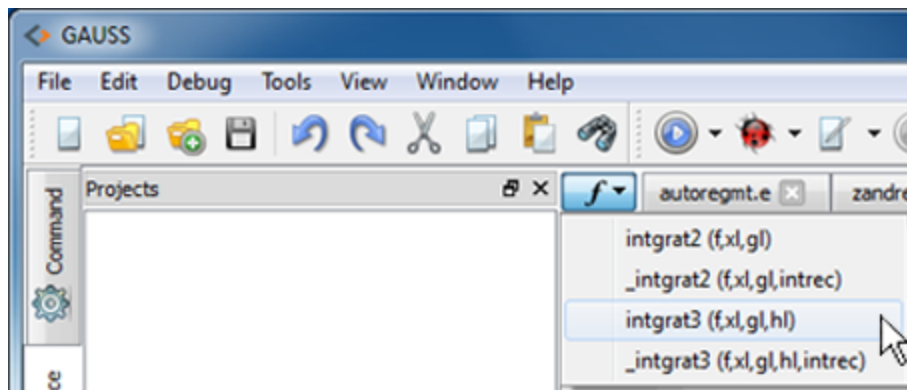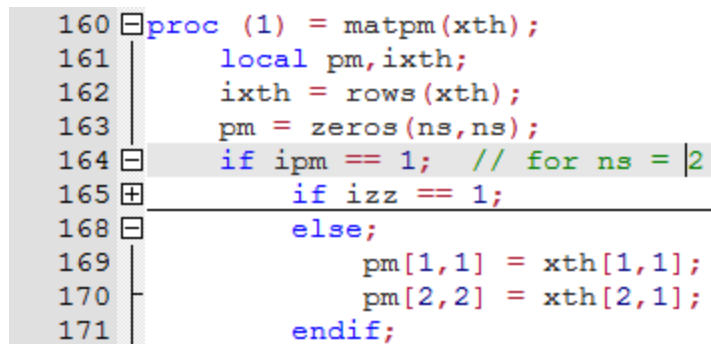
Figure 3.13: Function Browser

5. **Code folding:**At the start of code blocks (e.g., procedure definitions, `do` and `for`

loops, and `if` statements), the left margin of the programming editor will contain
a +. Clicking the + will hide the block of code from view and place a horizontal
line across the editor indicating folded code and changing the + to a -. Clicking on
the - will reveal the hidden code.

```
160 ⊟proc (1) = matpm(xth);
161  │      local pm,ixth;
162  │      ixth = rows(xth);
163  │      pm = zeros(ns,ns);
164 ⊟      if ipm == 1;   // for ns = 2
165 ⊞          if izz == 1;
168 ⊟          else;
169  │              pm[1,1] = xth[1,1];
170  ├              pm[2,2] = xth[2,1];
171  │          endif;
```

Figure 3.14: Code folding of 'if' block

6. **Autoindenting:** The **GAUSS** programming editor provides automatic code indent-
   ing and deindenting. Autoindenting not only simplifies the process of writing code
   but also encourages the creation of readable code.

7. **Variable highlighting:**Click on a variable name and all instances of that variable
   will be highlighted.

Figure 3.15: Variable highlighting

8. **Find usages:** Right click on a variable or procedure and select 'Find Usages' from the context menu to create an organized clickable list of all usages in your file.
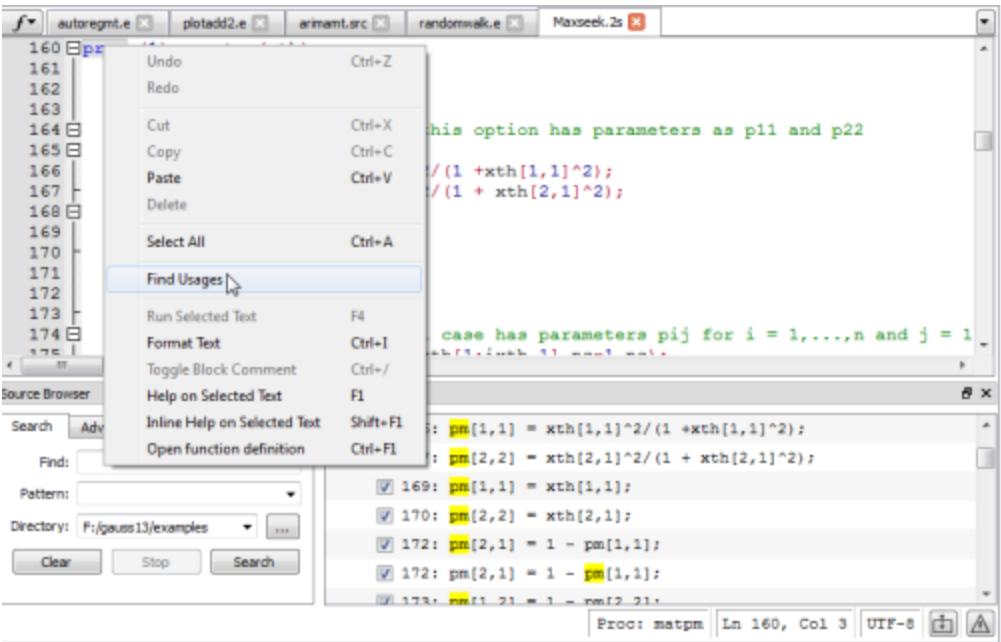
Figure 3.16: Find usages

9. **Bookmarks:**Bookmarks allow quick navigation to often visited lines of code. To insert a bookmark, hold down the SHIFT key and left click in the left margin of the file on the line where you would like to set the bookmark.

   If a file contains more than one bookmark, you may use F2 to navigate to the next bookmark in the file. To navigate between bookmarks in multiple files, use the bookmark window. The bookmark window contains a list of all of your bookmarks.

   Double-click on a listed bookmark and **GAUSS** will bring you to the file and line of your bookmark regardless of whether the file is already open for editing or not.

10. **Source code formatting:** The **GAUSS** editor will correctly format a source file with incorrect formatting. This is available from the context menu or with the hot key CTRL+I.

Programming Editor Hot Keys

CTRL+A	Select All.
CTRL+C	Copy.
CTRL+D	Debug current file.
CTRL+F	Find and replace.
CTRL+G	Go to line.
CTRL+I	Formats the current file.
CTRL+L	Delete line.
CTRL+N	Open new file.
CTRL+O	Open existing file.
CTRL+P	Print file.
CTRL+/	Used for block commenting.
CTRL+R	Run current file.
CTRL+S	Save current file.

CTRL+T	Switches current line with the line above.	
CTRL+V	Paste.	
CTRL+W	Closes the current file.	
CTRL+Z	Undo.	
CTRL+Y	Redo.	
CTRL+~	Cycles through open editor windows.	
CTRL+SHIFT>	Activates function autocomplete or tooltips, depending upon the context.	
CTRL+F1	Go to definition of function under cursor.	
F1	Go to Help Page for function under cursor.	
SHIFT+F1	Open inline help for function under cursor.	
ALT+Left-Arrow	Go to previous cursor position, which may be in a previous file	
ALT+Right-Arrow	After ALT+Left-Arrow, go to next cursor position.	

3.3.3 Find and Replace

From the Edit Menu, selecting **Find and Replace** or pressing CTRL+F will bring up the find and replace widget at the bottom of your open programming editor. If a word is highlighted when you access find and replace, it will automatically be present in the find box when the find and replace widget is opened. Press the ENTER key or > to search forward. Press the < button to search backwards. To close the find and replace widget, press ESC or click the button on the left.
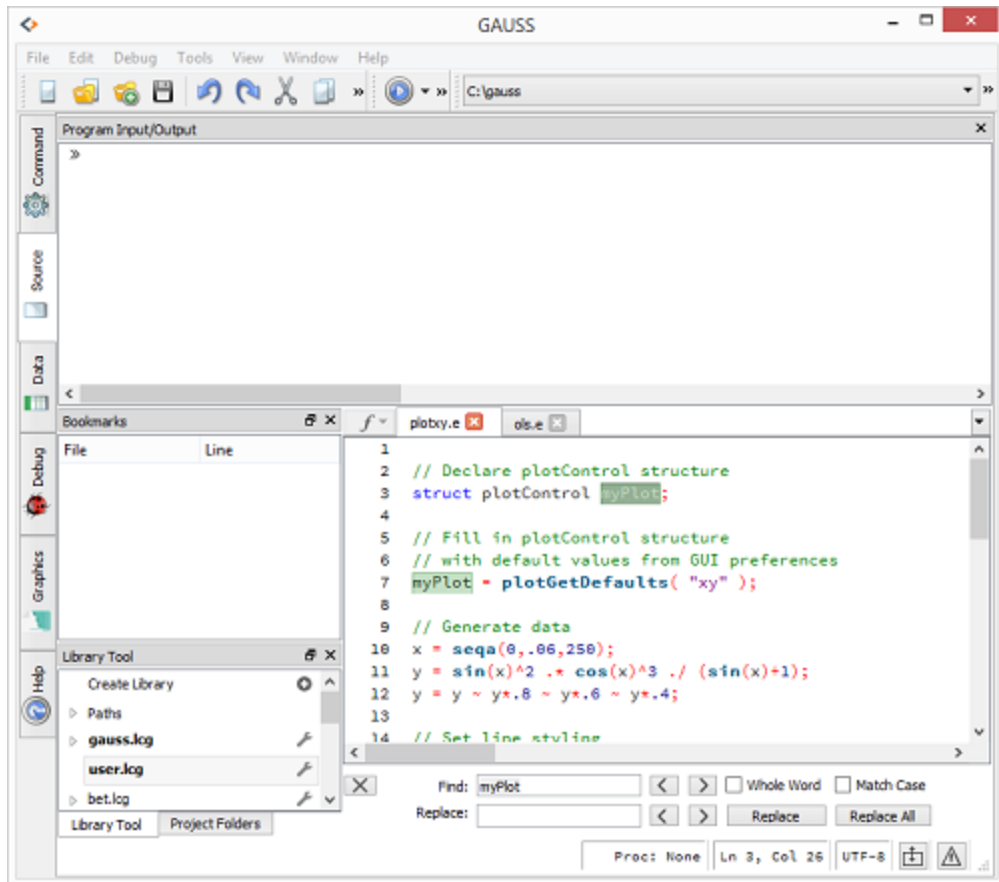
Figure 3.17: Find and Replace

The Replace Box has three buttons: > means replace the highlighted expression and
search forwards, < means replace the highlighted expression and search backwards and
replace means replace the highlighted text and do not change the cursor position.

Regular Expressions

Find and Replace in **GAUSS** supports regular expression searching. Regular expression searching gives users tremendous power allowing quick and precise search and replace throughout an entire file. For example, let us start with a file containing the following commands:

```
r = 100;
c = 50;
x = rndn(r,c);
y = rndu(r,c);
z = x.*rndn(r,c);
```

Regular expressions allow you to perform very specific find and replace commands. Suppose that we want to find all usages of **rndu** and **rndn** and replace them with **rndKMu**.
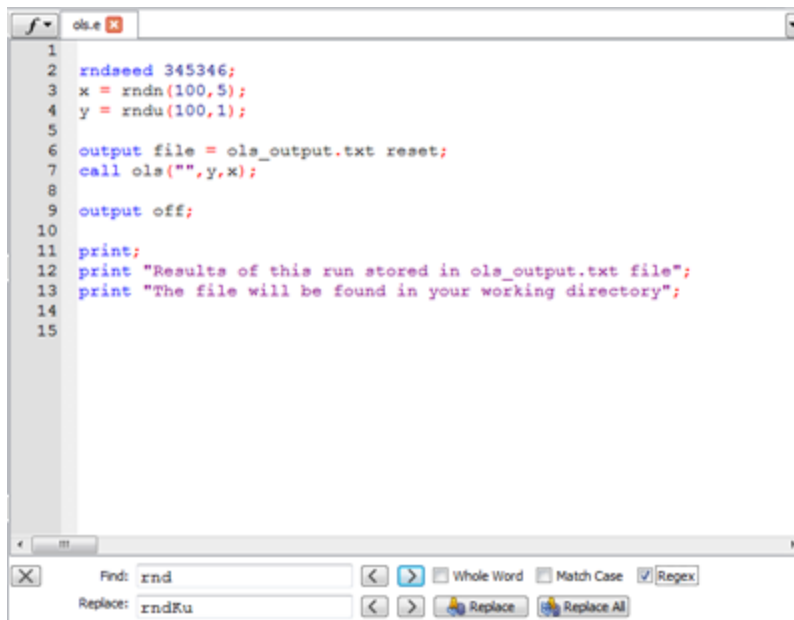
Figure 3.18: Find and Replace Regular Expression

To open **Find and Replace**, we enter CTRL+F in out open text editor. In the Find and Replace widget, select the check box next to Regex to enable regular expression searching. One of the most simple regular expression options is to add a '.'. The '.' means any character. So, if we search for "**rnd.**" that will find any string that contains **rnd** followed by any character, such as **rnda**, **rndb**, **rndc**, **rndn**, **rndu**, etc. Now enter "**rndKMu**" in the replace box and click *Replace All*. Now all instances of **rndu** and **rndn** should be replaced with **rndKMu**.

3.3.4 Changing Editor Properties

Programming editor preferences can be accessed by selecting: **Tools->Preferences** from the menu bar. From the Preferences window, select **Source** from the tree on the left. Here you can customize the programming editor's behavior.
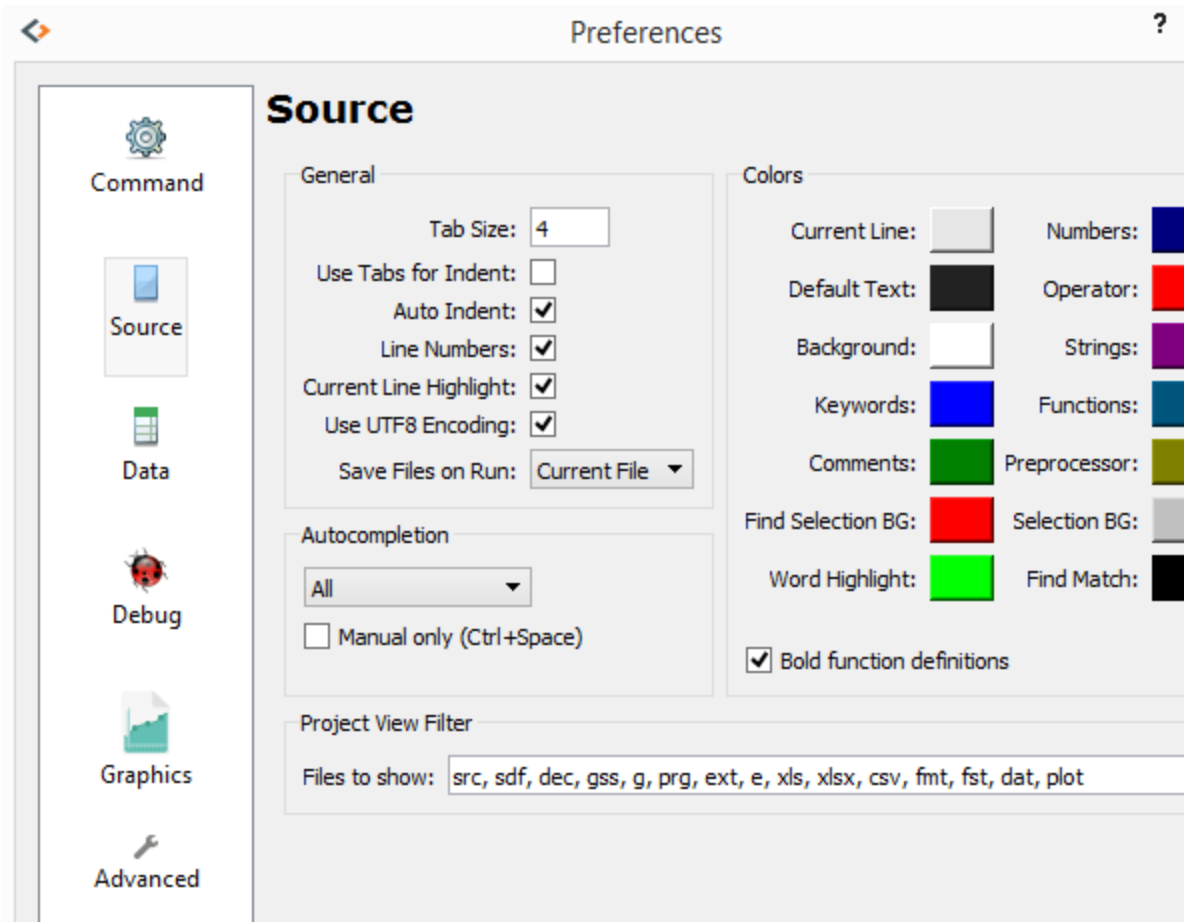
Figure 3.19: Editor Preferences

The editor preferences has three main sections:

General Settings

The general settings are located on the top left.

Colors

This section is located along the bottom half of the preferences dialog window. Click one of the color buttons to change an element in the color scheme used in the editor.

Autocomplete Settings

On the top right of the Source Preferences, under the heading "Autocompletion Source" , are the controls for autocompletion settings. This allows you to choose the contents of your dropdown autocomplete suggestions. The options are:

GAUSS Library	Provides autocomplete suggestions for **GAUSS** intrinsic functions.
Current Document	Provides autocomplete suggestions for all variables, procedures, **GAUSS** functions and keywords that are used in the file.
All	Provides autocomplete suggestions for both above.
None	Provides autocomplete suggestions only when CTRL+SPACE are pressed.

3.3.5 Error Output Window

The Error Output Window can be accessed by toggling the Error Output button on the right side of the status bar. For details regarding the features and usage of the Error Output Window, see Section 3.2.8 .
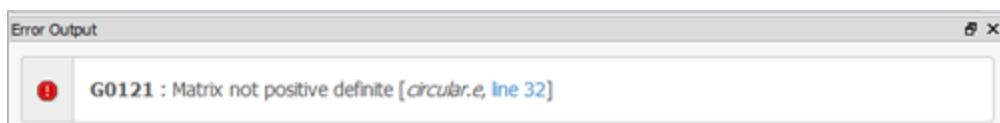
Figure 3.20: Error Output Window

3.3.6 The Source Browser: Advanced Search and Replace

The Source Browser is a powerful search and replace tool that makes it much easier to work with your code--especially when completing larger projects. You may open the Source Browser by selecting **View->Source Browser** from the Source Page or entering the hot-key SHIFT+CTRL+F.
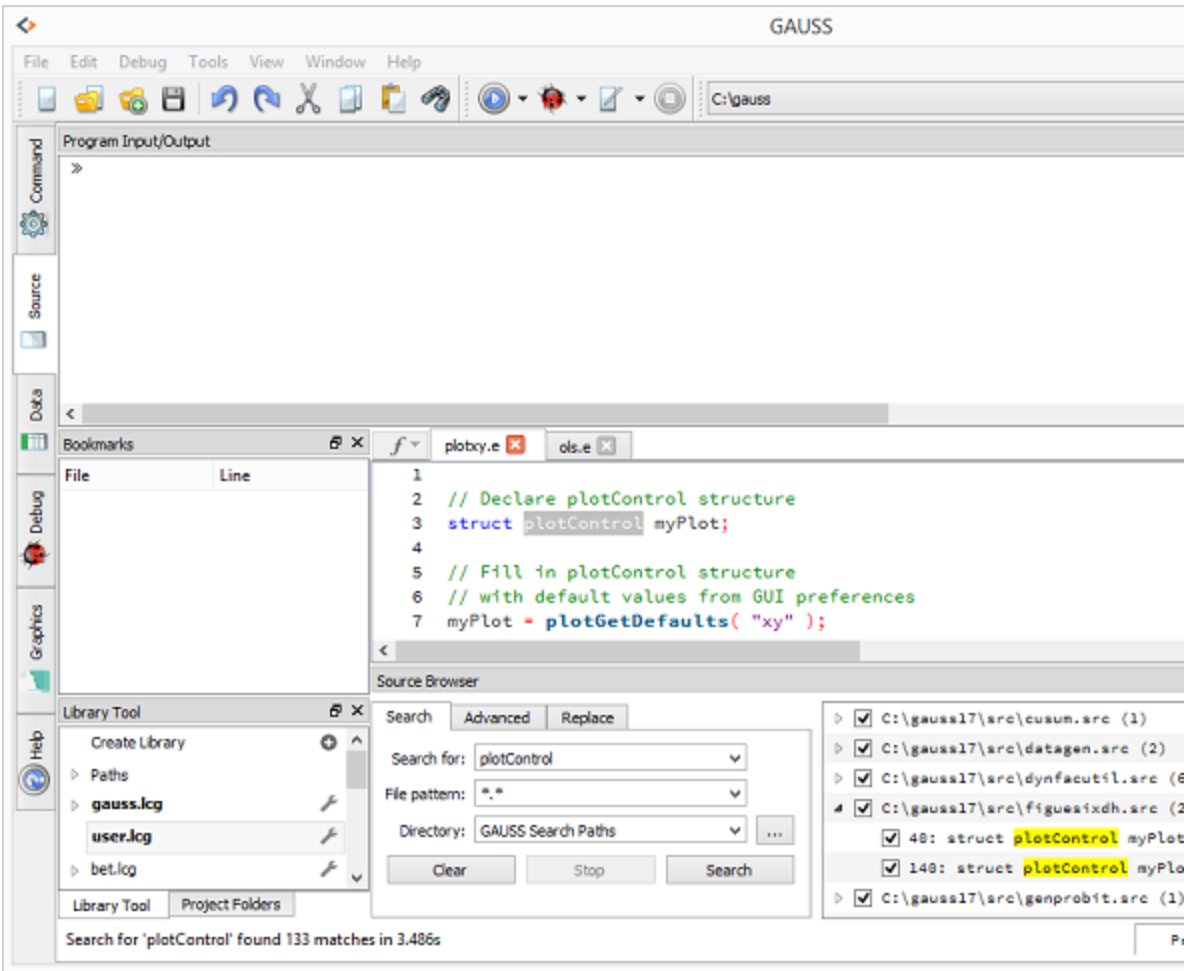
Figure 3.21: Source Browser

The Source Browser is made up of two main sections. The control tabs are on the left and on the results display on the right.

You can enter your search query in the Search tab, which allows you to specify a particular file pattern, filename, or directory to search. The button with three dots [...] to the right of the directory dropdown menu is a browse button that will open a dialog window allowing you to browse for the desired directory.

Search	Advanced	Replace
Find:	rndu	▼
Pattern:	*.src	▼
Directory:	C:/gauss14/src	▼ [...]
Clear	Stop	Search
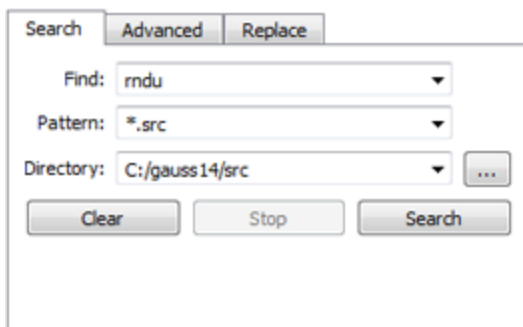
Figure 3.22: Search and Replace

The Advanced tab allows you to refine your search criteria. Checking the References check box will limit your search to only locations where a variable is referenced. Alternatively, checking the Assignments box would limit your search to only those places where the variable was assigned to. For example, if you were searching for the variable 'nobs', this is an assignment:

```
nobs = 300;
```

whereas this is a reference:

```
myVar = nobs - 1;
```
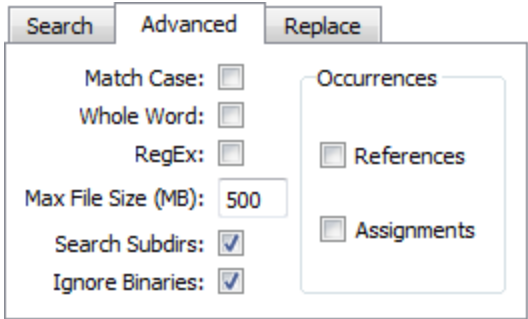
Figure 3.23: Advanced Search and Replace

You may also select to limit your search to whole words, match case or use regular expressions. This tab also allows you to control whether your search extends to sub-directories and if you would like to exclude files larger than a certain size. It is recommended that you always select Ignore Binaries.

Executing a search

Once you have a search string in the Find box and have your desired parameters set, press the 'search' button to execute your search. In the results display window, you will see a list of files that contain matches to your search. To the left of each file in the list you will see a check box which can be used to include or exclude the matches in a file from any Replace actions. To the right of the file name you will see a number in parentheses which indicates how many matches were found in the file.
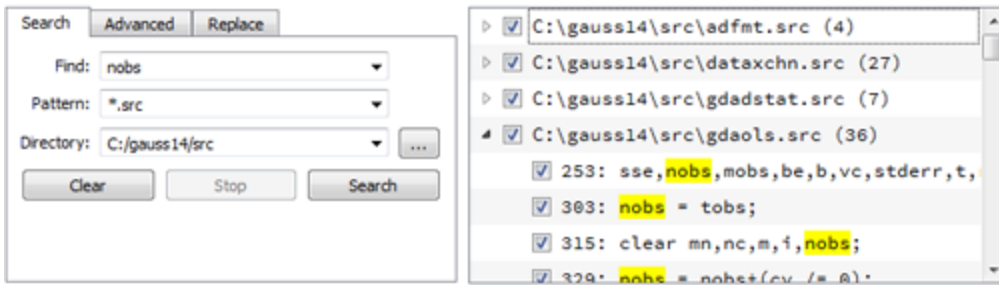
Figure 3.24: Executing a search

Expand the node of a file to examine its list of matches, or right-click and select 'Expand All' from the context menu. Once you expand the node for a file, you will see the line number of the match followed by a colon and then the line of code with the match highlighted in yellow. Double-click on a line to edit that file.

Executing a replace

After you have completed your search and unchecked any matches that you do not want replaced, click the Replace tab. Enter your replacement term and press the Replace button. GAUSS will warn you that any changes in unsaved files cannot be undone. If you are certain of your replacement, click OK.

If you are making a change across a large number of files, particularly if you are not using a version control system like Subversion or Git, it is a good idea to select the 'Backup Original' check box. With this selected, before GAUSS makes the replacements to your file it will save a copy of the original with a `.bak` extension. For example if your original file was `myfile.gss`, the backup file will be called `myfile.gss.bak`.

If you select the 'Overwrite Backup' checkbox, the next time that you make a change to this file, the file `myfile.gss.bak` will be overwritten. If you do NOT check the 'Overwrite Backup' check box, GAUSS will make incremented backup files: `myfile.gss.bak(1)`, `myfile.gss.bak(2)`, etc.

3.3.7 Project Organizer: Keeping track of files with the Project Window

The Project Window displays the contents of any folder you choose as well as its subfolders. To add a folder to the Project Window, either right-click in the Project Window and select "Add Folder" from the bottom of the context menu or click the "Open Folder" toolbar button.
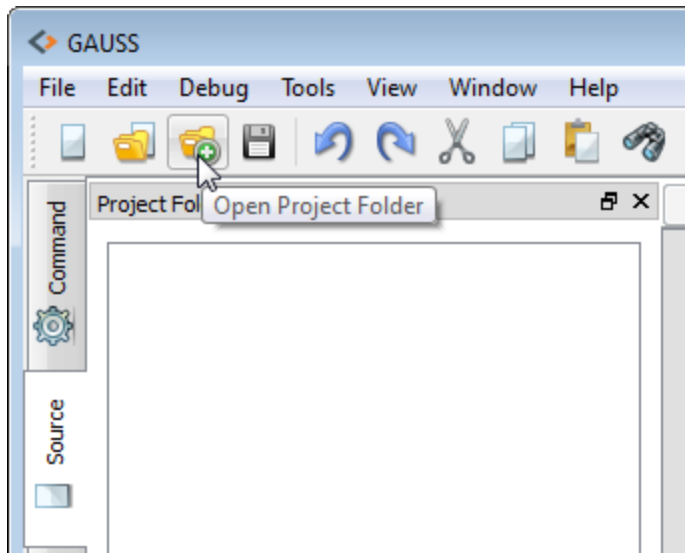
Figure 3.25: Open Project Folder

By default, all files in the folder will be shown. If you would like the Project Window to display only certain file types, you can enable file filtering by right-clicking in the Project Window and selecting "Filter GAUSS Files."
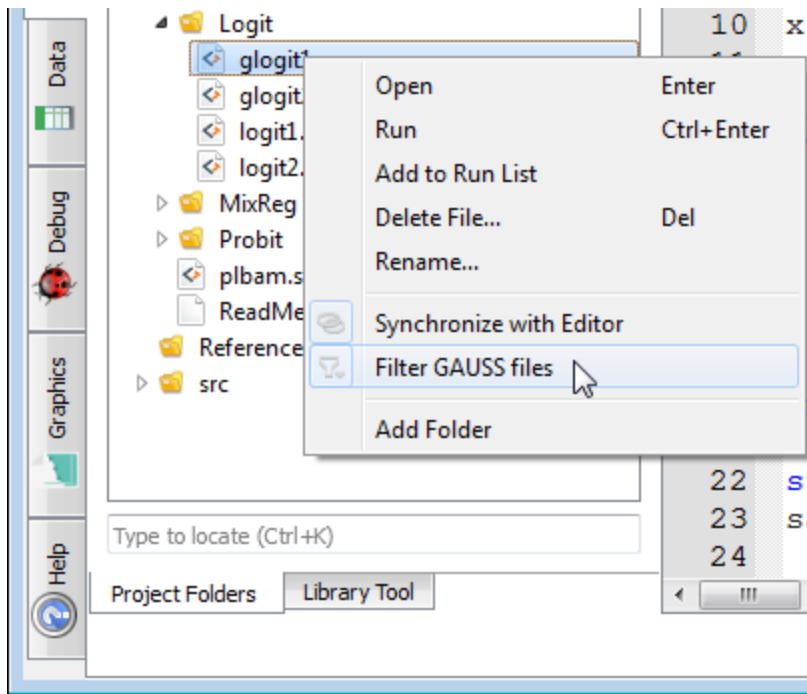
Figure 3.26: Filter File List

With "Filter GAUSS Files" enabled, the Project Window will display only files that match the list of file extensions found under "Tools->Preferences->Source->Project View Filter: Files to Show". This is a comma-separated list of file extensions, which may be modified by the user.

Interacting with files in "Project Window"

Double-click on a file name to open it or right click to bring up a context menu that will allow you to perform other actions, such as: "open", "run", "delete" and "rename".

Searching for text through all files in a directory

Right-clicking on a directory in the Project Window and selecting "Find in folder" will open up the "Source Browser" an advanced find-and-replace window. The "Directory" field will be pre-populated with the directory you selected.
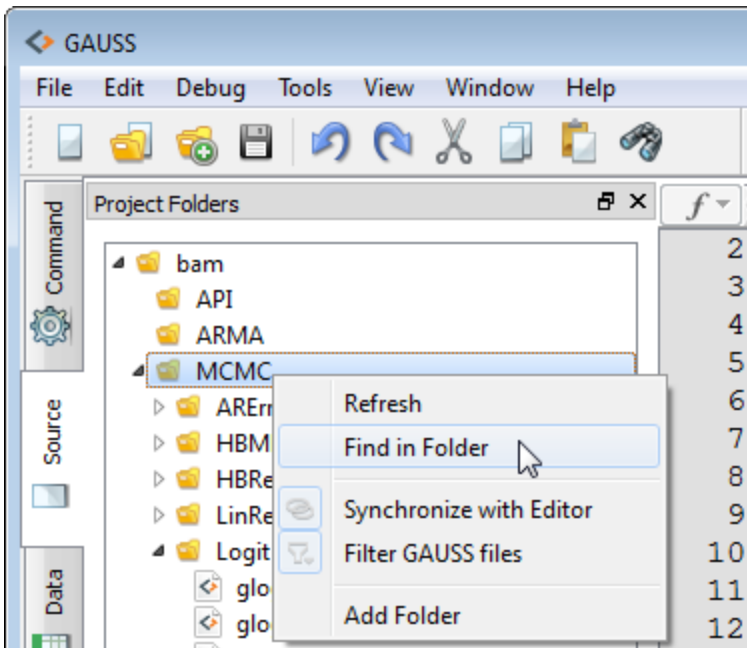
Figure 3.27: "Find in Folder"

By default the "File Pattern" field will contain "*.*". This means GAUSS will search in all files in the specified directory that have any type of file extension. If you would like to include in your search files that do not have file extensions, change the "File Pattern" field to just "*". Finally, add the text you wish to search for to the "Find" field and click the "search" button.
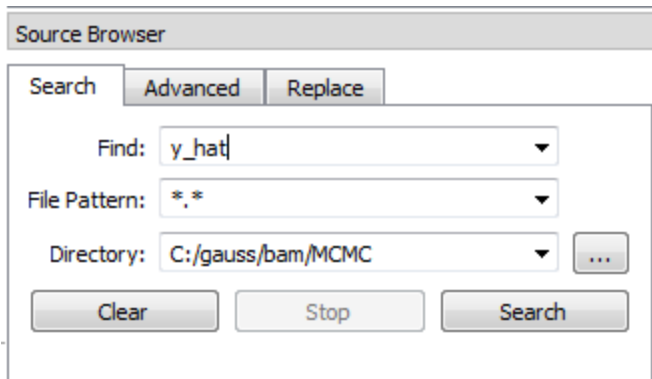
Figure 3.28: "Find in Folder" opens the Source Browser

Searching for a file by name

When working with large numbers of files over multiple directories, it can be difficult to search through a growing list of tabs. The GAUSS Project View Window allows you to almost instantly locate any file that is in one of the folders you have open or in your GAUSS SRC_PATH. To search for a file, click in the text edit box at the bottom of the Project Window, or use the Ctrl+K shortcut. As you begin to enter the filename, an auto-complete list of matching files will appear. The up and down arrow keys will navigate the autocomplete list. Press ENTER to open the file.
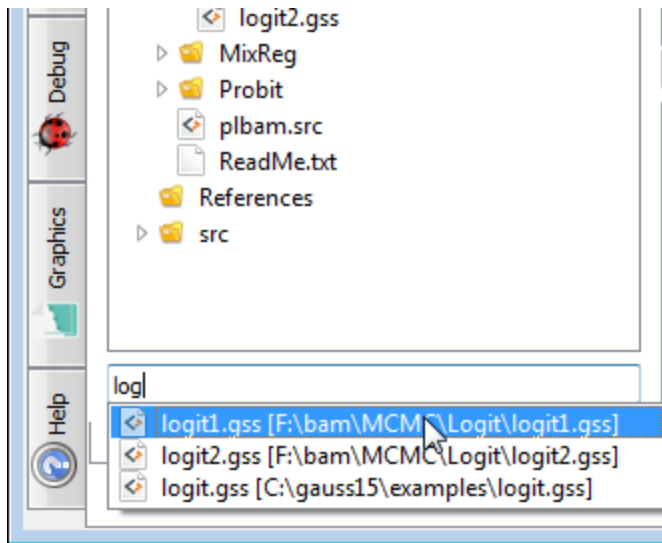
Figure 3.29: Find File

3.4 Data Page

The Data Page provides a tree view of all of the active symbols in your GAUSS work-space organized by data type. Double-click one of your active symbols to open the data in a Symbol Editor. Symbol Editors provide a spreadsheet like view of your data.
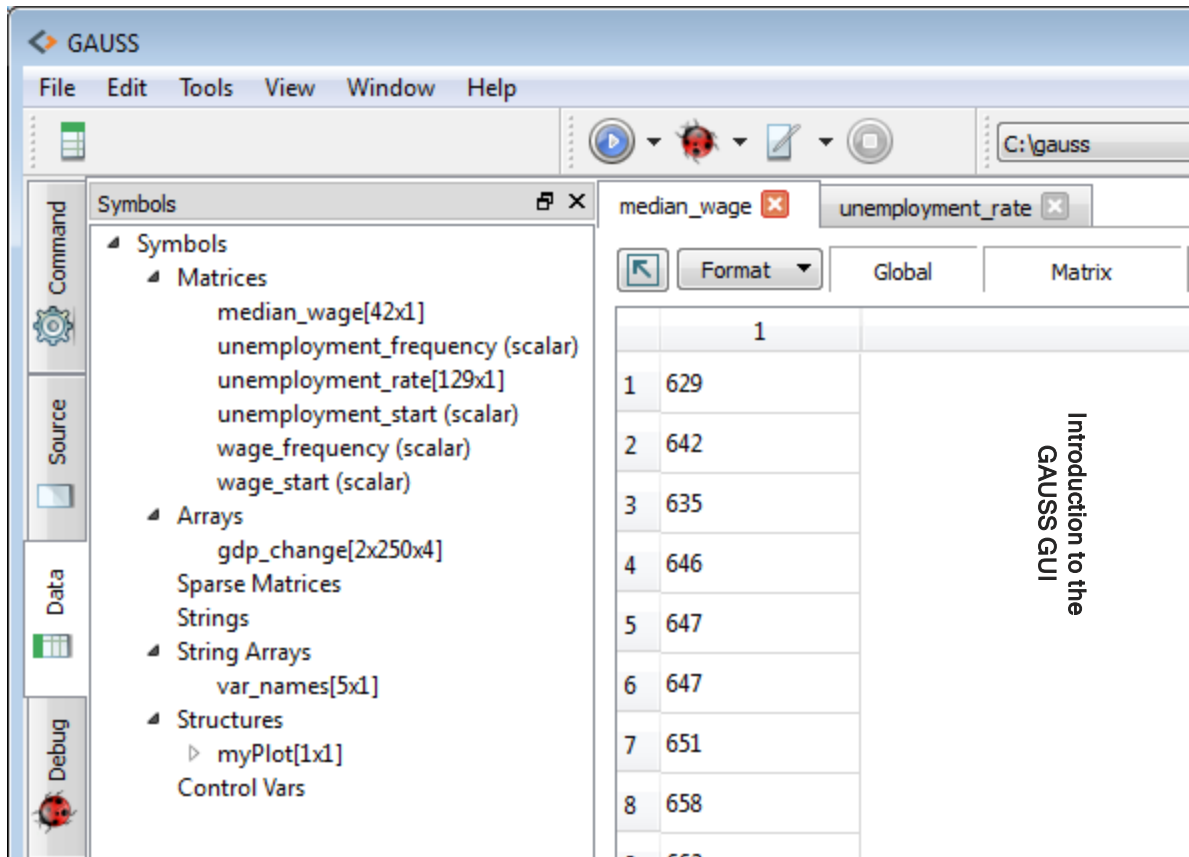
Figure 3.30: Symbol Editor

3.4.1 Changing values

You may change the value of a single cell in a symbol editor by double-clicking in the cell and entering the new value. Note that hitting enter or clicking outside the cell will save your change. To revert an in-process change before it is saved, hit the escape key.
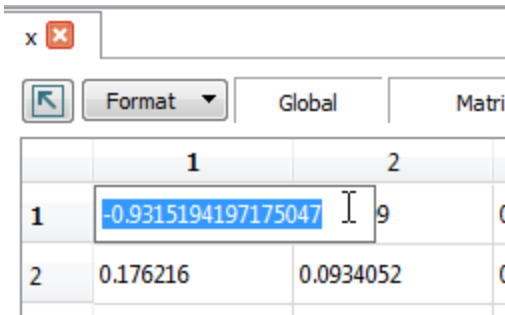
Figure 3.31: Change Cell Value

3.4.2 Changing formatting

Change the formatting of the numbers in a symbol editor by clicking the "Format" button and then, selecting a format type and then selecting the precision.
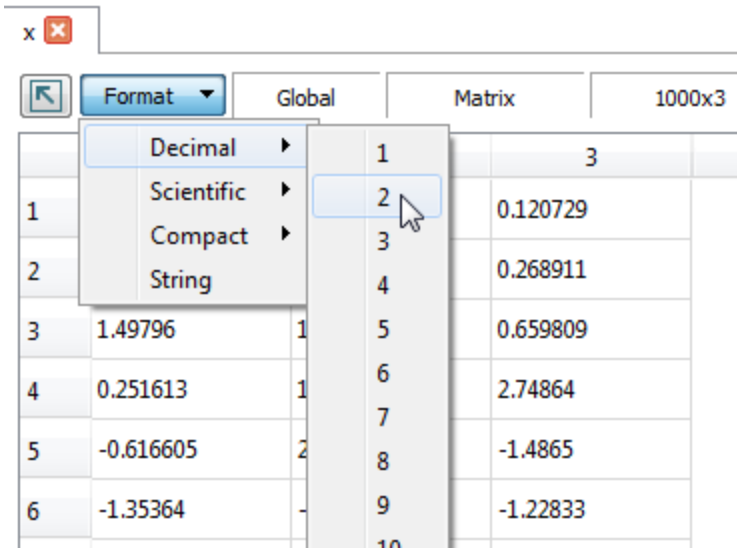
Figure 3.32: Change Number Format

The "Compact" format will choose either "Scientific" or "Decimal", whichever creates the most compact representation of the data.

If none of the cells are selected, the formatting will be applied to all cells. If you select a cell, column or row before you choose the format style, it will only be applied to the selected data. This allows you to apply different formatting to each of your variables if you desire.

3.4.3 Navigating multi-dimensional arrays

At the top-right of an array symbol editor you will find a listing of the all the dimensions in the array. All but the final two dimensions will be list widgets. Click them to select a particular dimension to view.
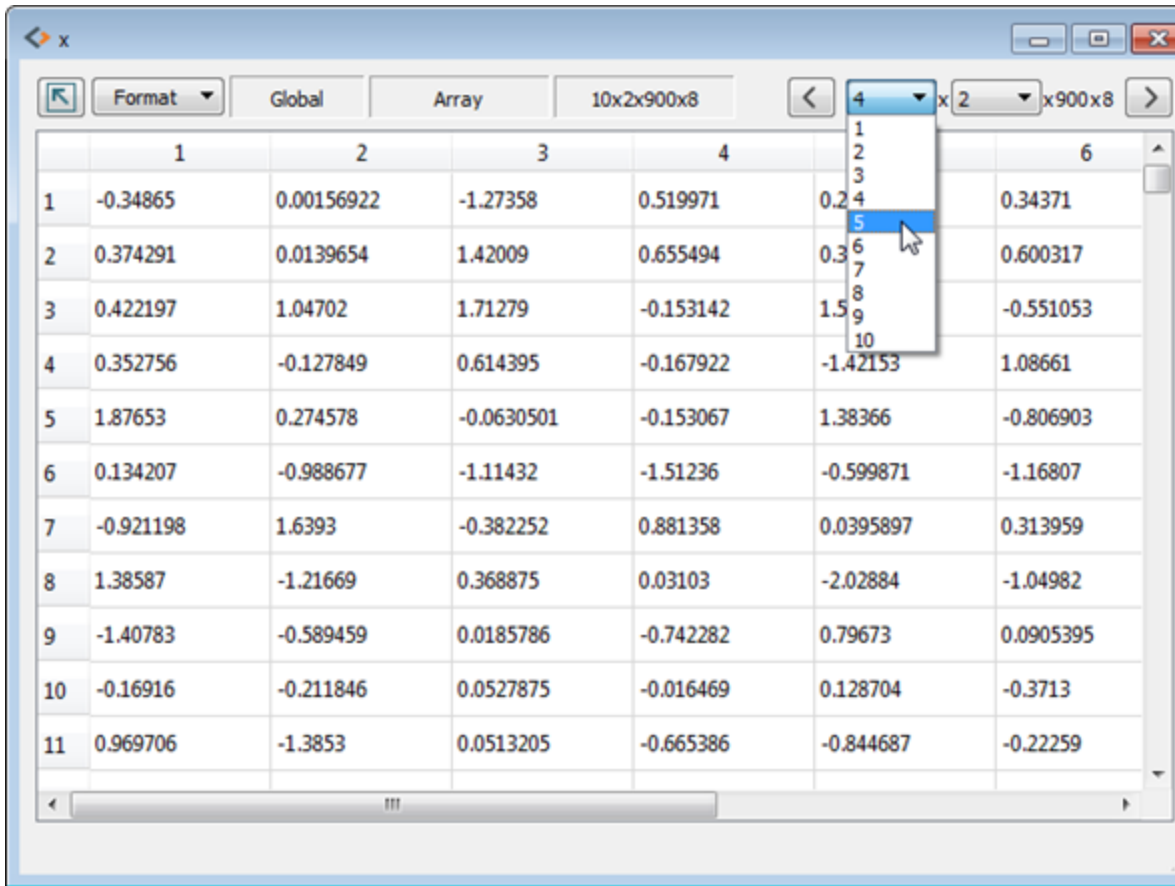
Figure 3.33: Select Dimension to View

On the left of the array dimensions is a button with a leftward facing chevron. Clicking this button will navigate towards the start of the array. The corresponding button on the right of the array dimensions will navigate towards the end of the array.

3.4.4 Viewing structure members

Double-click on a structure member in the Active Symbol Tree to open it in a symbol editor.

3.4.5 Creating a floating watch window

You may undock the symbol editors from the "Data Page" and place them anywhere on your screen by clicking the "Toggle dock" button which looks like an arrow to the left of the "Format" button. They will be updated every time that a program ends or a command-line statement is executed, which can make this a good way to keep track of your data.

3.5 Debug Page

3.5.1 Menus and Toolbars
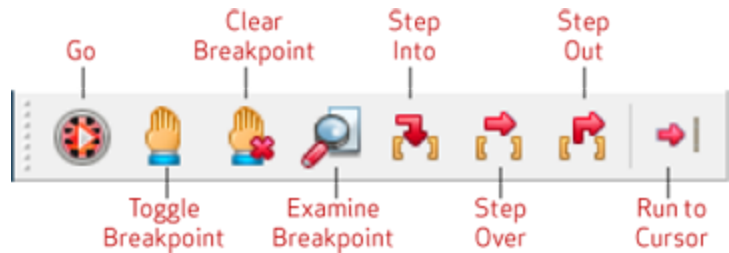
Figure 3.34: Debug Toolbar

Go	Runs the program to the next breakpoint.
Toggle Breakpoint	Sets/Clears a breakpoint at the cursor.
Clear Breakpoints	Clears all breakpoints in a file.
Examine Variable	Opens a watch variable in a symbol editor.
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, 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.
Run to Cursor	Runs the program until it reaches the cursor position.
Stop	Terminates a debugging session.

Components and Usage

The Debug Page is composed of five main widgets:

Breakpoint List	An interactive list of all breakpoints.
Call Stack Window	An interactive display of the chain of procedure calls.
Local Variable Window	An interactive display of all variables that are in scope.

| Variable Dump Window | Displays the full contents of a variable. |
| Watch Window | An interactive display of user specified variables. |

The Debug Window indicates which line it is on by the yellow line highlight and the arrow in the left margin. The left margin is also the location where breakpoints are added. To add a breakpoint, click in the left margin of the Debug Window on the line you wish to add the breakpoint. Clicking an active breakpoint will remove it.

Figure 3.35: Debug Window

Starting and Stopping the Debugger

You can start debugging of a file you are in by pressing CTRL+D. Click the Debug button to debug the file in the top of the Action List. Placing your mouse over the Debug button will reveal a tooltip with the name of this file, or click the downward pointing triangle next to the debug button and select a file from the list.
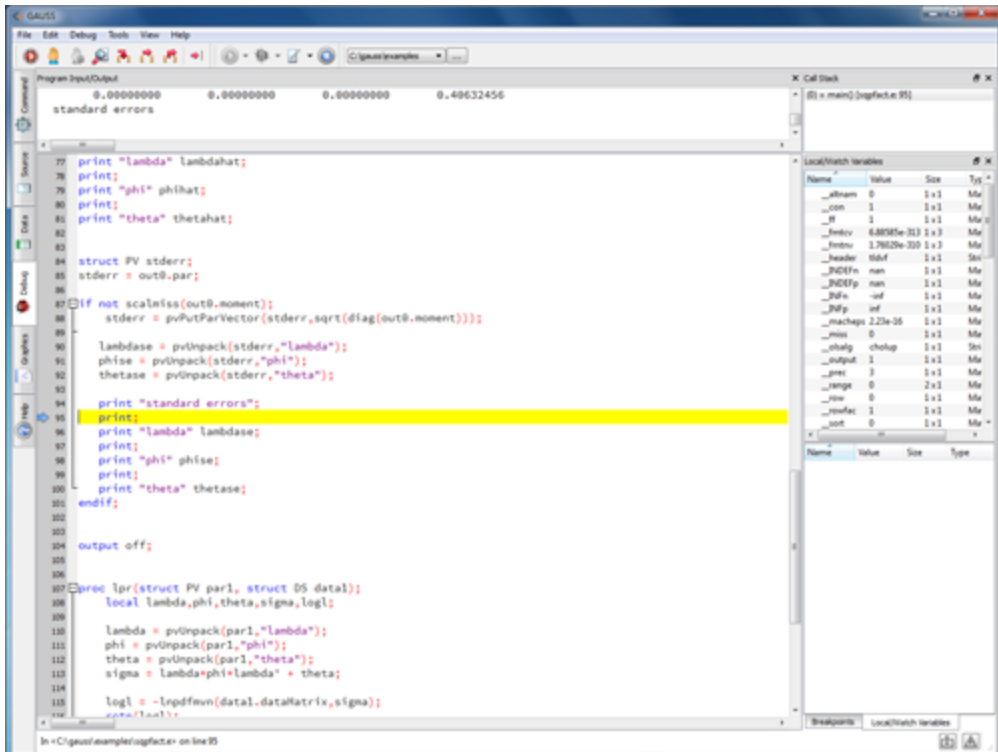
Introduction to the
GAUSS GUI
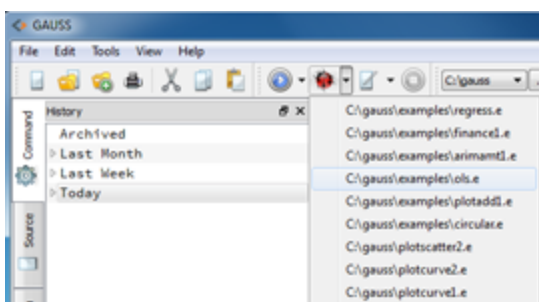
Figure 3.36: Debug Button

When the debugger is started, it will highlight the first line of code to be run. Any breakpoints are shown in the left margin of the window. You can stop debugging at any time by clicking the Stop button on the debug toolbar.

3.5.2 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** option on the Debug menu.

Setting and Clearing Breakpoints

To set breakpoints in any part of the file not currently being executed, just click in the left margin of the line on which you would like the breakpoint. Alternatively, you can highlight a line then click Toggle Breakpoint.
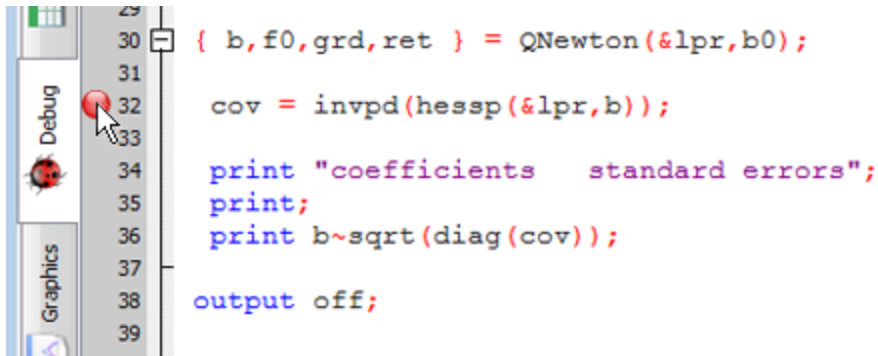
Figure 3.37: Setting Breakpoints

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

3.5.3 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.
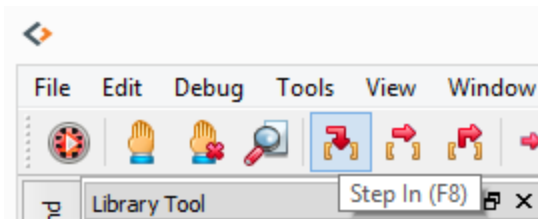
Figure 3.38: Step In (F8)

Use **Step Out** to execute to the end of the current function without pause and return to the calling function.
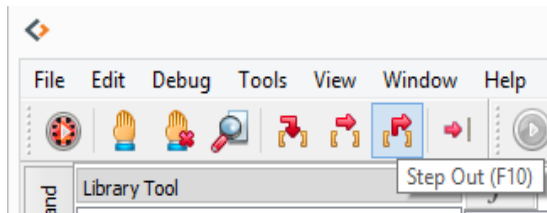
Figure 3.39: Step Out (F10)

Use **Step Over** to execute the line of code currently highlighted by the debugger without entering the functions that are called.
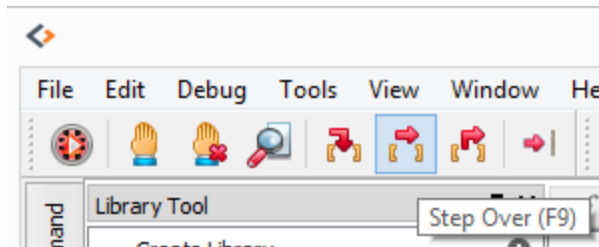
Figure 3.40: Step Over (F9)

Use **Stop** to stop the debugger.
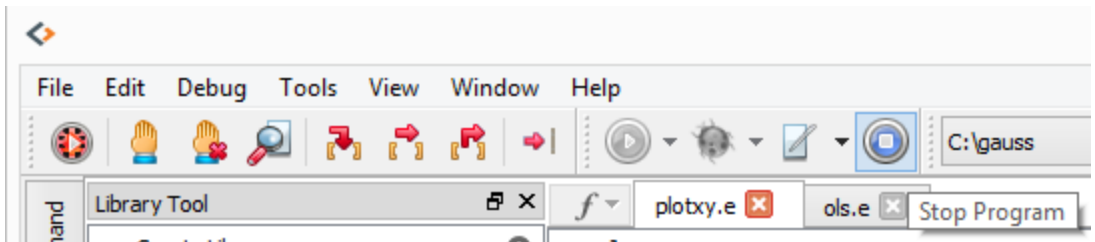
Figure 3.41: Stop

3.5.4 Viewing Variables

GAUSS allows you several ways to view the values of variables during debugging.

Hover Tool-tip View

Once the debugger is started, hovering your mouse over a variable name will open tool-tip with a preview of the values in the variable.

Figure 3.42: Hover Tool-tip

These tooltip views are only intended to give a quick view of the data, so they may not show all data held by the variable. If you need to view more data, you may either click on the variable name in the file and type CTRL+E, click the Examine Variable toolbar button and enter the variable name, or double-click the variable in the Local/Watch Window to view the variable in a floating symbol editor.
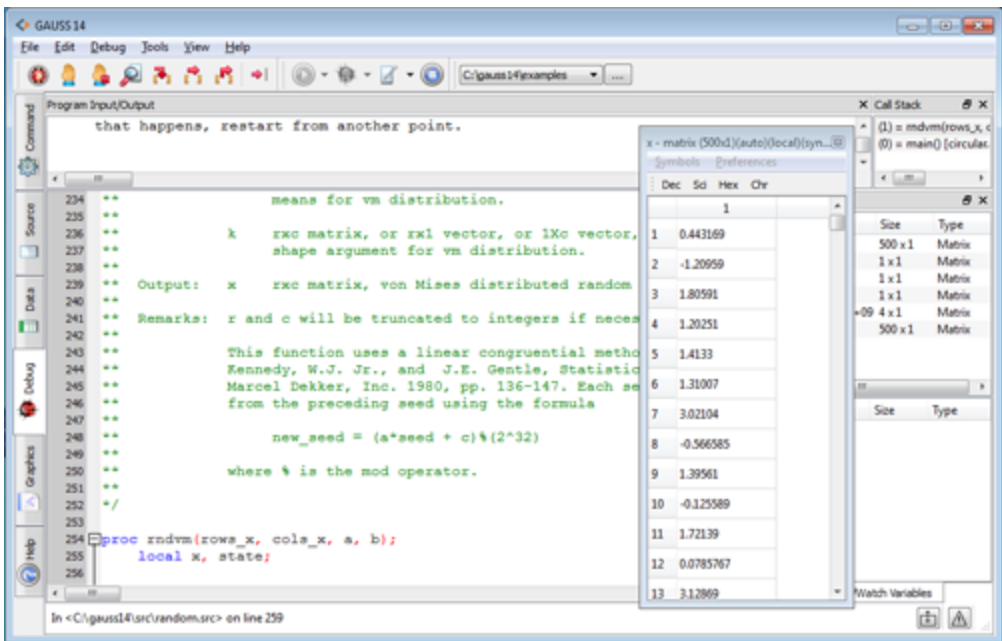
Figure 3.43: Watch Window

Setting a Watch Variable

The Watch Window list is directly below the the Local Variable list. It displays the same information about the variables in its list as does the Local Variable list. By default this list starts empty. You may add variables to the list by either right-clicking over the name of a variable in the Local Variable list and selecting "Add to watch" from the context menu, or by right-clicking directly in the Watch Window and selecting "Add new".

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

1. A local variable within a currently active procedure.

2. A global variable.

3.6 Help Page

The Help Page gives you access to the entire **GAUSS** help system in HTML format. The table of contents tree is on the left. Click the + symbol to expand a particular section of the contents and double-click on the title to view the page. As on the other pages, the Program Input/Output Window and the Error Window are available via toggle buttons on the status bar. It can be helpful to enter an interactive command and/or view error output while simultaneously viewing the relevant documentation.
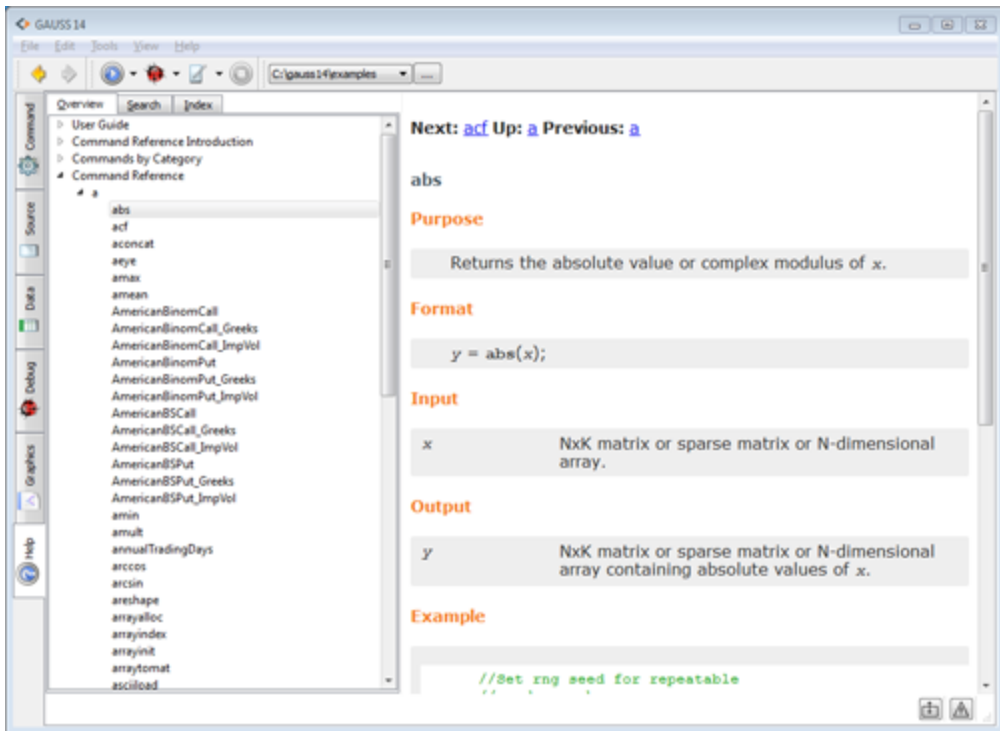
Figure 3.44: Help Page

Hot Keys

F1	Opens the Command Reference section for the highlighted command.
CTRL+F1	Opens a programming editor with the function definition of a highlighted procedure.
SHIFT+F1	Opens a floating Help Window for the highlighted command.

4 Hot Keys and Shortcuts

4.1 Hot Keys and Shortcuts

Action Hot Keys

F2	Navigates to the next bookmark in the Active File.
F3	Find again.
F4	Runs highlighted text, or of no text is highlighted, runs the statement under cursor and goes to the next statement.
F5	Run file at top of Action List.

F6	Debug file at top of Action List. Inside a debug session, F6 will cause the debugger to run to the next breakpoint, or the end of the file if no breakpoints are encountered.
F7	Edit file at top of Action List.
F8	Step in (during a debug session).
F9	Step over (during a debug session).
F10	Step out (during a debug session).
CTR+D	Debug the Active File.
CTRL+R	Run the Active File.

Editor Hot Keys

CTRL+A	Select all.
CTRL+C	Copy.
CTRL+D	Debug the current file.
CTRL+E	Opens the variable under cursor in a floating symbol editor window.
CTRL+F	Find and replace.
SHIFT+CTRL+F	Open Source Browser--advanced find and replace inside a file, or across directories.
CTRL+G	Go to line.
CTRL+I	Format, or autoindent, current file or selected text.
CTRL+K	Locates a file in the project view.
CTRL+L	Delete current line.
CTRL+N	Open new file.
CTRL+O	Open file.
CTRL+R	Run active file.
CTRL+S	Save current file.

`CTRL+T`	Swaps current line with line above.
`CTRL+W`	Close current file.
`CTRL+Y (CMD+SHIFT+Z on Mac)`	Redo.
`CTRL+Z`	Undo.
`CTRL+/`	Toggle block comment.
`CTRL+~`	Cycle through open editor windows.
`CTRL+F1`	Go to definition of procedure under cursor.
`CTRL+SPACE`	Activates autocomplete.
`ALT+Left Arrow`	Go to previous cursor position (may be in a previously edited file).
`ALT+Right Arrow`	Go to next cursor position (after going backwards with ALT+Right Arrow).
`ESCAPE`	Close 'find and replace' or 'source browser' windows, if they are open.

Help Hot Keys

`F1`	Opens the Command Reference page for command under cursor on the Help Page.
`SHIFT+F1`	Opens a floating Command Reference page for the command under cursor.

4.2 Navigation Hot Keys

GAUSS automatically navigates between pages when you perform certain actions. For example, if you are editing a file in the Source Page and type `CTRL+D` to debug that file, **GAUSS** will automatically take you to the Debug Page.

However, there are also several hot keys that allow you to navigate quickly and easily between pages without performing any actions:

CTRL+1	Brings up the Command Page.
CTRL+2	Brings up the Source Page.
CTRL+3	Brings up the Data Page.
CTRL+4	Brings up the Debug Page.
CTRL+5	Brings up the Help Page.
CTRL+TAB	Brings up the next page. For example, CTRL+TAB from the Command Page will bring up the Source Page. CTRL+TAB from the Help Page will wrap and bring up the Command Page.
ALT+TAB	Cycles between any pages that are undocked as well as other open programs.
WINDOW+TAB	Windows only: Cycles between any pages that are undocked as well as other open programs.
Mouse Scroll Wheel	When floating over any set of tabs, the mouse scroll wheel will cycle through the open tabs. This will work for programming editor tabs, symbol editor tabs, and the main page tabs on the left of the main application.

Hot Keys and Shortcuts

4.3 Focus Program Output on I/O

Under the **Tools->Preferences->Command** is a check box entitled **Focus Program Output on I/O**. Selecting this option will open up a Program Output Window if any program output is printed or if any input is requested by the **GAUSS** commands `key`, `keyw`, or `cons`.

4.4 F1 Help

If your cursor is on the name of a **GAUSS** command in an editor, you can press F1 and it will take you to the Command Reference listing for that command. Inside the Help system, highlight command names by double-clicking on them to enable F1 help navigation.

Shift+F1 Inline Help

Typing SHIFT+F1 when your cursor is on a **GAUSS** command will open an inline help window that floats above your editor window.
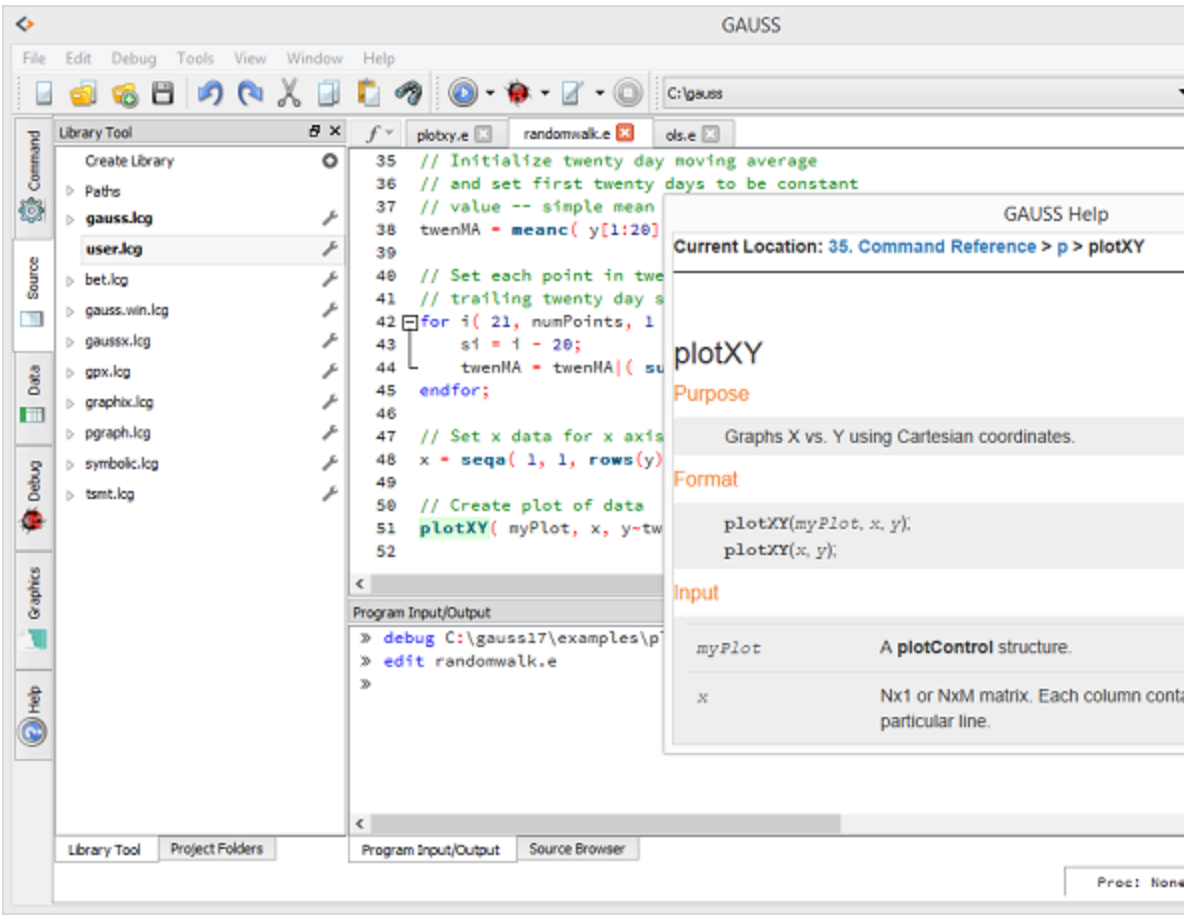
Figure 4.1: Inline help window opened with SHIFT+F1

4.5 CTRL+F1 Source Browsing

If your cursor is on the name of a procedure or global variable that resides in an active **GAUSS** Library (.lcg file), typing CTRL+F1 will open the source file in a Programming Editor.

To learn more about creating a User Library for your procedures, see **LIBRARIES**, CHAPTER 19 .

5 Using the GAUSS Debugger

The **GAUSS** debugger is a powerful tool to speed up and simplify your program development. Debugging takes place on the Debug Page, which is a full-featured dashboard providing you a wealth of information about the status of your variables and your program every step of the way.
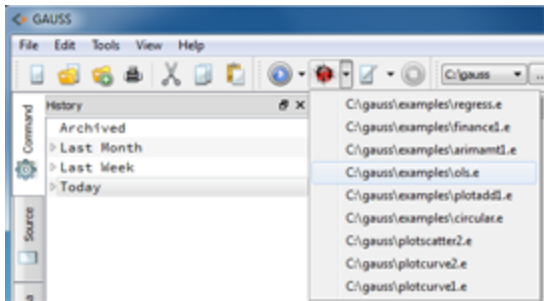
Figure 5.1: The debug button automatically opens the Debug Page

5.1 Starting the Debugger

Figure 5.2: Debug Button

You may start a debug session by selecting the Action List Debug button, by entering CTRL+D from the file that you would like to debug, or by entering **debug** *filename* from the command line. **GAUSS** will bring the Debug Page to the front, and the debug session will begin with the first line of code highlighted, waiting to be run.
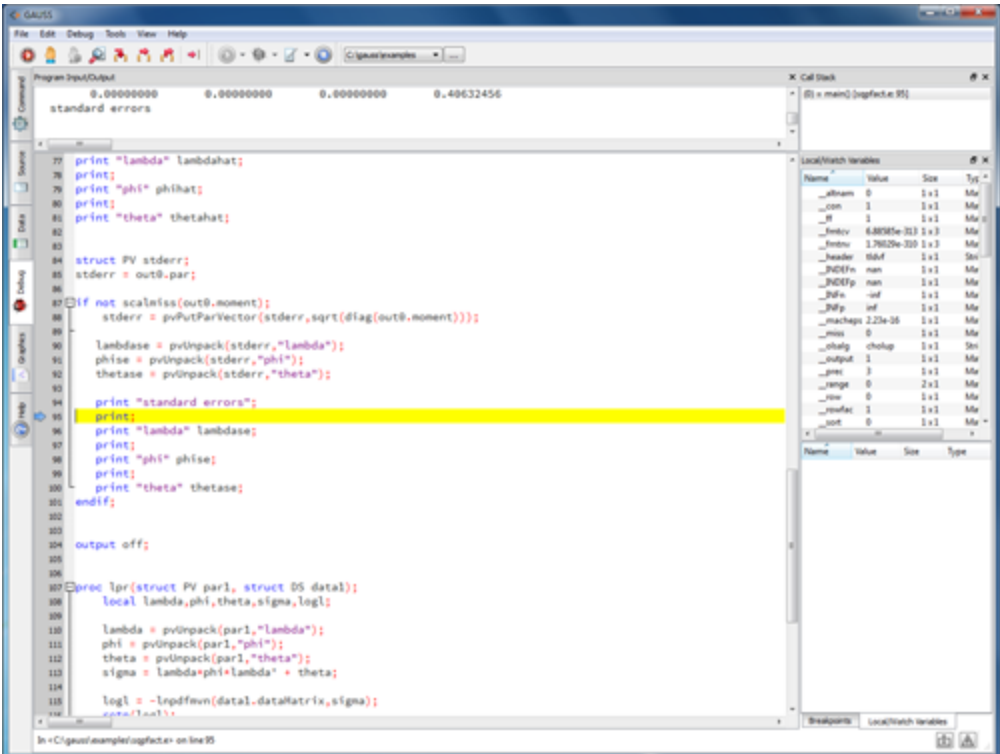
Figure 5.3: Debug Window

From here you can step through your program line-by-line, or run to a line of interest. To tell the debugger to run to a particular line, you must first set a breakpoint. Setting a breakpoint on a line in your program tells the debugger to pause when it gets to that line. You may set a breakpoint in any source file by simply clicking in the margin to the left of the line numbers. You will see a red dot appear in the margin, and a new entry will be added to the breakpoint window, which can be viewed on both the Source and Debug Pages. New breakpoints can be added before or during a debug session.

The Debug Page toolbar gives you the controls you need to navigate during debugging. Hover over any of the buttons to get a tooltip with a description of the button's function and its corresponding hot key.

Figure 5.4: Debug Toolbar

Debug Run	Runs to the next breakpoint.
Debug Stop	Stops the debug session.
Toggle Breakpoint	Toggles a breakpoint on the current line.
Clear Breakpoints	Removes all breakpoints from all files.
Examine Variable	Opens a user-specified variable in a floating Symbol Editor window.
Step in	Steps to the next line. If the next line is a procedure, it will step inside this procedure.
Step over	Steps to the next line without stepping into other procedures.
Step out	Runs through the end of the current procedure and stops in the calling file.

| Run to cursor | Runs to the location of your cursor as if it were a breakpoint. |

5.2 Examining Variables
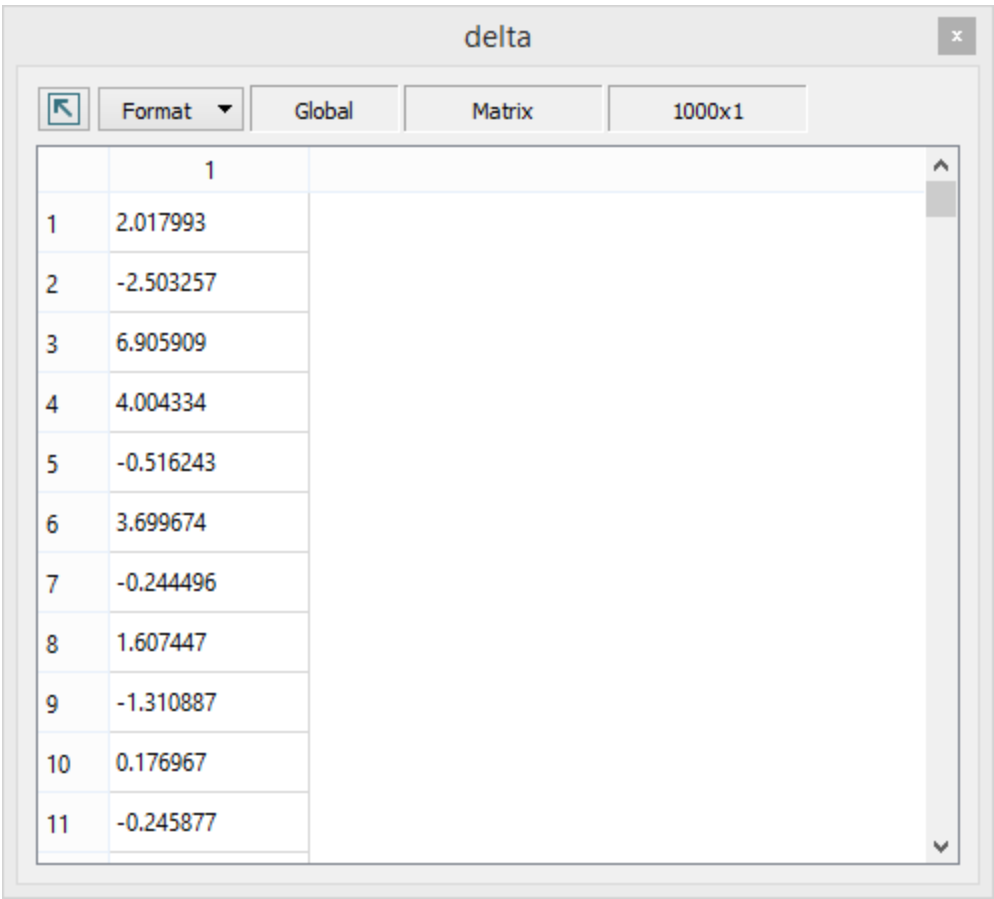
When the debugger is paused at a line in your program, you may view any of the variables in your **GAUSS** workspace. **GAUSS** offers several options for this. The first is through the Local Variable window. This window contains an alphabetical list of every variable that is currently in scope. It also lists the first value of each variable, its type (matrix, string, structure, etc.) and dimensions. The contents of this window are immediately updated each time you step to a new line of your program.

Local/Watch Variables

Name	Value	Size	Type
_rtl_ver	14	3x1	Matrix
_ss_db	0	1x1	Matrix
annotation_type	"rectangle"	10	String
delta	-1.45467	1000x1	Matrix
myEllipse			Struct
myPlot			Struct
numPoints	1000		Scalar
seed	6.46637e+06		Scalar
state	2.59287e-307	318x1	Matrix
x	-1.03359	100x5	Matrix
x_end	0.737021		Scalar
x_start	0.0458879		Scalar
y	1000	1000x1	Matrix
y_end	0.995587		Scalar
y_start	0.807611		Scalar

Name	Value	Size	Type
delta	-1.45467	1000x1	Matrix
y_end	0.995587		Scalar

Figure 5.5: Local Variable Window

Double-clicking on a variable in the Local Variable window opens that variable in a floating symbol editor. This symbol editor window looks much like a spreadsheet and will display the entire contents of the variable.

Located just below the Local Variable window is the Watch window. The Watch window looks and acts much the same as the Local Variable window, but it holds only variables that you specifically add to it. This allows you to place variables of interest in a smaller list that is easier to scan. You can add a variable to the Watch window by either right-clicking on a variable in the Locals list and selecting "Add to watchlist", by right-clicking in the Watch window and selecting "Add new", or by highlighting a variable in the file and then dragging and dropping it into the watch window.

Figure 5.6: Symbol Editor Window

You may see a tooltip preview of any variable in your file by floating your cursor over the variable name. Placing your cursor on a variable and pressing CTRL-E will open that variable in a floating symbol editor.

5.3 The Call Stack Window

The Call Stack window is like a map into your program. The top entry in the Call Stack window is always your current location. It lists the name of the procedure you are in, the arguments that this procedure takes, the name of the file it is in, and the line number you are on. The next item in the list displays the same information for the location from which your current procedure was called. The item after that displays the location from which that procedure was called and so on.
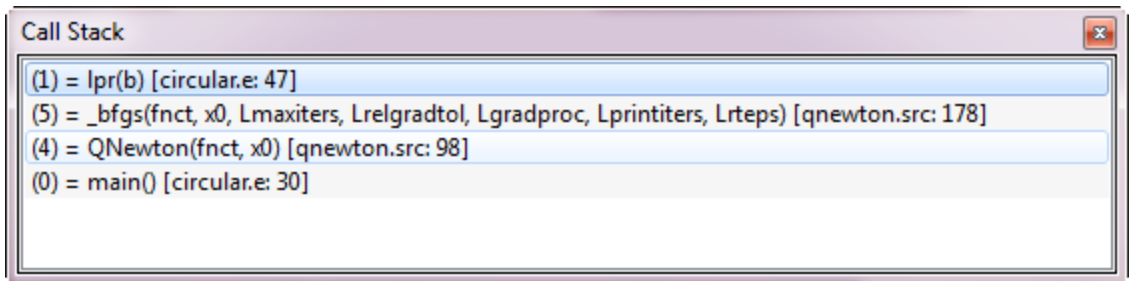
Figure 5.7: Call Stack Window

This list is interactive. Single-clicking on any of the items in the Call Stack window will bring you to that particular line and file. For example, if you would like to examine the line from which your current procedure was called, click on the second item in the call stack list. Not only will this open that line and file in your Debug window, but it will also update the Local Variable window to display all the variables that exist at that location, as well as their current values. To return to your current location, click the top line in the Call Stack window.

5.4 Ending Your Debug Session

If you would like to terminate a debug session before the debugger has run through the entire program, click the Debug Stop button, located at the top left of the Debug Page.
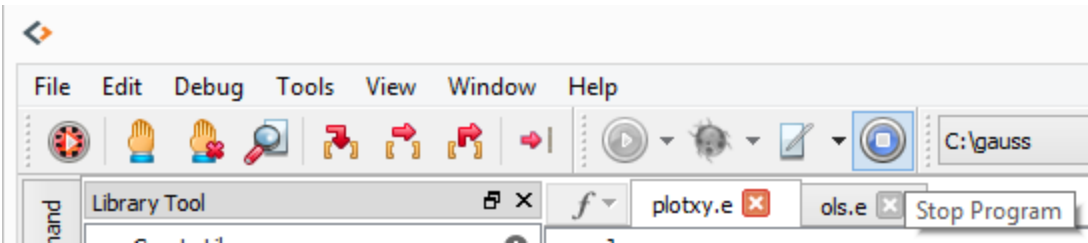
Figure 5.8: Debug stop button

6 GAUSS Graphics

6.1 Overview

The plotting functionality available in **GAUSS** is designed to provide intuitive methods for visualizing data using a variety of plot types. The main plot drawing functions are:

plotArea	Creates a cumulative area plot.
plotBar	Creates a bar plot.
plotBox	Creates a box plot using the box graph percentile method.
plotContour	Creates a contour plot.
plotHist	Calculates and creates a frequency histogram plot.

plotHistF	Creates a histogram plot from a vector of frequencies.
plotHistP	Calculates and creates a percentage frequency histogram plot.
plotLogLog	Creates a 2-dimensional line plot with logarithmic scaling of the both the x and y axes.
plotLogX	Creates a 2-dimensional line plot with logarithmic scaling of the x-axis.
plotLogY	Creates a 2-dimensional line plot with logarithmic scaling of the y-axis.
plotPolar	Creates a polar plot.
plotScatter	Creates a 2-dimensional scatter plot.
plotSurface	Creates a 3-dimensional surface plot.
plotTS	Creates a graph of time series data.
plotXY	Creates a 2-dimensional line plot.

6.2 Basic Plotting

The simplest way to plot data in **GAUSS** is to use default values for the plot settings, such as line color, line size, legend text, etc. These settings may be changed by the user from the main menu bar: **Tools-> Preferences-> Graphics**. To create a plot using default plot settings, simply call one of the plotting functions with your data as inputs:

```
//Create a sequential column vector from
//-5 to 5 with 101 steps
x = seqa(-5, 0.1, 101);

//Set y to the normal probability density function
y = pdfn(x);

//Plot the data
plotXY(x, y);
```
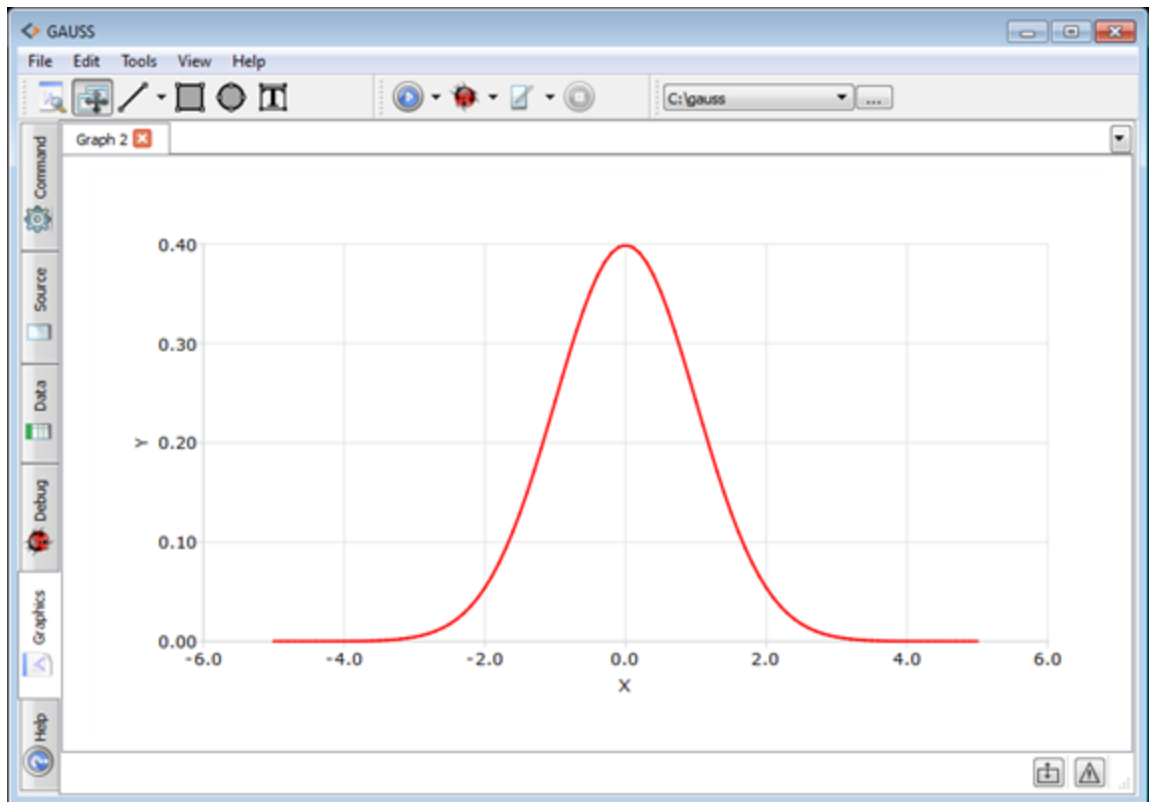
Figure 6.1: One-Curve Plot

6.2.1 Plotting multiple curves

Each column of the input matrices is treated as a separate curve or line on the graph.
Below is an example that uses the variables created from the example above and adds
an additional line:

```
x = seqa(-5, 0.1, 101);
y1 = pdfn(x);
```

```
//Set y2 to the Cauchy probability density function
y2 = pdfCauchy(x, 1, 1);

//Plot the data using the ~ operator to horizontally
//concatenate y1 and y2
//Note that y1 and y2 will be plotted against the same
//x values
plotXY(x, y1~y2);
```
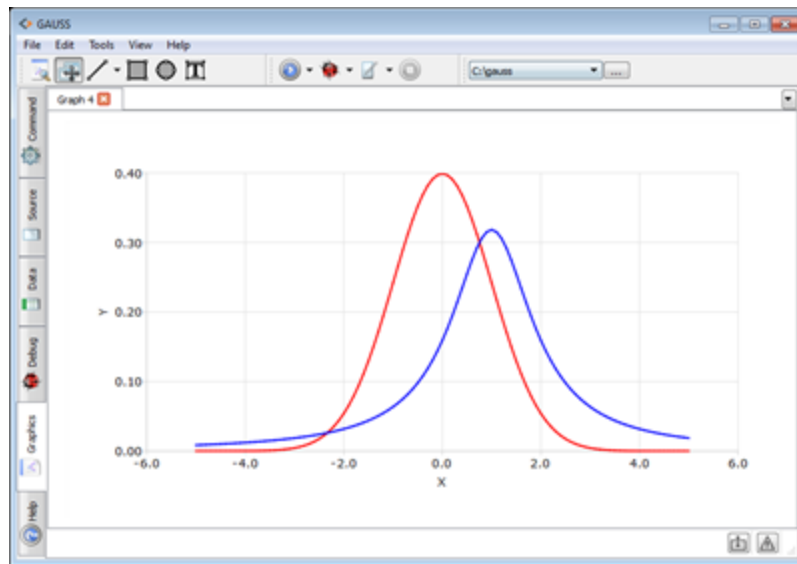

Figure 6.2: Two-Curve Plot

6.3 Plot Customization

GAUSS offers two ways to customize your graphs. The first is through the graphics preferences dialog window. The second method for plot customization is using a **plotControl** structure.

6.3.1 Using the Graphics Preferences Settings Window

Main Graph Settings

The default settings for graphics can be opened by selecting Tools->Preferences from the main application menu bar . Then select Graphics from the list on the left of the preferences window.
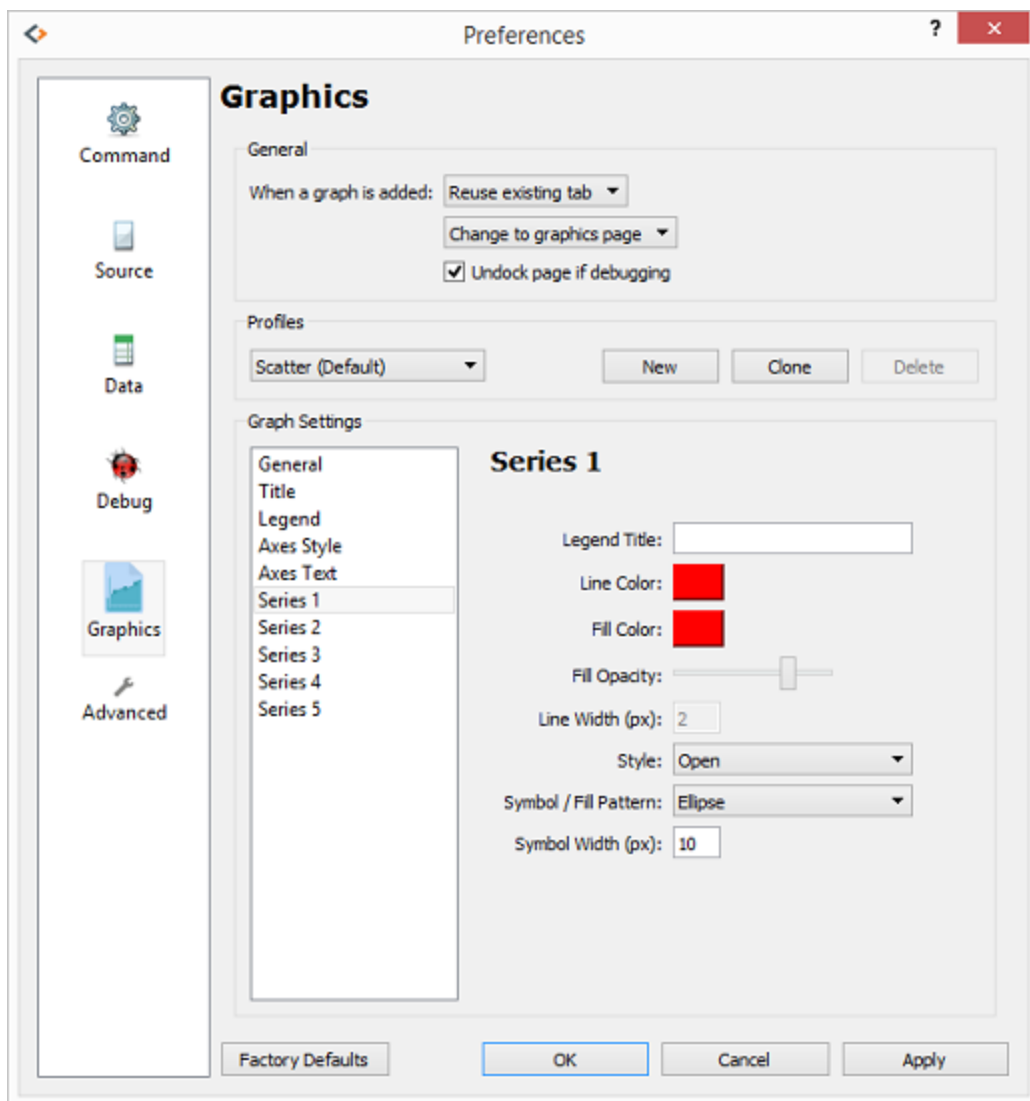
Figure 6.3: The Graph Settings Tab

Selecting a graph type

Each type of graph in GAUSS has its own settings. Click on the graph type list widget to expand the list of graph types. Click on one of the listed graph types to view its settings.
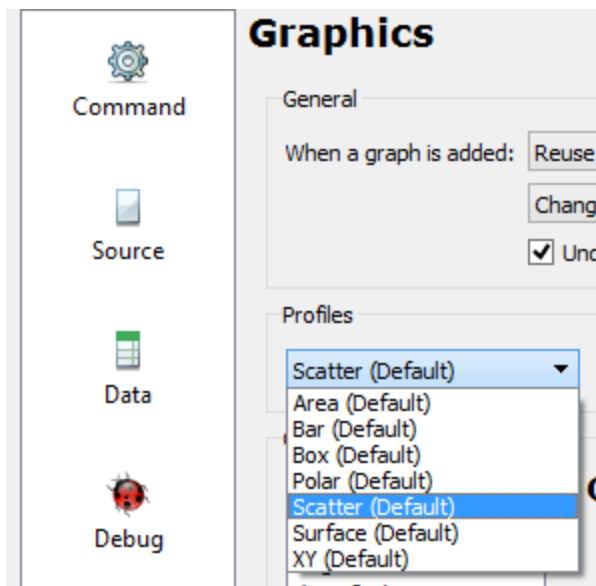
Figure 6.4: The Group Tab

Once you have selected a graph type, browse through the settings by clicking on a an individual graph component, such as the "Legend", "Axis Style" or "Axis Text". "Series 1", "Series 2", etc contain the preferences for the curves such as "line color", "line thickness", "symbol type", etc.
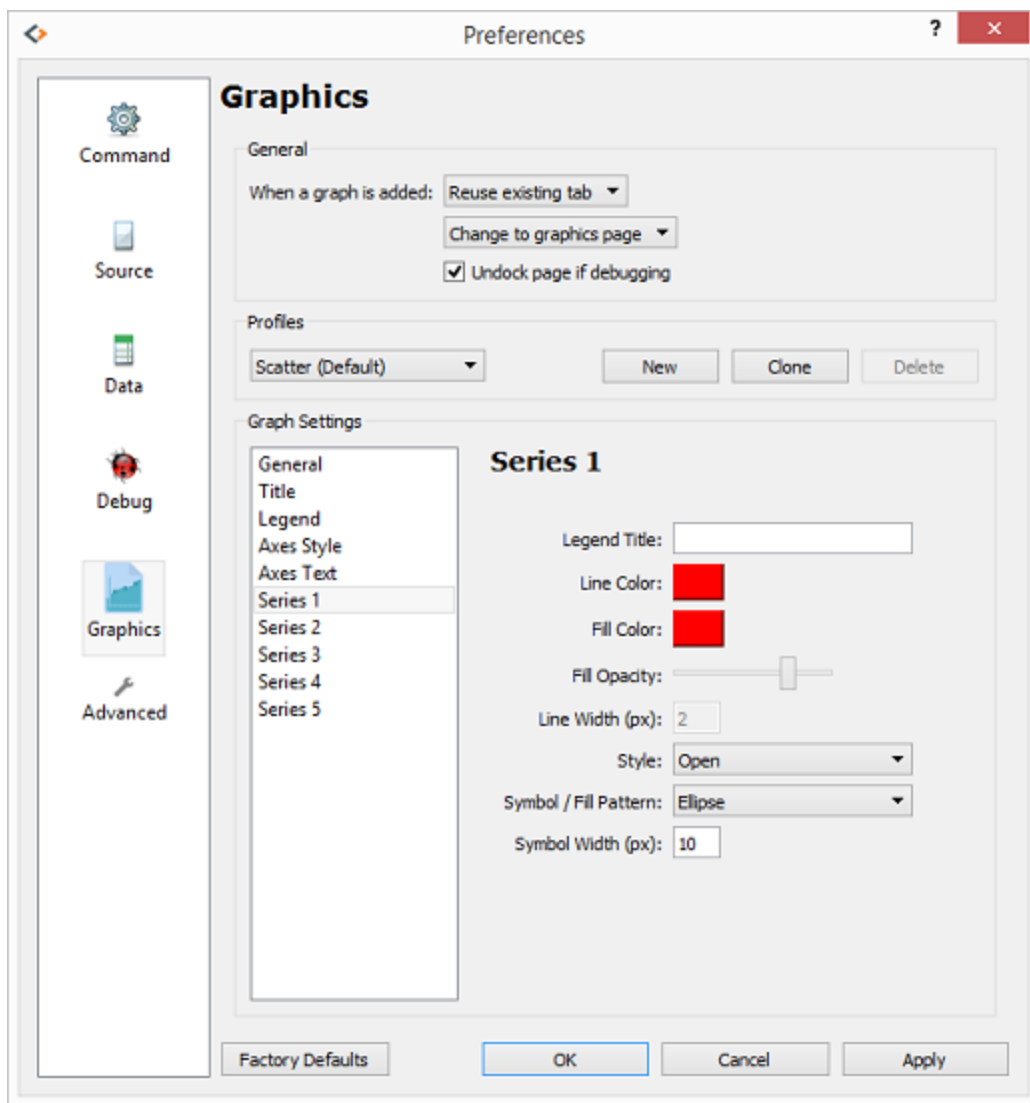
Figure 6.5: Scatter plot graph settings

After you have made your desired changes, Click the 'Apply' and 'OK' buttons. GAUSS will use these preferences for all future graphs that are made without passing in a

plotControl structure. As we will see in the next section, these settings will also be the starting point when you create a plotControl structure. These settings may be changed at any time.

Contour/Surface Plot Z-Level Color Settings
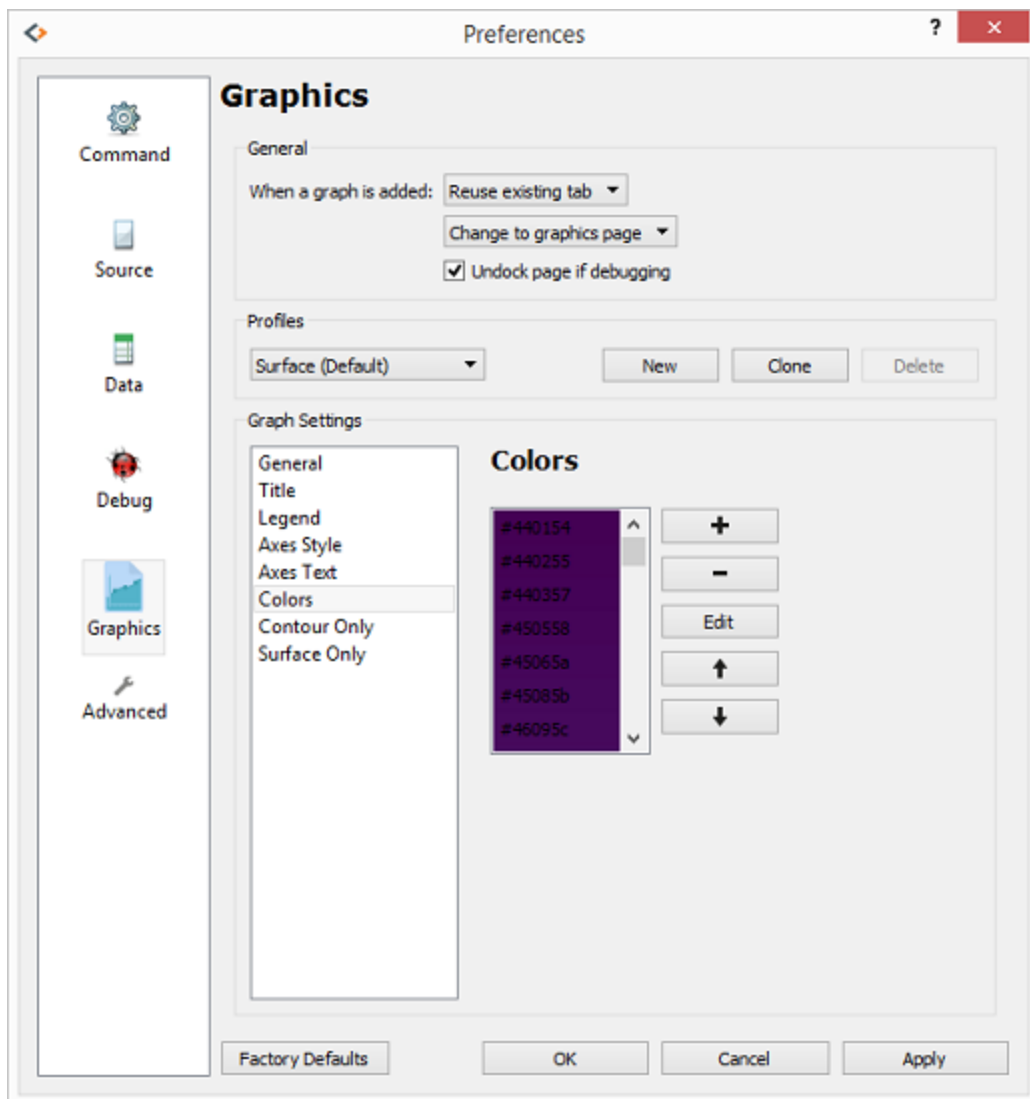
Figure 6.6: Surface/Contour Level Color Settings

If you select the 'Surface' graph type, you will see a 'Colors', 'Contour Only' and 'Surface Only' options instead of 'Series 1, Series 2...' as shown above. The "Colors" control allows you to add and subtract default z-level colors and to also change their order.

Available Actions

Add a new color	Click the add (+) button
Edit a color	Double-click the color
Select a color	Single-click the color
Delete a color	Select a color and click the minus (-) button
Change color order	Click-and-drag color to desired position, or select a color and click the up or down button

Contour Only Settings

Preferences

Graphics

General

When a graph is added: Reuse existing tab ▼

Change to graphics page ▼

☑ Undock page if debugging

Profiles

Surface (Default) ▼   New   Clone   Delete

Graph Settings

General
Title
Legend
Axes Style
Axes Text
Colors
Contour Only
Surface Only

Contour Only

Level Count: 10

Show Labels: ☑

Label Format: g (most concise of 'e' or 'f')   ▼

Label Precision: 3

Label Example: 1.23e-07
                      1.23
                      -0.000123
                      -1.23

Factory Defaults   OK   Cancel   Apply

Figure 6.7: Contour Only Settings

The "Contour Only" control allows you to add and subtract default level count and to also change the label format.

Available Actions

Level Count	Change the level count number
Show Labels	Check the ballot box
Label Format	Choose an option from 'e', 'E', 'f', 'g', 'G'
Label Precision	Type the precision of labels

Surface Only Settings

Preferences

Graphics

General

When a graph is added: [Reuse existing tab ▼]

[Change to graphics page ▼]

☑ Undock page if debugging

Profiles

[Surface (Default) ▼]　　[New]　[Clone]　[Delete]

Graph Settings

| General |
| Title |
| Legend |
| Axes Style |
| Axes Text |
| Colors |
| Contour Only |
| **Surface Only** |

Surface Only

Show Wireframe: ☑

Lighting: ———————

Camera Angle: [Isometric Left ▼]

Zoom Level: ———————

[Factory Defaults]　　[OK]　[Cancel]　[Apply]

Figure 6.8: Surface Only Settings

The "Surface Only" control allows you to change the lighting and camera angle, add and subtract zoom level.

Available Actions

Show Wireframe	Check the ballot box
Lighting	Move the mouse scroll wheel to adjust light of surface
Camera Angle	Choose an option like 'Isometric Left', 'Directly Above', etc.
Zoom Level	Move the mouse scroll wheel to adjust size of plot

6.4 PlotControl Structures

The **GAUSSplotControl** structure provides a powerful and flexible method for programmatic control of your graphs in **GAUSS**. This structure is a convenient package that stores all of the information about how you would like a specific graph to be displayed. These structures may be saved to disk and reloaded during a future session or passed on to colleagues. Using a **plotControl** structure requires just two easy steps:

1. Declare the structure.
2. Initialize the structure.

Once these steps are completed, you may change any of the plot settings in the structure. Once the **plotControl** structure is set how you would like, you can pass it in as the first argument to any of the **GAUSS** plot-creating functions. Below is an example that draws a graph using a **plotControl** structure set to default values:

```
//Declare the structure
struct plotControl myPlot;

//Initialize the structure
myplot = plotGetDefaults("xy");
```

```
//Create a column vector from -3 to 3 with a step size of 0.1
x = seqa(-3, 0.1, 60);
y = sin(x);

//Plot the data using the plotControl structure
plotXY(myPlot, x, y);
```

The available **plotControl** setting functions include:

plotGetDefaults	Applies the user defined default settings for a specified graph type to a **plotControl** structure.
plotSetAxesPen	Sets the color and thickness of the axes lines.
plotSetBar	Sets the fill style and format of bars in a histogram or bar graph.
plotSetBkdColor	Sets background color of a graph.
plotSetColorMap	Sets color map of a contour or surface graph.
plotSetContourLabels	Sets contour labels of a contour graph.
plotSetFill	Sets the fill style, opacity and color for area and bar plots as well as histograms.
plotSetGrid	Controls the settings for the background grid of a plot.
plotSetLegend	Adds a legend to a graph and optionally applies settings.
plotSetLegendFont	Set legend font for a graph.
plotSetLineColor	Sets line colors for a graph.

`plotSetLineStyle`	Sets line styles for a graph.
`plotSetLineSymbol`	Sets line symbols displayed on the plotted points of a graph.
`plotSetLineThickness`	Sets line thickness for a graph.
`plotSetNewWindow`	Sets whether new graphs should be drawn in the same window or in a newly created window.
`plotSetTextInterpreter`	Controls the text interpreter settings for a graph.
`plotSetTitle`	Controls the settings for the title for a graph.
`plotSetWhichYAxis`	Controls whether a particular curve uses the right or left Y-axis.
`plotSetXLabel`	Controls the settings for the X-axis label on a graph.
`plotSetXRange`	Controls the minimum and maximum values of the X-axis.
`plotSetXTicCount`	Sets the number of major tics along the X-axis.
`plotSetXTicInterval`	Controls the interval between X-axis tic labels and optionally at which X-value to place the first tic label.
`plotSetXTicLabel`	Controls the formatting and angle of X-axis tic labels for 2-D graphs.
`plotSetYLabel`	Controls the settings for the Y-axis label on a graph.
`plotSetYRange`	Controls the minimum and maximum values of the Y-axis.
`plotSetYTicCount`	Sets the number of major tics along

	the Y-axis.
plotSetZLabel	Controls the settings for the Z-axis label on a graph.
plotSetZLevels	Controls the heights at which lines are drawn on a contour plot.

Most of these functions begin with **plotSet**. This has two main advantages. First, you may quickly and easily survey the **plotSet** options by typing **plotSet** in a **GAUSS** editor and scrolling through the auto-complete list without needing to consult the documentation. Second, this convention makes your **GAUSS** code easy to read and understand.

Example

```
//Declare plotControl structure
struct plotControl myPlot;

//Initialize the structure with the "bar" default values from
//the main application menu Tools->Preferences->Graphics
myPlot = plotGetDefaults("bar");

//Create data to plot
x = seqa(1, 1, 10);
y = abs(rndn( 10, 1 ));

//Change plot settings
//Turn off the grid
plotSetGrid(&myPlot, "off");

// Set x-axis label
plotSetXLabel(&myPlot, "Day of Project");

//Set the title, title font and font size
plotSetTitle(&myPlot, "Example Bar Plot", "arial", 18);
```

```
//Draw graph
plotBar(myPlot, x, y);
```
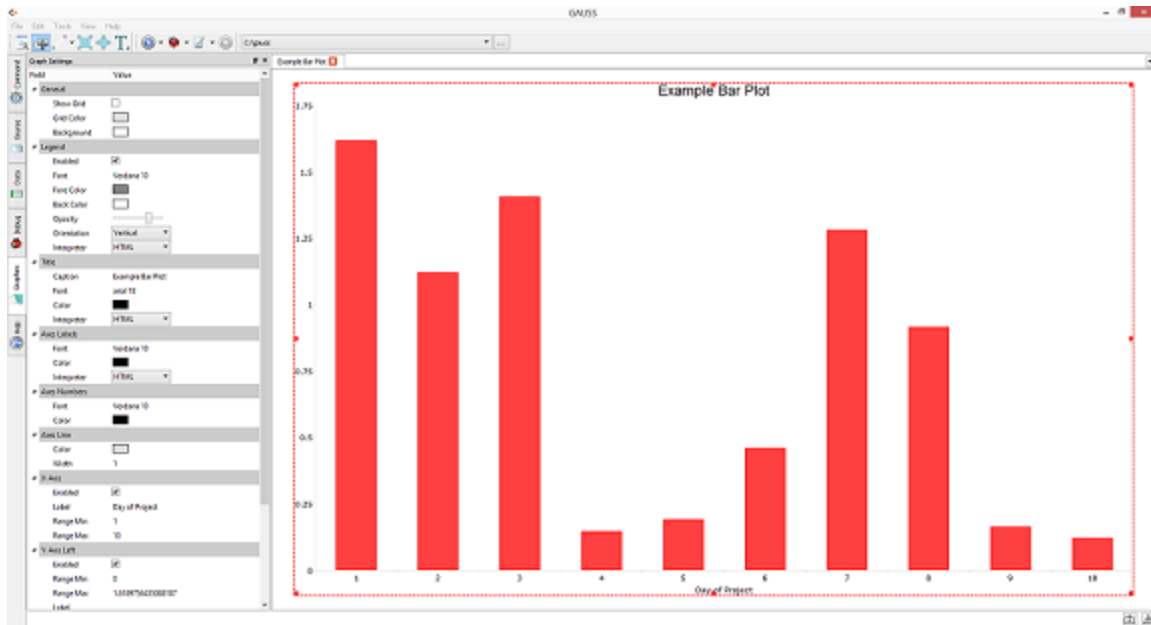

Figure 6.9: Example Bar Plot

Notice in the previous example that the first input to the **plotSetTitle** function is *&myPlot* and not just the variable name, *myPlot*. This is because all of the **plotControl** setting functions take a structure pointer as the first input. The ampersand (&) in front of the variable name makes the input argument a pointer to that structure.

Passing structure pointers as arguments is much faster and more efficient than passing an entire structure. The rule for remembering when to use a **plotControl** structure pointer instead of **plotControl** structure is simple. If the function you are calling sets a value in the structure, you need to use a structure pointer. In all other cases, use the structure.

Fortunately the function completion tooltip in the **GAUSS** editor will indicate which is required, in case you forget this simple rule. For more information on structures and structure pointers see **STRUCTURES**, CHAPTER 16 .

6.5 Adding Data to Existing Plots

You may add additional data to 2-D plots. This functionality is accessed through the following:

plotAddArea	Adds a cumulative area plot an existing 2-D graph.
plotAddBar	Adds a set of bars to an existing 2-D graph.
plotAddBox	Adds a box plot to an existing 2-D graph.
plotAddHist	Adds a histogram to an existing 2-D graph.
plotAddErrorBar	Adds an error bar or a set of bars to an existing 2-D graph.
plotAddHistF	Adds a frequency histogram to an existing 2-D graph.
plotAddHistP	Adds a percentage histogram to an existing 2-D graph.
plotAddPolar	Adds a polar plot to an existing polar graph.
plotAddScatter	Adds a scatter plot to an existing 2-D graph.
plotAddSurface	Adds a surface plot to an existing 3-D plot.
plotAddTS	Adds a time series plot to an existing time series graph.
plotAddXY	Adds an XY line to an existing 2-D graph.

Any 2-D plot type may be added to any other 2-D plot type, with the exception of contour plots, polar plots and time series plots. You may add a polar plot to a previous polar plot or a time series plot to a time series plot. Use the **plotAdd** functions only to add data to an existing graph. Do not attempt to use them to create a new graph.

Example

```
//Create and plot multivariate random normal data
rndseed 19823434;
cov = { 1 0.9, 0.9 1 };
mu = { 2, 2 };
y = rndMVn(300, mu, cov);
plotScatter(y[.,1], y[.,2]);

//Create line coordinates and add to scatter plot
x = { -0.3, 4.8 };
y2 = { 0, 4.3 };
plotAddXY(x, y2);
```
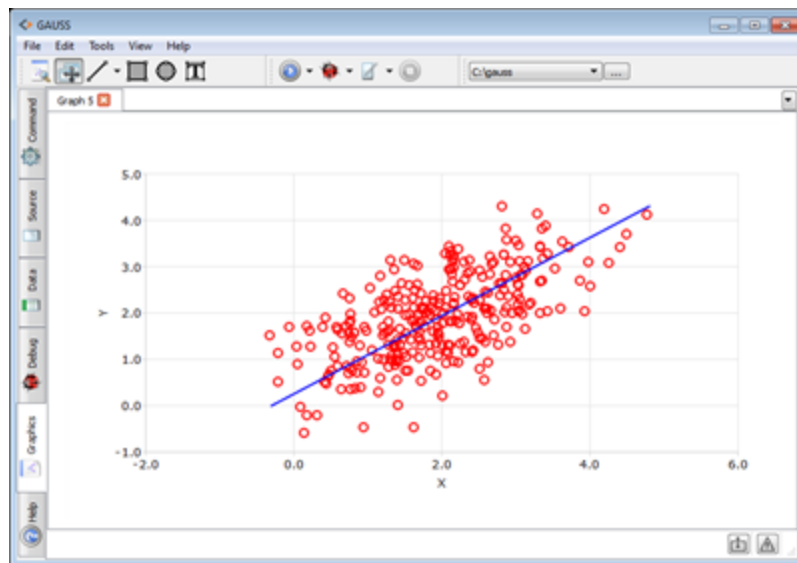

Figure 6.10: Example Adding XY Plot to a Plot

6.5.1 Styling and the plotAdd functions

Plot-wide attributes

The **plotAdd** functions will not make any styling changes to the existing plot-wide attributes, such as the title, axes labels, grid, etc. It will, however, increase the scale of the view if needed to accommodate the new curve.

Curve level attributes

If the **plotAdd** call does not use a **plotControl** structure, attributes for the added curves such as line or fill color, symbol style and the curve's legend text will be controlled by the preferences settings. The new curve will be treated as an additional series in sequence with any curves that already exist on the plot. For example if you draw a scatter plot with one column of data and then add another column of data with **plotAddScatter** and do not pass in a **plotControl** structure, the second set of scatter points will use the scatter settings from "Tools->Preferences->Graphics->Scatter->Series 2".

```
//Create and plot multivariate random normal data
rndseed 19823434;
cov = { 1 0.9, 0.9 1 };
mu = { 2, 2 };
y = rndMVn(300, mu, cov);

//Use scatter "Series 1" settings
plotScatter(y[.,1], y[.,2]);

cov = { 1 0, 0 1 };
mu = { -2, 2 };
y2 = rndMVn(300, mu, cov);

//Use scatter "Series 2" settings
plotAddScatter(y2[.,1], y2[.,2]);
```
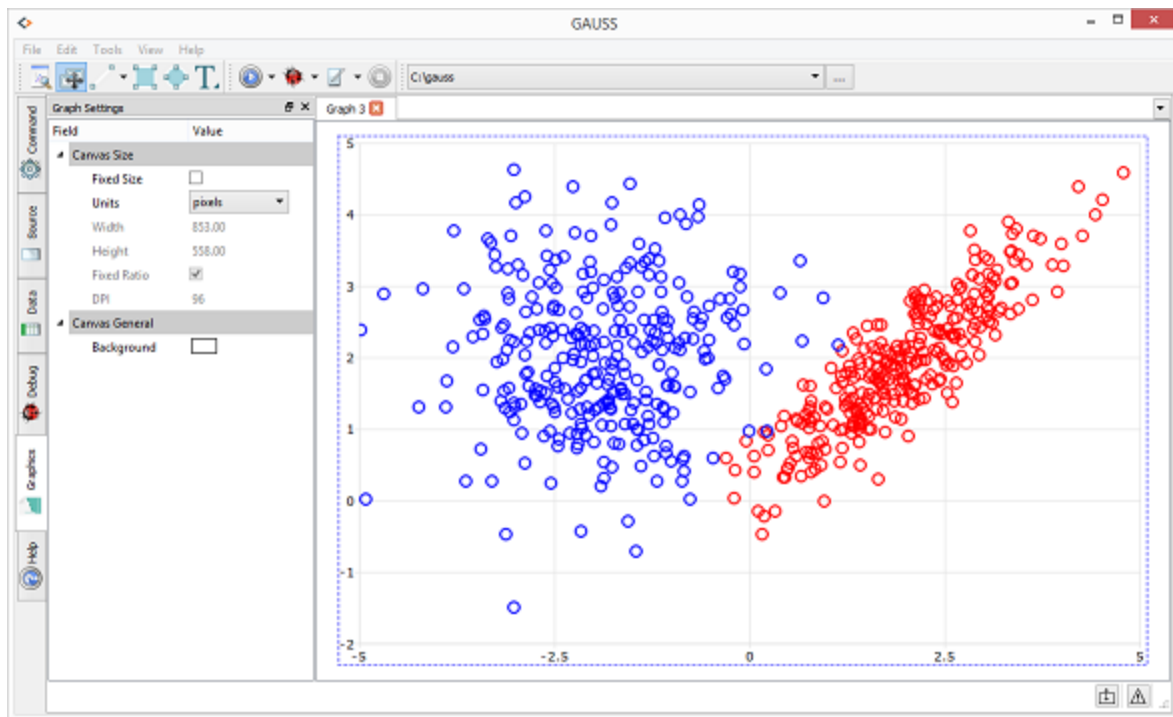
Figure 6.11: Example Adding Scatter Plot to a Plot

If you pass in a **plotControl** structure to the **plotAdd*** call, GAUSS will apply the first series curve settings in the **plotControl** structure to the first series of data that is added. For example:

```
//Create three multivariate random normal data set
rndseed 19823434;
cov = { 1 0.9, 0.9 1 };
mu = { 2, 2 };
y1 = rndMVn(300, mu, cov);

cov = { 1 0, 0 1 };
mu = { -2, 2 };
y2 = rndMVn(300, mu, cov);
```

```
cov = { 1 0.7, 0.7 1 };
mu = { 2, -2 };
y3 = rndMVn(300, mu, cov);

struct plotControl myPlot;
myPlot = plotGetDefaults("scatter");

//Use the Series 1 line color settings
//set in 'myPlot' by 'plotGetDefaults'
plotScatter(myPlot, y1[.,1], y1[.,2]);

plotSetLineColor(&myPlot, "purple"$|"blue");

//Use the Series 1 line color settings
//in the plotControl structure, purple
//for data set y2, use blue for data set y3

plotAddScatter(myPlot, y2[.,1]~y3[.,1], y2[.,2]~y3[.,2]);
```
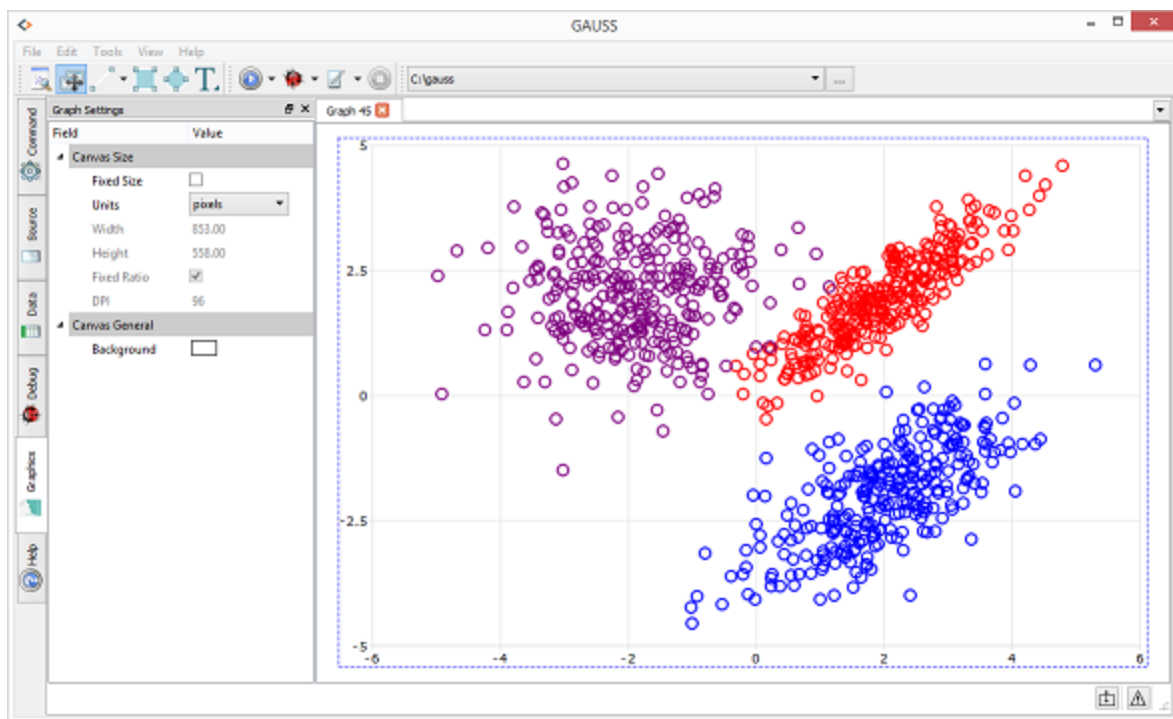
Figure 6.12: Example of plotControl for Adding Scatter Plot to a Plot

The above code first creates a scatter plot of data set *y1*. It then changes the line color settings inside the **plotControl** structure to purple for the second set of scatter points and blue for the third set of scatter points. Finally, it adds the scatter series *y2* which will be drawn in purple and *y3* which will be drawn in blue.

6.6 LaTeX in GAUSS Graphics

Graph text items such as titles, axes labels, legend labels and text boxes can interpret LaTeX. **GAUSS** uses the MathJax library for interpretation of LaTeX equations.

1. Since backslashes are used for escape sequences in strings, you must use double backslashes. For example, to add the greek letter beta to the X axis label:

   ```
   plotSetXLabel(&myPlot, "\\beta");
   ```

2. To use LaTeX, the text interpreter must be set to 'latex'. In a program, use the **GAUSS** function, plotSetTextInterpreter. The Graphics Editor has a drop-down menu, allowing you to interactively change the text interpreter type for text elements of a graph.

3. Strings interpreted as LaTeX are automatically in 'inline equation mode'. Therefore to add regular text, you must enclose the text inside of \\text{ }. For example:

   ```
   //Since LaTeX does not respect spaces inside of equa-
   tions
   //add a space after 'equation is' inside of the text sec-
   tion
   "\\text{The equation is } \\alpha + \\beta X"
   ```

4. Text inside of a '\\text{}' section will use the font set by the user as it would if the text interpreter were set to plain or HTML.

   ```
   //'Posterior distribution' will be in Times 24 point
   font
   //The greek lambda will be proportional in size to Times
   24, but will be in TeX font
   plotSetTitle(&myPlot, "\\text{Posterior distribution of
   } \\lambda_1", "Times", 24);
   ```

5. Equations will use the Tex font. They will be scaled with the size of the font set for the text section.

6. A list of supported LaTeX commands can be found here.

6.7 Creating Subplots

GAUSS allows you to create two types of graphs within graphs. The first is called a subplot. In **GAUSS** a subplot divides the canvas up into tiles of equal size and shape.

This functionality is controlled with the **plotLayout** function. The **plotLayout** function splits the canvas up into a specified number of rows and columns and also specifies into which cell to draw the next graph.

```
//Divide the canvas into 2 rows and 3 columns
and
//set the next drawn graph to be placed into the
//last position (which is the sixth cell);
//plotLayouts use row major ordering
plotLayout(2, 3, 6);
```


Figure 6.13: **plotLayout**s use row major ordering

After calling **plotLayout**, all future graphs will be placed in the cell location specified by the **plotLayout** call until **plotLayout** is called again with new parameters or a call to **plotClearLayout** is made. A call to **plotClearLayout** will cause the next graph to be drawn to take up the entire canvas.

This next graph, after a call to **plotClearLayout**, will either draw over any existing graphs or be drawn in a new window depending on your graphics preferences. This setting can be controlled from the "New graphs" setting, located under **Tools-> Preferences-> Graphics**. It may also be set with the **plotSetNewWindow** function.

Example

```
//Divide the canvas into 2 rows and 2 columns
and
//place each successive graph in the next
//available cell
```

```
for i( 1, 4, 1);
   plotLayout(2, 2, i);
   //Plot a percentage histogram with 10*i bins
of
   //random normal data
   plotHistP(rndn(1e5, 1 ), 10*i);
endfor;
//Clear the layout so future graphs are not
drawn in
//this layout
plotClearLayout();
```
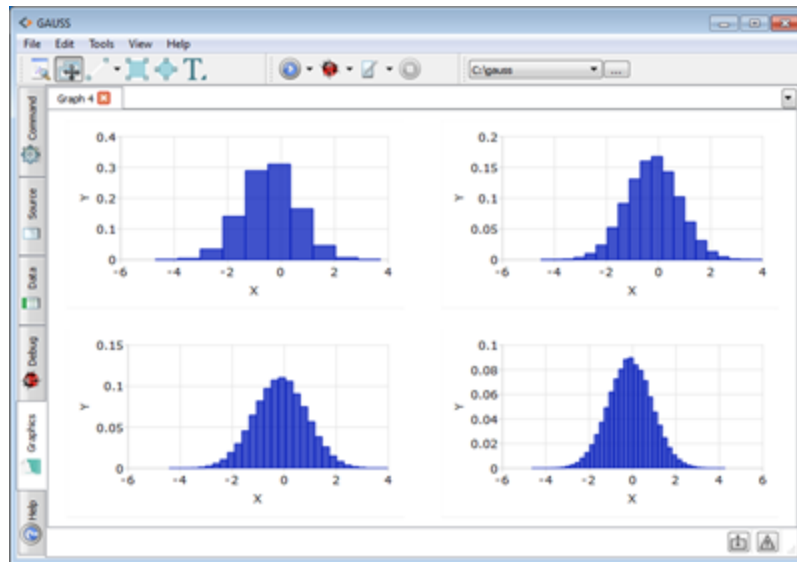

Figure 6.14: Example Subplot

6.7.1 Creating Mixed Layouts

In **GAUSS**, **plotLayouts** may not overlap. For this functionality use the **plotCus-tomLayout** function. However, you may divide the canvas into sections of different size as long as they do not overlap.

Example

```
//Divide the canvas into a 2x2 grid and fill in the first row
plotLayout(2, 2, 1);
plotHistP(rndn(1e5,1), 20);
plotLayout(2, 2, 2);
plotHistP(rndn(1e5, 1), 40);

//Divide the canvas into a 2x1 grid and fill in the
//bottom half, leaving the top section alone
plotLayout(2, 1, 2);
plotHistP(rndu(1e5, 1 ), 80);
```
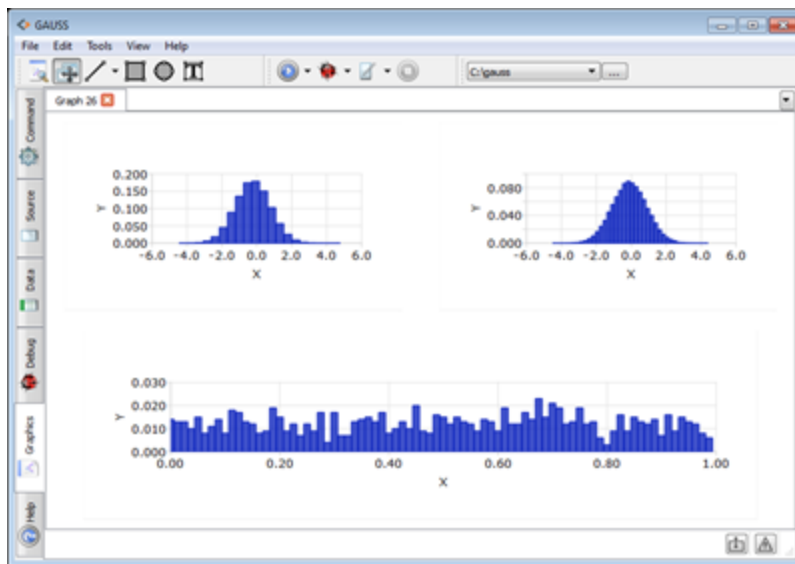

Figure 6.15: Example Mixed Layout

6.7.2 Creating Custom Regions

Custom regions in **GAUSS** are graphs inside of graphs. They may be placed in any location on the canvas and may be any size. This functionality is controlled by the **plotCustomLayout** function. It takes the location and size parameters for the custom region as a percentage of the entire canvas. As with **plotLayout**, these settings will be applied to the next drawn graph.

```
//Draw a custom region that starts 25% of the
//distance from the left edge of the canvas, 10%
//from the bottom of the canvas with a width of 40%
//of the canvas and a height of 28% of the canvas
plotCustomLayout(0.25, 0.1, 0.4, 0.28);
```

Unlike with **plotLayout**, these custom regions will not delete any graphs below them and can thus be used to place a small graph over a portion of a larger graph.

Example

```
//Draw a full size graph
rndseed 908734;
y = rndn(1e5, 1);
plotHistP(y, 30);

//Draw a custom region, close-up over the bottom left
// of the previously created graph
plotCustomLayout(0.1, 0.3, 0.2, 0.4);

//Set up plotControl struct with simple view
struct plotControl myPlot;
myPlot = plotGetDefaults("box");
plotSetLegend(&myPlot, "off");
plotSetGrid(&myPlot, "off");
plotSetXAxisShow(&myPlot, 0);
plotSetYAxisShow(&myPlot, 0);

//Create plot in custom region defined above
```

```
plotBox(myPlot, 0, y);

//Clear the layout, so future graphs will not be
//drawn in this custom region
plotClearLayout();
```
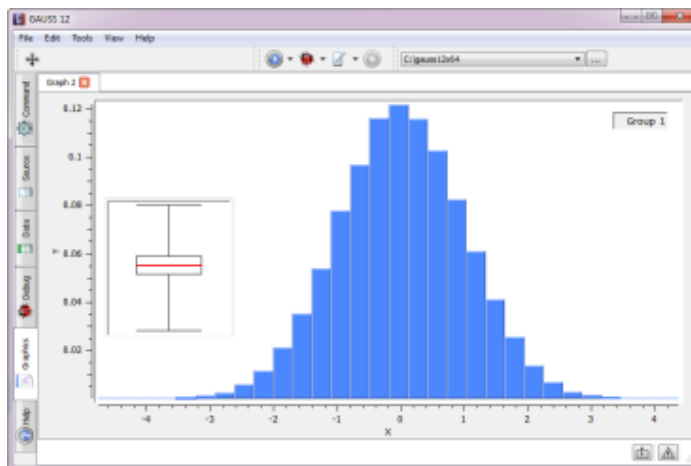

Figure 6.16: Example Custom Layout

6.8 Time Series Plots in GAUSS

 GAUSS provides the following functions to simplify the process of creating time series graphics:

plotAddTS	Adds a time series curve to a previously draw time series graph.
plotTS	Creates a graph of time series data.
plotSetXTicLabel	Controls the formatting and angle of X-axis tic labels for 2-D time series graphs.
plotSetXTicInterval	Controls the interval between X-axis tic labels and also allows the user to specify

the first tic to be labeled for 2-D time
series graphs.

To create a basic time series plot in **GAUSS**, you will use the **plotTS** function. In addition to the *y* data to plot, this function requires the following inputs:

dstart	Scalar, the starting date of the time series in DT scalar format.
frequency	Scalar, the frequency of the data per year. Valid options include:
	12 Monthly
	4 Quarterly
	1 Yearly data

Example

```
//Data to plot
y = rndu(24, 1);

//Start the series in January of 1987
dstart = 1987;

//Specify the data as monthly
freq = 12;

//Plot the data
plotTS(dstart, freq, y);
```

The example program above will create a graph of monthly data from January of 1987 through December of 1988. It should look similar to the graph below:
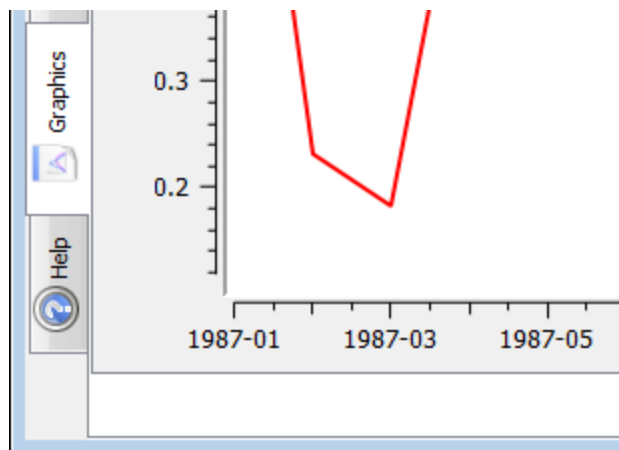
Figure 6.17: Close-up of monthly data plot

Understanding the dstart parameter

The *dstart* parameter is specified to be a scalar value in DT scalar format. In DT scalar format the leading four digits are the year and the next two digits, if present, represent the month. If the month information is not present, as in the example above, **GAUSS** will assume the first month of the year. This means that this:

```
2008
```

and this:

```
200801
```

will both be treated as specifying January of 2008.

To specify a starting quarter in DT scalar format, set the final digits of your *dstart* to be the first month of the quarter. The following represent 2008 Q1, 2008 Q2, 2008 Q3 and 2008 Q4 respectively.

```
200801
200804
```

```
200807
200810
```

6.8.1 Quarterly Example

Using what we have just learned, we will create a quarterly graph that starts at 2007 Q4 and runs for 16 quarters.

```
//Create 16 random data points
y = rndu(16, 1);

//Start the series in 2007 Q4
dstart = 200710;

//Specify the data as quarterly
freq = 4;

//Plot the data
plotTS(dstart, freq, y);
```

This should produce a graph that looks similar to this:
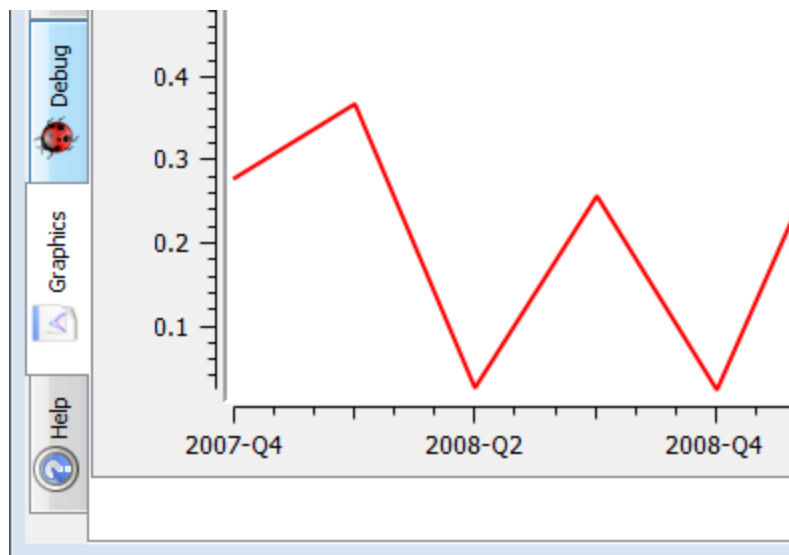
Figure 6.18: Close-up of quarterly data plot

6.8.2 Controlling Tic Label Locations

The first tic label on the X-axis of the graph that we drew at the end of the last section is located on the first data point, 2007 Q4. Let us suppose that we would prefer the graph to have tic labels only on the first quarter of each year. We can accomplish this by using the function **plotSetXTicInterval**.

plotSetXTicInterval has the following parameters:

interval	the number of data points between each X-tic that is labeled.
firstLabeled	the value of the first data point to be labeled.

Since we are working with quarterly data and would like to place X-tic labels on each year, we will set the interval parameter equal to four and we will set the

firstLabeled to be the first full year in our series, 2008. This will change our program to look like this:

```
//Declare and initialize plotControl structure
struct plotControl myPlot;
myPlot = plotGetDefaults("xy");

//Create 16 example data points
y = exp(seqa(0, 0.2, 16) + rndu(16, 1));

//Start the series in 2007 Q4 and specify
//quarterly data
dstart = 200710;
freq = 4;

//Start x-tic labels at 2008 and label every 4th
//data point
interval = 4;
firstLabeled = 2008;
plotSetXTicInterval(&myPlot, interval, firstLabeled);

//Plot the data
plotTS(myPlot, dstart, freq, y);
```

This code should create a graph with x-tic labels only on the first quarter of each year as you see below:
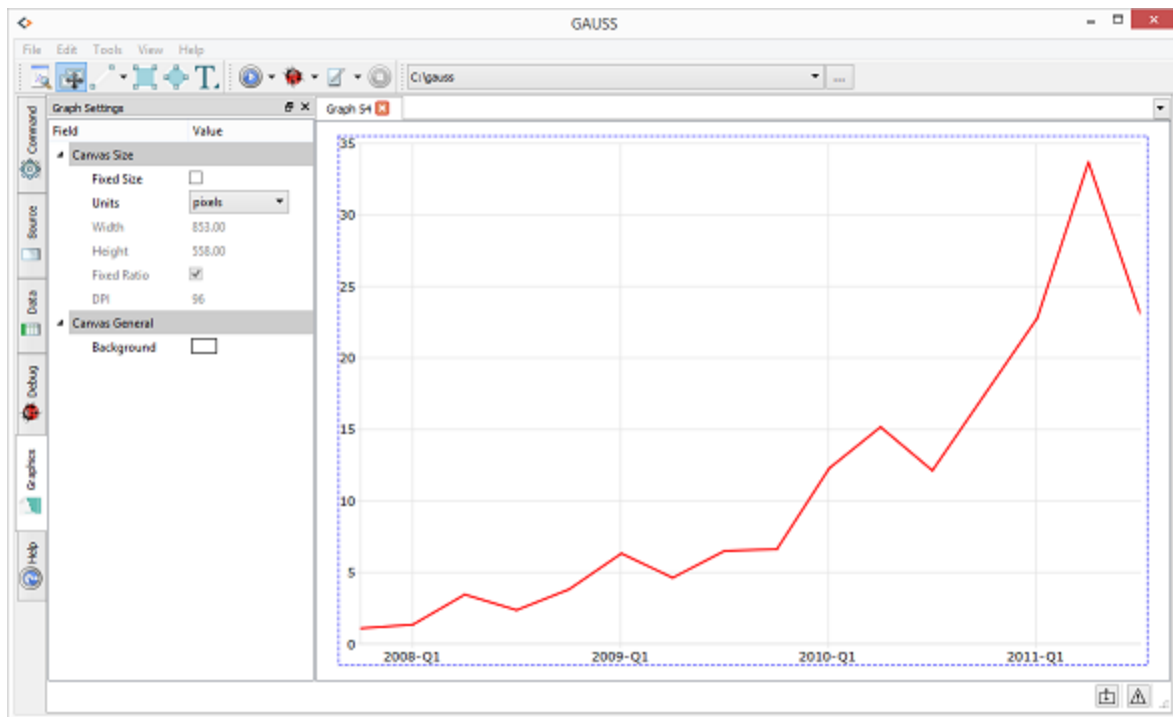
Figure 6.19: Quarterly data graph

We can see that the X-tics are in the correct location. However since we are only labeling each year, we may not want Q1 printed next to the year on each of the X-tic labels.

6.8.3 Tic Label Formatting

We can change the formatting of these labels and optionally the angle at which they are printed with the function **plotSetXTicLabel**. This function has the following arguments:

fmt	string, specifying the format with which to format the X-tic labels.
angle	scalar, the angle at which to print the labels

The *fmt* input can take many forms, see dttostr and strtodt for all options, but for this purpose we are only concerned with these three:

YYYY	print the four digit year.
QQ	print the quarter as Q1, Q2, Q3 or Q4.
MO	print the month as a two digit number.

These format specifiers can be placed in any order. You may place any characters in between them and they will be printed literally. For example if we start with 201204:

```
fmt = "YYYY-QQ";
```

will tell GAUSS to print the date as:

```
2012-Q2
```

whereas:

```
fmt = "MO/YYYY";
```

tells GAUSS to print the date as:

```
04/2012
```

Since we want to print only the four digit year on our x-tic labels, we will set:

```
fmt = "YYYY";
```

Giving us the following program:

```
//Declare and initialize plotControl structure
struct plotControl myPlot;
myPlot = plotGetDefaults("xy");

//Create 16 example data points
y = exp(seqa(0, 0.2, 16) + rndu(16, 1));
```

```
//Start the series in 2007 Q4 and specify
//quarterly data
dstart = 200710;
freq = 4;

//Start x-tic labels at 2008 and label every 4th
//data point
interval = 4;
firstLabeled = 2008;
plotSetXTicInterval(&myPlot, interval, firstLabeled);

//Label X-tics with only the 4 digit year
plotSetXTicLabel(&myPlot, "YYYY");

//Plot the data
plotTS(myPlot, dstart, freq, y);
```

This final program should yield a graph that looks similar to:
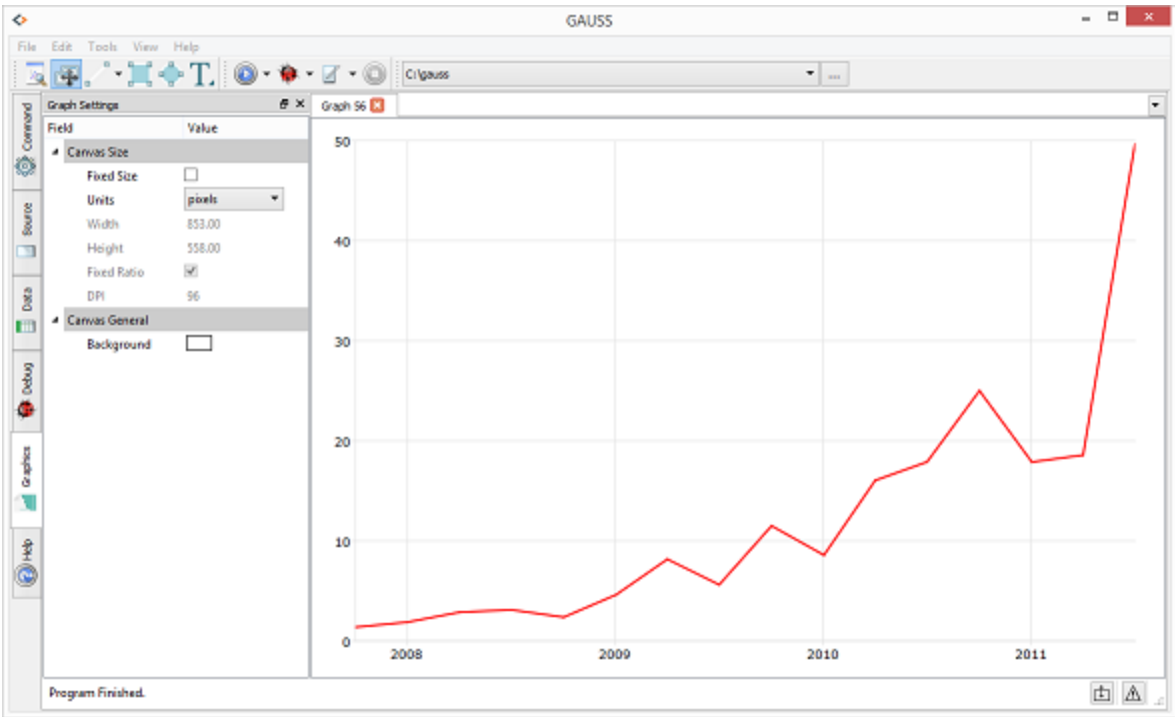
Figure 6.20: Quarterly data graph

6.9 Interacting with Plots in GAUSS

Once a graph has been created, **GAUSS** provides interactive zooming, panning, and the moving of legends and subplots as well as the ability to hide and restore individual curves on the graph. Plot rotation is also available for 3-dimensional plots.

6.9.1 Interacting with 2-D Plots

To interact with plots in **GAUSS**, first select the "Zoom/Pan Plot" button from the tool-bar. With this button selected, you will be able to perform actions on the view of a particular plot. Zooming is controlled by the mouse scroll wheel. Scroll the mouse wheel forward to zoom in. Pan and scroll is accomplished by drag-and-drop with the mouse. Note that upon zooming or panning, the axes will be automatically updated to reflect the new view.
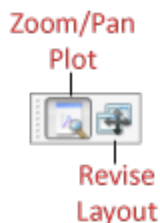
Figure 6.21: Zoom/Pan Plot Toolbar

Hide/Restore Curves

Each legend item is also a button that hides and restores individual curves, bars, etc. Click on a legend item to hide the corresponding curve. Notice that the axes will automatically scale to the remaining curves. Click the legend item again to restore the curve to the graph.
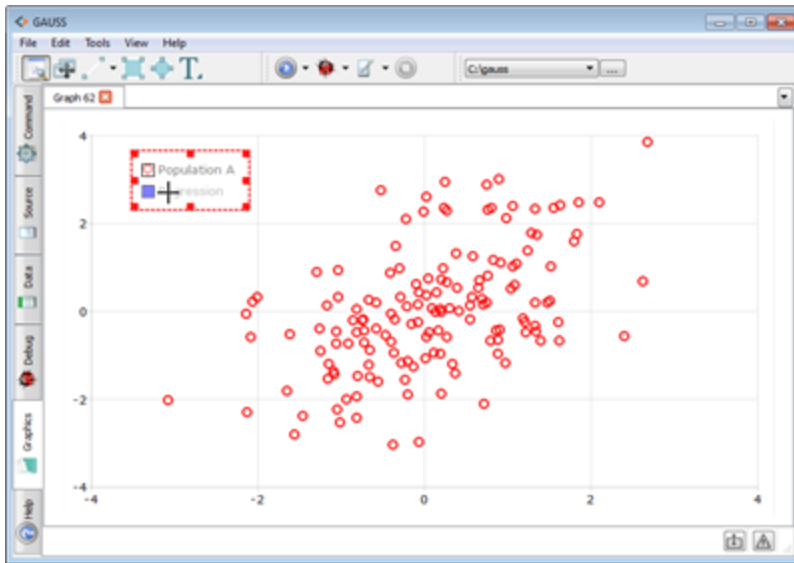
Figure 6.22: The legend acts as a button to hide curves

Relocating Graphic Items

To relocate subplots or legends first select the "Revise Layout" button. You may then drag and drop the item to its desired location. If you would like the legend to maintain its position relative to the graph when it is relocated, you can accomplish this by selecting the legend, right-clicking and selecting "regroup" from the context menu.
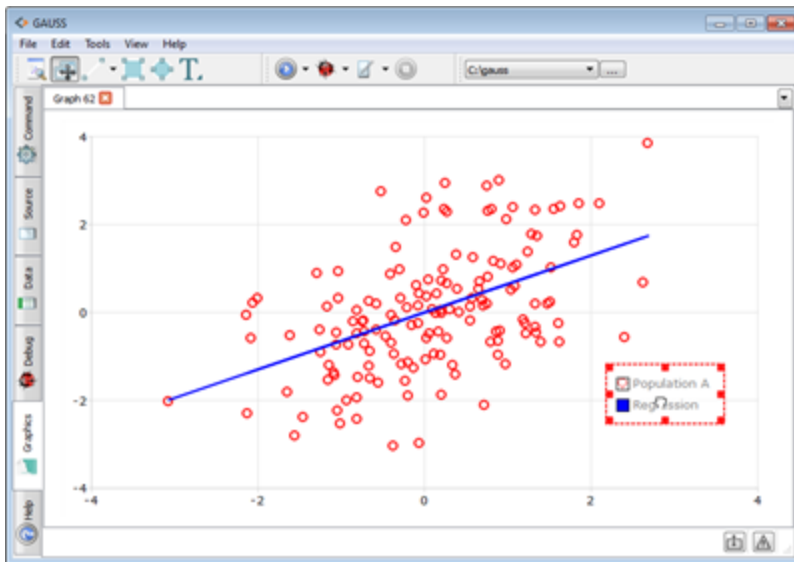
Figure 6.23: Drag and drop to relocate the legend

If you would like to change the z-order of an item in a layout, you may select it, right-click to bring up the context menu and then choose, either "Send to Front" or "Send to Back." Individual plots and their respective legends will both be affected.
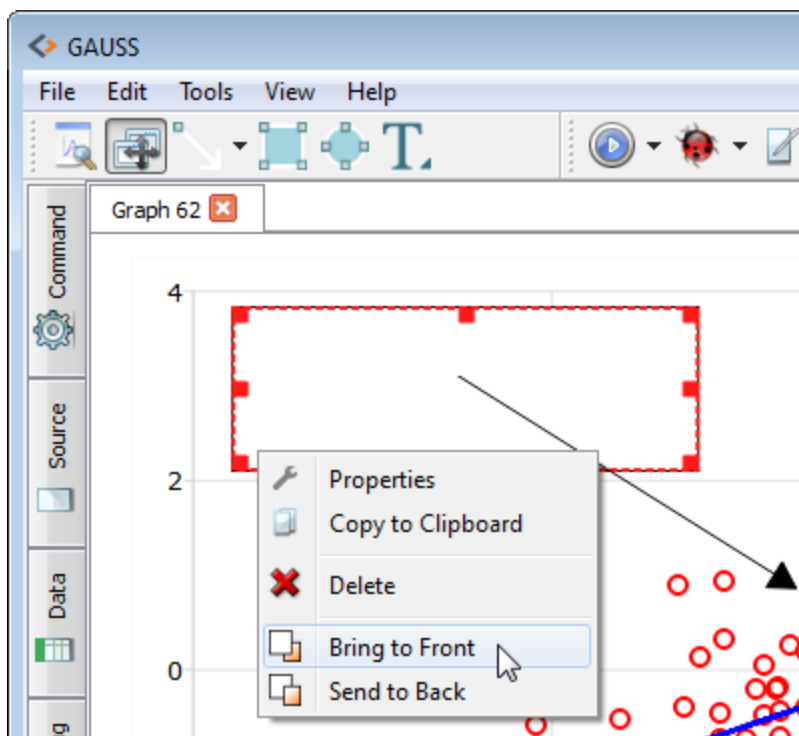
Figure 6.24: Bring textbox to top of z-order to cover arrow tail

6.9.2 3-D Plots
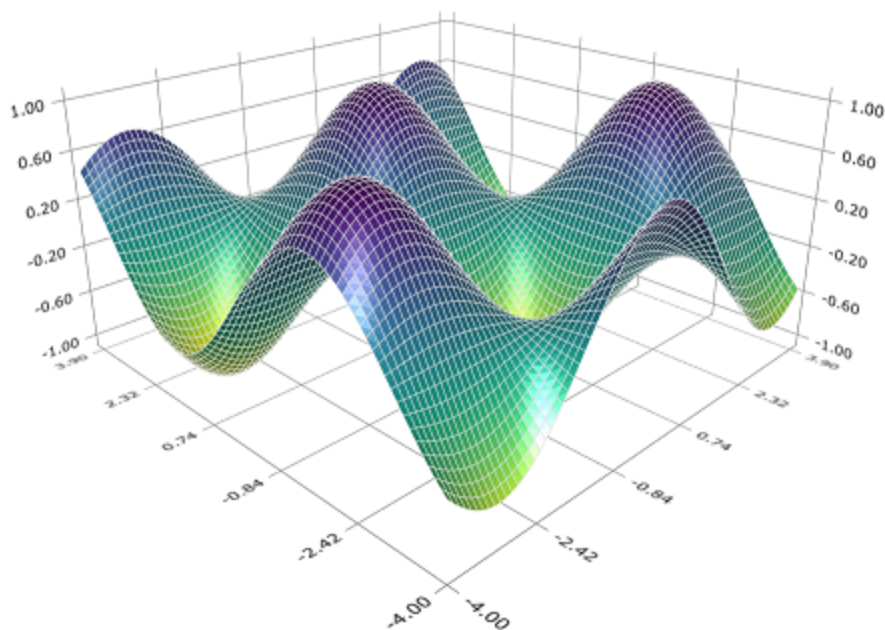
Figure 6.25: Surface plot
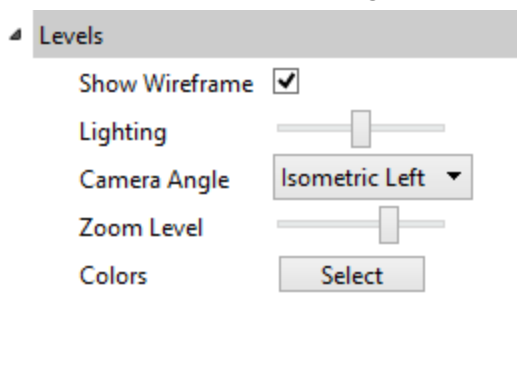
Figure 6.26: Surface plot interaction

Lighting

Use the mouse scroll wheel to adjust the lighting density of a 3-D graph.

Zoom

Zooming in and out on a 3-dimensional graph may be accomplished by use of the mouse scroll wheel.

Camera Angle Change

Use the drop down menu to change the camera angle view of a 3-D graph.

Colors Change

Use the select button to change the color of a specific area of a 3-D graph.

Rotate/Viewpoint Change

To rotate 3-dimensional plots and examine the graph from different viewpoints, right-click and drag with the mouse (two-finger click and drag on Mac).

6.9.3 File Export

When exporting a graph interactively, the first step is to set the size of the entire graph image, called the Canvas.

Customizing the Canvas

To access the Canvas settings, from the Graphics Page toolbar select "View->Graphics Settings". This will display the settings for whichever graphics object has focus. This could be the graph and its legend, an annotation object such as an arrow or textbox, or the Canvas. To give focus to the Canvas, click between the edge of your graph and the edge of the graphics tab.
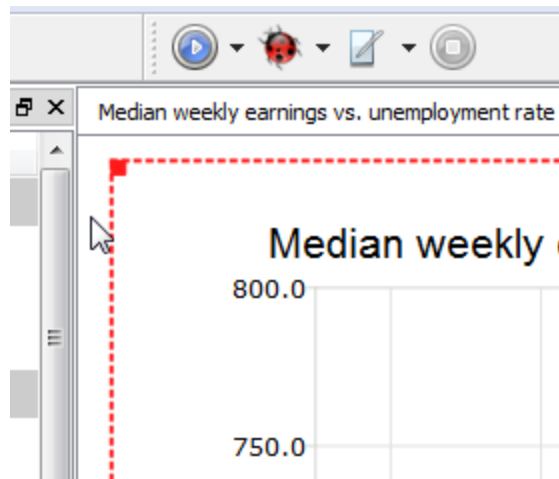
Figure 6.27: Select Canvas

Once you have opened the Canvas settings, set "Fixed Size" to "true". This will prevent the Canvas, and your graph, from being resized if you manually resize your GAUSS window. With this box is checked, the Canvas size will be controlled only by the Canvas Settings window.

Setting the size in terms of pixels

If you will be using the image in an electronic format, you will want to set the image size in terms of pixels. The "Units" Canvas setting controls whether the canvas will be sized in terms of pixels or a unit of physical size. By default, "Units" will be set to "pixels".

Double-click the numbers in the "Width" and "Height" boxes to edit the numbers. If the "Fixed Ratio" option is set to "true", then when you change the "Width" or the "Height", the other dimension will be changed to keep the height-to-width ratio the same.
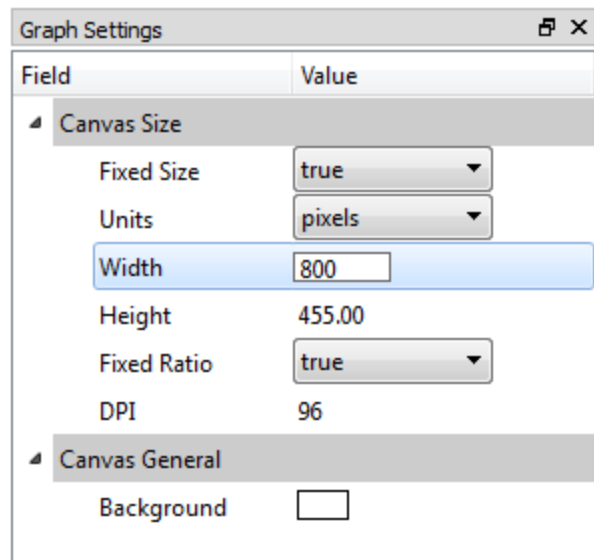
Figure 6.28: Control canvas size

Setting the size in terms of physical size (inches, millimeters)

If you are exporting an image for print, you may want to set the image size in terms of a physical size. To accomplish this, you will need to set the "Units" to either "millimeters" or "inches" and you will have to set "Width", "Height", and the "DPI" or dots per inch. This will allow GAUSS to create an image with enough pixels for your image to be printed at your desired size with the required dots per inch.

Make any final adjustments

As you will see, when you change the Canvas dimensions in GAUSS, the graph image will be adjusted to reflect those changes. This gives you a live preview of what your exported image will look like. After you have changed the overall size of your Canvas, you may want to make some adjustments to the size of fonts or lines on your graph to make them proportional to the new size of the whole. Click on the graph to display the Graph Settings in the Graphics Editor window, then make any necessary adjustments.

Exporting the file

Once you have the image as you would like it, select File->Export Graph from the main application menu and then select a filename, image type, and folder in which to save your file.

Copy and paste graphs into other programs

To copy and paste a graph from GAUSS to another program such as PowerPoint® or Word®, right click on the graph inside of GAUSS and select "Copy to clipboard". Next, open the other program and "paste" the graph at an appropriate location into your file.

Note that when you copy an image to the clipboard, it becomes a raster image, which means that resizing may decrease the image quality. Therefore, it is recommended that you first appropriately size the image in GAUSS before copying the image to the clipboard.

6.9.4 Saving Graphs

Graphs may be saved and then reloaded later. To save a graph, select the graph you would like to save and then select **File->Save Graph** from the main menu. **GAUSS** graphs are saved in a file with JSON format and a .plot extension.
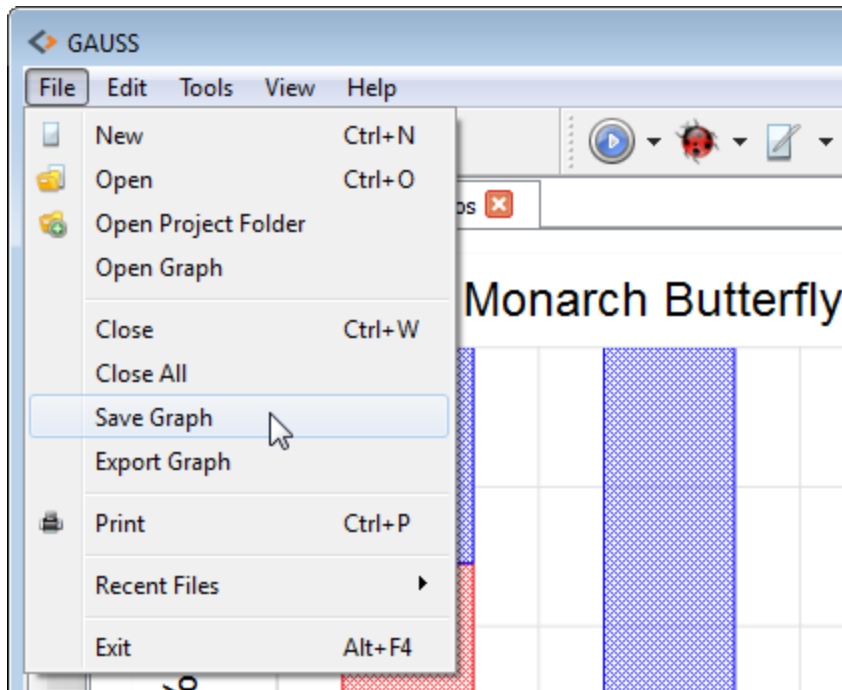
Figure 6.29: Save Graph menu

Graphs may also be saved programmatically with the **plotSave** function.

6.10 Adding Annotations Programmatically in GAUSS

In addition to adding annotations such as text boxes, arrows and lines interactively with the Graphics Editor, you may also add annotations in your GAUSS program. This functionality is provided through the following functions:

plotAddArrow	Adds arrows or lines to the last created graph.
plotAddShape	Adds ellipses and rectangles to the last created graph.

plotAddTextbox   Adds a textbox to the last created graph.

Basic rules

1. As you can see from the function prefix, plotAdd, these functions are meant to add to a graph that already exists. They will not create a new graph.

2. Added annotations will not expand the range of the X and Y axis so that they will fit inside the visible axes. This allows you to place annotations outside the axes or between different subplots.

6.10.1 Basic usage

Like the plotting functions, the graph annotation functions can be called without any styling options specified, or the user can pass in a control structure which will set the styling for the annotation. Here is a basic example of adding a text box:

```
//Create and plot some data
x = seqa(0, 0.01, 628);
y = cos(x)~sin(x);
plotXY(x, y);

//Text to place in the text box
text_string = "Sine is the derivative of cosine";

//Location of the top left corner of the text box
```

```
//will be the position (3,0.8) on the graph
x_start = 3;
y_start = 0.8;

//Add text box
plotAddTextbox(text_string, x_start, y_start);
```

The positioning of the annotation, set by *x_start* and *y_start* above, are in terms of the points on the graph to which the annotation is added.

Here is another basic example of adding an arrow:

```
//Arrow start and end locations
x_start = 3;
y_start = 0.8;
x_end = 1.07;
y_end = 0.5;

//Specify the head size
//at the end of the arrow
//in terms of points
head_size = 10;

//Add the arrow to the last created graph
plotAddArrow(x_start, y_start, x_end, y_end, head_size);
```

6.10.2 Creating multiple annotations with vector inputs

You may create multiple annotations with one function call by passing vector inputs to **plotAddArrow**, **plotAddShape** or **plotAddTextbox**. Here is an example adding three text boxes to a graph at once:

```
//Create x and y coordinates
x = { 67, 80, 46 };
y = { 2.9, 3.6, 1.4 };
```

```
//Create scatter plot
plotScatter(x, y);

//Create string array of text labels
countries = "France" $| "Germany" $| "Spain";

//Add labels to scatter points
plotAddTextbox(countries, x, y);
```

6.10.3 Customization with a plotAnnotation structure

After an annotation has been added to a graph, you can customize it with the Graphics Editor. If you would to set the style for your annotation inside of your GAUSS program, you can do this with a plotAnnotation structure. The steps for using a plotAnnotation structure are:

1. Declare an instance of a plotAnnotation structure

2. Fill in the plotAnnotation with default values

3. Set your desired options

4. Pass the structure in as the first argument to one of the annotation add functions

If you would like to create an instance of a plotAnnotation structure called myAnnotation and fill it in with default values, steps 1 and 2 above, you would do this:

```
//Declare 'myAnnotation' to be an instance
//of a 'plotAnnotation' structure
struct plotAnnotation myAnnotation;
```

```
//Fill 'myAnnotation' with default values
myAnnotation = annotationGetDefaults();
```

Now you are ready to apply your desired style preferences to 'myAnnotation' with the 'annotationSet' functions. Available functions include:

annotationSetBkd	Sets the color and transparency level of the annotation's background
annotationSetFont	Sets the font style, font size and font color for a text box
annotationSetLineColor	Sets the line color for an arrow or the border of a shape
annotationSetLineStyle	Sets the line style for an arrow or the border of a shape
annotationSetLineThickness	Sets the line thickness for an arrow or the border of a shape

You do not need to memorize this list or consult it when you are writing a program. Simply start typing "annotationSet" into the GAUSS editor and the autocomplete will provide the full list of available functions.

6.10.4 Using the annotationSet functions

The 'annotationSet' functions work in the same manner as the 'plotSet' functions. The first argument will be a pointer to a 'plotAnnotation' structure (which simply means that there is an ampersand (&) in front of the name). The remaining arguments to the function will be the settings to apply.

Example: Styling an ellipse

Let's say that we would like to highlight a section of the center of our graph. We can do this by adding a semi-transparent circle to the center of the graph.

```
//Declare instance of 'plotAnnotation' structure
struct plotAnnotation myEllipse;

//Fill in 'myEllipse' with default values
myEllipse = annotationGetDefaults();

//Set the ellipse color to be violet
//and make it 30% opaque
annotationSetBkd(&myEllipse, "violet", 0.3);

//Add ellipse to the center of the graph
x_start = 2;
y_start = 5;
x_end = 3;
y_end = 8;
plotAddShape(myEllipse, "ellipse", x_start, y_start, x_end,
y_end);
```

To see examples of styling text boxes or arrows see the documentation for **plotAddTextbox** and **plotAddArrow**.

7 Graphics Editing

GAUSS allows you to interactively change most all attributes of a created graph. First create a graph. For demonstration purposes you may run the example program ran-domwalk.e to create a graph. Next open the Graph Settings window if it is not open already, by selecting **View->Graph Settings** from the menu bar.
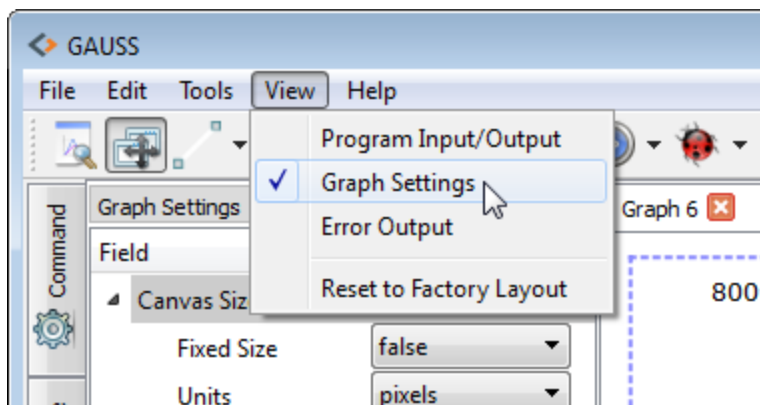
Figure 7.1: Open the Graph Settings Window

7.1 Changing Appearance

Click on the graph or a graph annotation object to view its settings in the Graph Settings Window. When an item is selected its outer border will be highlighted with a red, dash line. Interacting with the Graph Settings is simple. Single-click on color squares to open a color settings dialog. Single-click on drop-down menus to view the available options. Double-click on font names to open your systems font dialog and also double-click on text inputs, such as the "X-Axis" label, to enter new text.
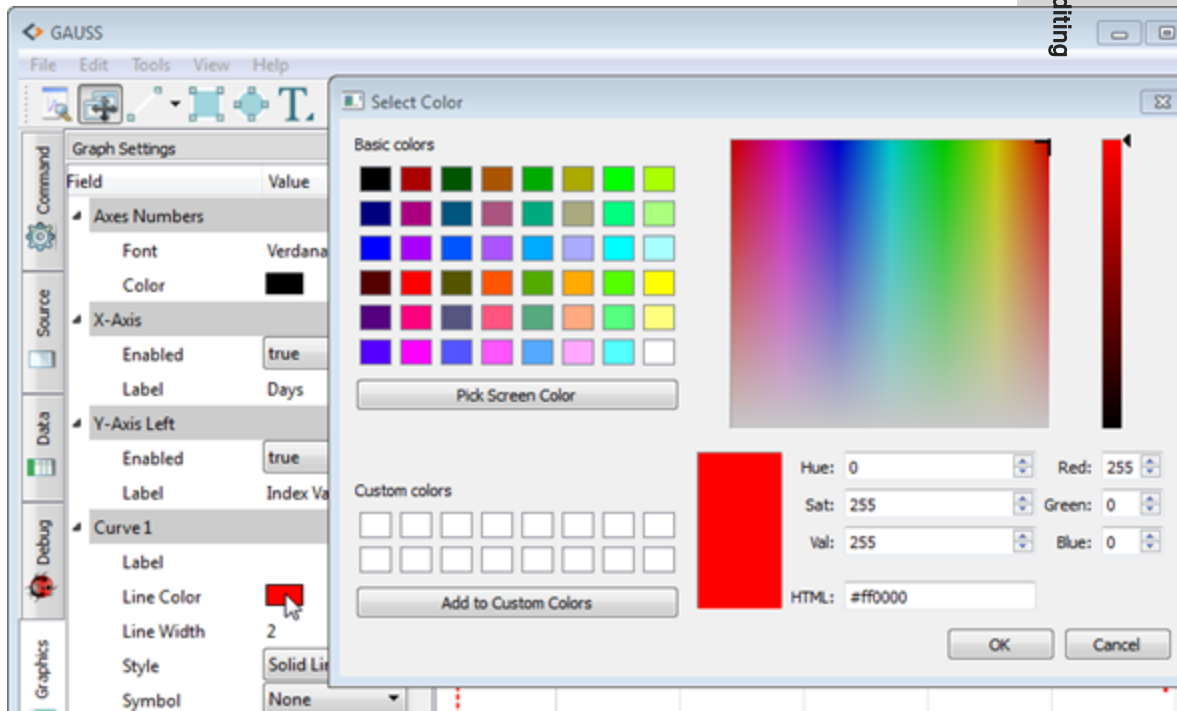
Figure 7.2: Color Palette

For example, click on the red color square next to "Line Color" under "Curve 1" to open the Color Palette dialog as shown above. Double-click the current text next to "Caption" under "Title" to enter text for the graph title as shown below.

Figure 7.3: Text Options

GAUSS User Guide

Double-click font names in the graph settings window to open a font dialog window.
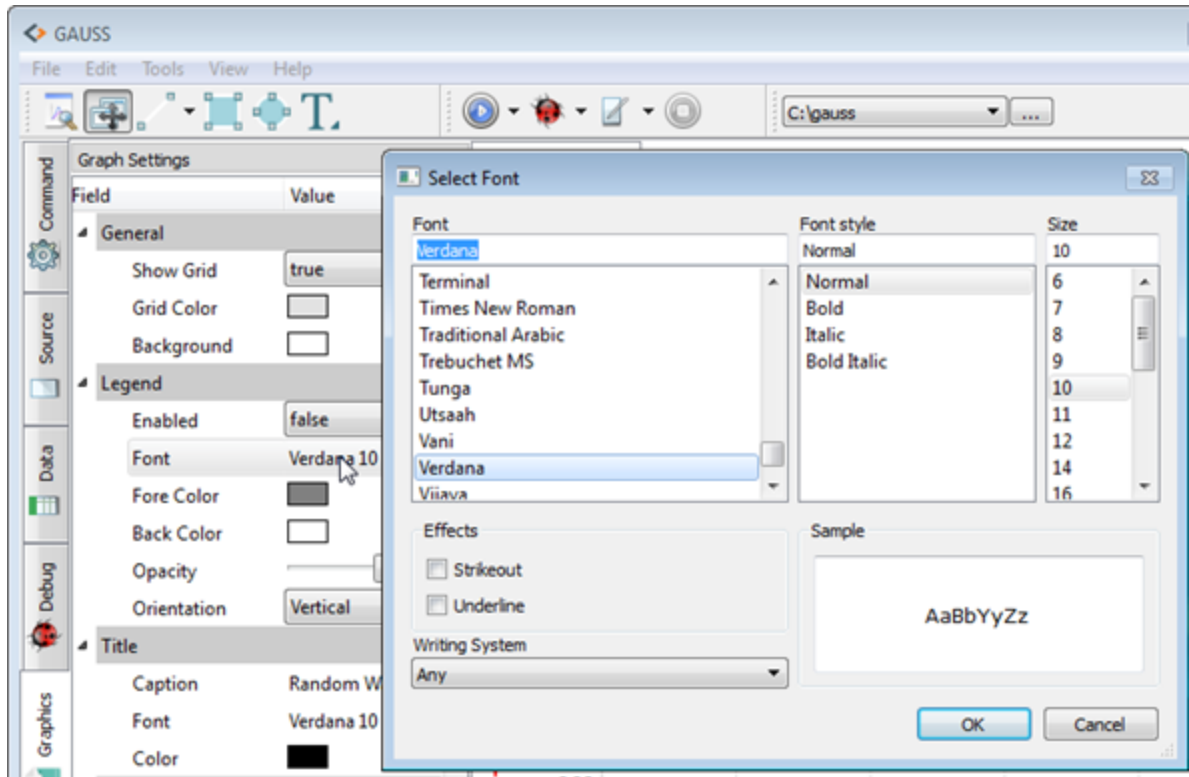
Figure 7.4: Font Options

7.2 Adding Items

The Graphics Page toolbar has buttons for adding

- Lines and arrows
- Rectangles

- Ellipses
- Text boxes

To add an item, click the appropriate toolbar button and then click and drag to create the object in your graphic window.
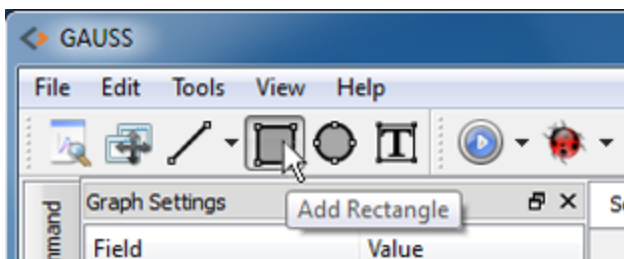
Figure 7.5: Select the item to add
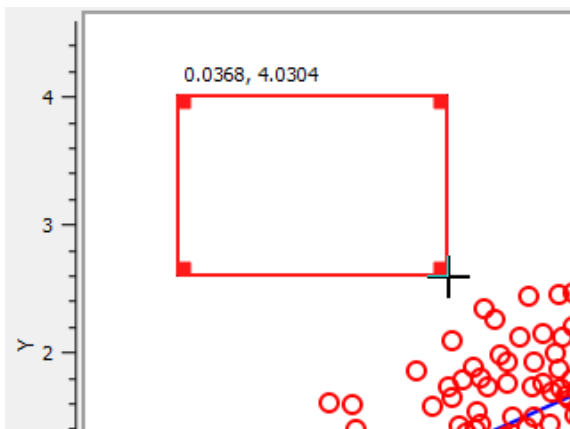
Figure 7.6: Click and drag to add the object

7.3 Adding Text to a Text Box

First add the textbox to the graph by selecting the textbox icon, which looks like a capital T, from the main GAUSS menubar. Then click and drag to size the textbox inside of your graph.

GAUSS User Guide
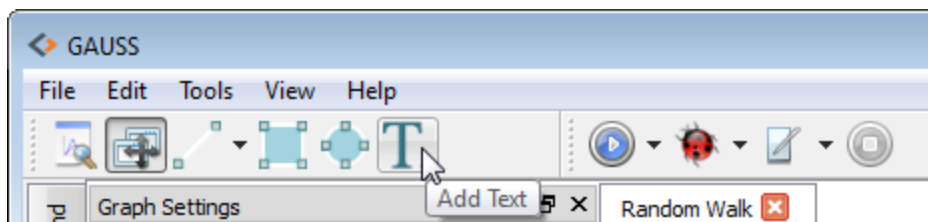
Figure 7.7: Color Palette

When you finish sizing the textbox, the Graphics Settings Window will open, displaying the graphical controls that allow you to customize the textbox. Towards the bottom of the graphics Settings Window will be the Content Input Pane. Its header will read "Content (HTML/Plain/LaTeX). Enter the text that you would like to appear inside the textbox in the Content Input Pane and click the "Apply" button. You may enter text which can include snippets of HTML to create subscripting, superscripting, Greek letters or other special characters and formatting.
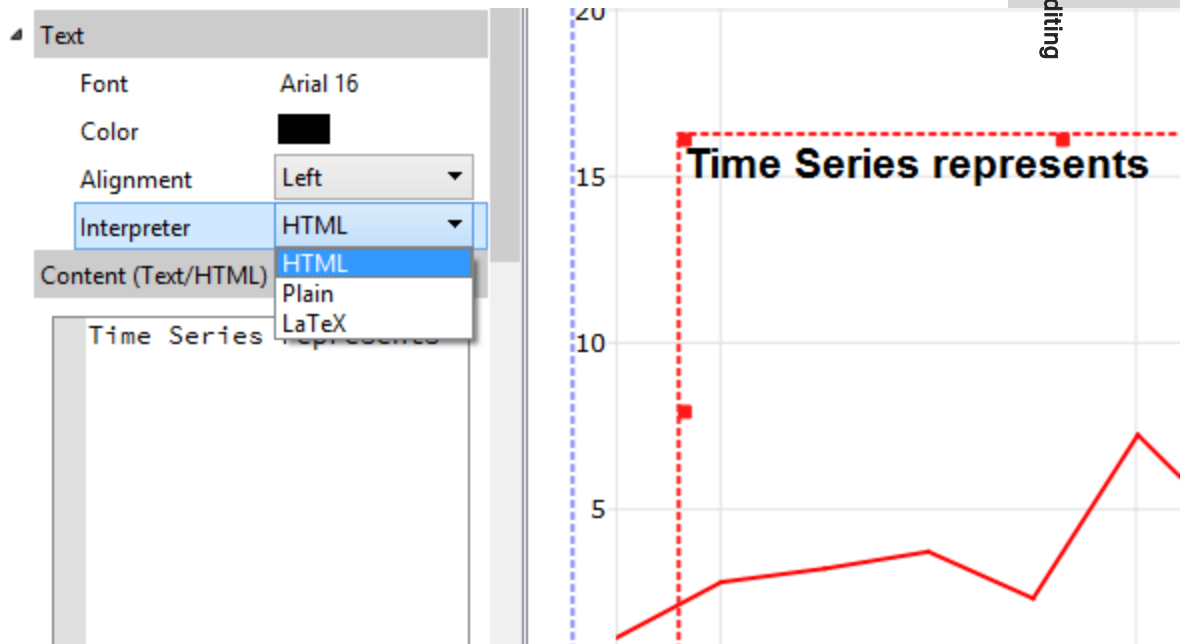
Figure 7.8: Entering content for a textbox

7.4 Which Object Will Be Edited?

If a graph or graphics object is actively selected, its border will be highlighted red. Changes made in the Graph Settings Window will apply to this selected object. If no object is actively selected, the Graph Settings Window will display the settings for the Canvas. The Canvas is the entire scene in which the graph or graphs are placed. Changing the Canvas properties is used to make final changes to the sizing of a graph before exporting.
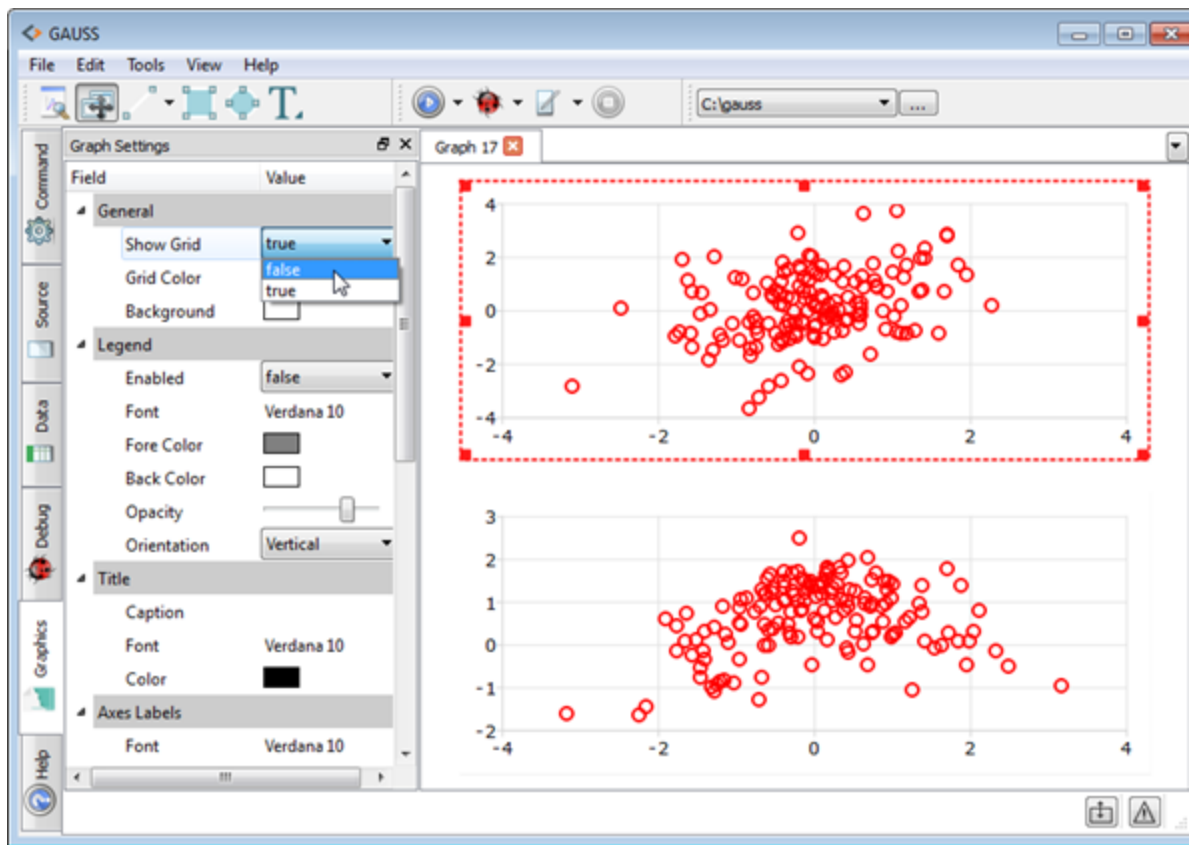
Figure 7.9: Highlighted Object

7.5 Moving and Resizing Objects

To move an object, including graphs, legends, textboxes and other annotations, hover your mouse over it. When your mouse pointer is a hand icon and the object you would like to move has a blue highlight around its edges, click and drag to move the object to its desired location.

Figure 7.10: Moving a graphic object

Hover over the corner of an object until your mouse pointer is two opposing arrows and the object has a blue edge highlight. Then click and drag to resize the object.
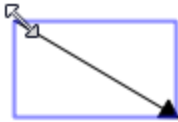
Figure 7.11: Resizing a graphic object

7.5.1 Note on Programmatic Annotations

Annotations that are added programmatically are tied to the coordinates to which they are placed. Items that are placed on the graph by two graph coordinate points such as arrow cannot be manually relocated. Items that are placed on the graph by only one coordinate point, such as textboxes, may be manually pivoted around the fixed point.

8 Using the Command Line Interface

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 process ID.
`-e expression`	Execute a **GAUSS** expression. This command is not logged when **GAUSS** is in batch mode.
`-o`	Suppress the sign-on banner (output only).
`-T`	Turn the dataloop translator on.
`-t`	Turn the dataloop translator off.

8.1 Viewing Graphics

Graphics using **plot*** functions can be saved or exported during a program run in
tgauss, using **plotSave**. See the plotSave Command Reference entry for usage.

The deprecated **Publication Quality Graphics** graphics generated .tkf files for graph-
ical output. The default output for these graphics is graphic.tkf. On Windows, you
can use vwr.exe to view the graphics file; on Linux/Mac, you can use vwrmp. 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.

8.2 Command Line History and Command Line Editing

When you run a command at the TGAUSS prompt, it is added to your command line history, which is stored in a file called .gauss_prompt_history in your $(HOME) directory on Linux or in your $(HOMEDRIVE)\\$(HOMEPATH) directory on Windows. A separate history for commands entered in the command line debugger is stored in a file called .gauss_debug_prompt_history in the same directory. By default, the last 500 commands executed at the TGAUSS and debugger command lines are stored in these files. You can change this number by changing prompt_hist_num in your `gauss.cfg` file. The following keystrokes are supported for movement and editing at the command line and for retrieving the command line history:

8.2.1 Movement

HOME or CTRL+A   Moves cursor to beginning of line
END or CTRL+E   Moves cursor to end of line

8.2.2 Editing

`DELETE` or `CTRL+D`	Deletes character at cursor
`BACKSPACE` or `CTRL+H`	Deletes character left of cursor
`CTRL+U`	Cuts all characters left of cursor
`CTRL+K`	Cuts all characters right of cursor, including cursor
`CTRL+X`	Cuts whole line
`ESC` (Win only)	Deletes whole line
`CTRL+V`	Pastes text from buffer to left of cursor
`CTRL+T`	Transposes character at cursor and character left of cursor

8.2.3 History Retrieval

`Up Arrow` or `CTRL+P`	Retrieves previous line in history
`Down Arrow` or `CTRL+P`	Retrieves next line in history
`PAGE UP` or `CTRL+W`	Retrieves previous line in history that matches text to left of cursor
`PAGE DOWN` or `CTRL+S`	Retrieves next line in history that matches text to left of cursor
`ALT+H` or `OPTION+H` (MAC only)	Prints prompt history to screen
`!!`	Runs last line in history
`! num`	Runs the *num* line in history

`!- num`	Runs the line `num` before current line in history; !-1 is equivalent to !!
`!text`	Runs last line in history beginning with `text`
`ALT+/` or `ALT+?` or `OPTION+/` (MAC only)	Prints help screen

Note that some of these keystrokes are mapped differently on different computers. For example, on some computers, SHIFT+RIGHT ARROW behaves the same as RIGHT ARROW, while ALT+RIGHT ARROW moves the cursor right one word. Therefore, multiple keystroke mappings have been supported to maximize the availability of these commandson any given machine.

8.3 Interactive Commands

This section discusses interactive commands available in TGAUSS.

8.3.1 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
```

8.3.2 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
```

8.3.3 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
```

8.3.4 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	Includes the following options:	
	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.	

8.4 Debugging

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

The format for `debug` is:

```
debug filename
```

8.4.1 General Functions

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

8.4.2 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.

8.4.3 Execution Functions

s *number*	Executes the specified number of lines, stepping into procedures.
n *number*	Executes the specified number of lines, stepping over procedures.
x *number*	Executes code from the beginning of the program to the specified line count, or until a breakpoint is hit.
g	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 argument is:

filename *line* *period*	The debugger will stop every *period* 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 *,, period*	The debugger will stop every *period* times it reaches any line in the named file.
line	The debugger will stop every

	period	*period* 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 period	The debugger will stop every *period* 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		Executes code to a specified line, procedure, or period 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.

8.4.4 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.
num, *num*	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.

8.4.5 Breakpoint Commands

lb		Shows all the breakpoints currently defined.
b		Sets a breakpoint in the code. The syntax for each optional argument is:
	filename line period	The debugger will stop every *period* 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 ,, period	The debugger will stop every *period* times it reaches any line in the named file.
	line period	The debugger will stop every *period* 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 period	The debugger will stop every *period* 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		Removes a previously specified breakpoint. The optional arguments are the same arguments as *b*, listed above.

8.5 Using the Source Browser in TGAUSS

To start the Source Browser in TGAUSS, type BROWSE followed by a symbol name. When the Source Browser is active, the prompt displays **Browse:**. **GAUSS** searches through all active libraries for the file in which the symbol is defined. If found, the file containing the source code is opened in the default editor.

Wildcard (*) searches can also be used. When using wildcard searches, each symbol that the string matches will be displayed on-screen in a numbered list. To select a specific command to view in the default editor, select the number from the list.

The Source Browser will remain active until you type CTRL+C to return to the **GAUSS** prompt.

9 Language Fundamentals

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.

Language
Fundamentals

9.1 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 '='.

9.2 Statements

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

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

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

```
print x*3;
```

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.

9.2.1 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 this code is in a loop, it will be executed each iteration of the loop.

Here are some examples of executable statements:

```
y = 34.25;

print y;

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

9.2.2 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.

9.3 Programs

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

9.3.1 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.

9.3.2 Secondary Sections

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

The secondary sections of the program can be left to the autoloader to locate and compile when they are needed. Secondary sections must have only procedure definitions and other nonexecutable statements.

`#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;
```

9.4 Compiler Directives

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

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**.
`#ifmac`	Compile code block if running Mac.
`#ifos2win`	Compile code block if running Windows.
`#ifunix`	Compile code block if running UNIX.
`#else`	Else clause for `#if-#else-#endif` code block.
`#endif`	End of `#if-#else-#endif` code block.
`#include`	Include code from another file in program.
`#lineson`	Compile program with line number and file name records.
`#linesoff`	Compile program without line number and file name records.
`#srcfile`	Insert source file name record at this point (currently used when doing data loop translation).
`#srcline`	Insert source file line number record at this point (currently used when doing data loop translation).

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

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

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**, Section 8.4 and see **Debugging Data Loops**, Section 25.3 . 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(/gauss/test.e)
```

9.5 Procedures

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

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

Procedures are isolated from the rest of your program and cannot be entered except by calling them. Some or all of the variables inside a procedure can be `local` variables. `local` variables 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**, CHAPTER 11 .

9.6 Data Types

There are four basic data types in **GAUSS**, matrices, N-dimensional arrays, strings and string arrays. It is not necessary to declare the type of a variable, but it is good pro-gramming 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 **Command Summary**, Section 28.1).

9.6.1 Constants

The following constant types are supported:

Decimal

Decimal constants can be either integer or floating point values:

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

Up to 18 consecutive digits before and after the decimal point(depending on the plat-form) are significant, but the final result will be rounded to double precision if neces-sary. The range is the same as for matrices (For details, see **Matrices**, Section 9.6.2

String

String constants are enclosed in quotation marks:

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

Hexadecimal Integer

Hexadecimal integer constants are prefixed with 0x:

```
0x0ab53def2
```

Hexadecimal Floating Point

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

```
0vfff8000000000000
```

9.6.2 Matrices

Matrices are 2-dimensional arrays of double precision numbers. All matrices are

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

```
[1,1] [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 elements of a **GAUSS** matrix 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	$4.19 \times 10^{-307} <= \|X\| <= 1.67 \times 10^{+308}$

Matrices with only one element (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

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

The statement

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

will result in

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

The statement

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

will result in

```
        1 1 1
x  =  1 1 1
        1 1 1
```

The statement

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

will result in

```
        0 0 0
x  =  0 0 0
        0 0 0
```

The statement

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

will result in

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

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 = 1+2i 3-4i
         5    0+6i
```

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 supported by several functions, including **rows** and **cols** and the
concatenation (~, |) operators.

```
    x = {};
    hsec0 = hsec;
    do until hsec-hsec0 > 6000;
```

```
 x = x ~ data_in(hsec-hsec0);
endo;
```

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

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 ver-
tical concatenation:

```
y = 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:

```
y = 2 4
    1 3
```

For more information, see **Operator Precedence**, Section 10.7 .

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 following code

```
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 the following prompt to be printed in the window:

```
-  (1,1)
```

indicating that the user should enter the [1,1] element of the matrix. Entering a number and then pressing ENTER will cause a prompt for the next element of the matrix to appear. Pressing *?* will display a help screen, and pressing *x* will exit.

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`.

9.6.3 Sparse Matrices

Many **GAUSS** operators and commands support the sparse matrix data type. You may use any of the following commands to create a sparse matrix:

denseToSp	Converts a dense matrix to a sparse matrix.
denseToSpRE	Converts a dense matrix to a sparse matrix, using a relative epsilon.
packedToSp	Creates a sparse matrix from a packed matrix of non-zero values and row and column indices.
spCreate	Creates a sparse matrix from vectors of non-zero values, row indices, and column indices.
spEye	Creates a sparse identity matrix.
spOnes	Generates a sparse matrix containing only ones and zeros
spZeros	Creates a sparse matrix containing no non-zero values.

See **SPARSE MATRICES**, CHAPTER 13 , for more information.

9.6.4 N-dimensional Arrays

Many **GAUSS** commands support arrays of N dimensions. The following commands may be used to create and manipulate an N-dimensional array:

aconcat	Concatenate conformable matrices and arrays in a user-specified dimension.
aeye	Create an N-dimensional array in which the planes described by the two trailing dimensions of the array are equal to the identity.
areshape	Reshape a scalar, matrix, or array into an array of user-specified size.

arrayalloc	Create an N-dimensional array with unspecified contents.
arrayinit	Create an N-dimensional array with a specified fill value.
mattoarray	Convert a matrix to a type array.

See **N-DIMENSIONAL ARRAYS**, CHAPTER 14 , for a more detailed explanation.

9.6.5 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	Convert 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	Convert 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	Split an Nx1 string vector into an NxK string array of the individual tokens.
strsplitPad	Split a string vector into a string array of the individual tokens. Pads on the right with null strings.
strtodt	Convert a string array of dates to a matrix in DT scalar format.
strtof	Convert a string array to a numeric matrix.
strtofcplx	Convert a string array to a 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="/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 sub-
stitution operator (^):

```
create f1=^dataset with ^vnames,0,2;
create f1=^dataset using ^cmdfile;
open f1=^dataset;
output file=^outfile;
load x=^datafile;
load path=^lpath x,y,z,t,w;
```

```
loadexe buf=^exefile;
save ^name=x;
save path=^spath;
dos ^cmdstr;
run ^prog;
msym ^mstring;
```

9.6.6 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	Reads 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 on 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 gives 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. The statements:

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

assign the values:

```
s = agepaysex
s = age   pay   sex
```

9.6.7 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.

9.6.8 Date and Time Formats

DT Scalar Format

The DT scalar format is a double precision representation of the date and time with up to 14 digits. Each group of digits represents a different aspect of the date and time such as the year or month. Using characters, to represent each digit in a DT scalar number would look like this:

```
YYYYMODDHHSS
```

Starting from the left: the first four digits represent the year; the fifth and sixth digits, if present, represent the month; the seventh and eight digits, if present, represent the day; the ninth and tenth digits represent the hour; the eleventh and twelfth digits, if present, represent the minutes and finally, the thirteenth and fourteenth digits represent the seconds.

For example, in DT scalar format, the number:

```
20120723143207
```

represents 14:32:07 or 2:32:07 PM on July 23, 2012. It is important to remember that the leading digits will always be the year. This becomes relevant if the DT scalar number contains fewer than 14 digits to the right of the decimal point. For example, the number:

```
201302
```

would represent February 2013, rather than the time 20:13:02 (or 8:13:02 PM).

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.

9.6.9 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 which do not correspond to a real number. These encodings are referred to as NaN's. NaN stands for Not A Number.

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

- $+\infty$ plus $-\infty$

- $+\infty$ minus $+\infty$

- $-\infty$ minus $-\infty$

- $0 * \infty$

- ∞/∞

- $0 / 0$

- 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. These procedures are not defined in the **Run-Time Library**. If you want to use them, you will need to define them yourself.

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 a platform-specific encoding.

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

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

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 values of the globals ___*INFn* and ___*INFp* to platform specific encodings.

9.7 Operator Precedence

The order in which an expression is evaluated is determined by the precedence of the operators involved and the order in which they are used. For example, the $*$ and $/$ operators have a higher precedence than the $+$ and $-$ operators. In expressions 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**, Section 10.7 .

The expression

```
-5+3/4+6*3
```

is evaluated as

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

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

The term

```
2^3^7
```

is evaluated

$$(2^3)^7$$

In the expression

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

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)	$((3.14^5) * 6)/(2 + (\textbf{sqrt}(3) / 4))$
-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!	$2 * (3!)$

9.8 Flow Control

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

9.8.1 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.)

9.8.2 Conditional Branching

The if statement controls conditional branching:

```
if scalar_expression;
  .
  .
  statements
  .
  .
elseif scalar_expression;
  .
  .
  statements
  .
  .
else;
  .
  .
  statements
  .
  .
endif;
```

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

GAUSS will test the expression after the `if` statement. If it is TRUE, then the first list of statements is executed. If it is FALSE, then **GAUSS** will move to the expression after the first `elseif` statement, if there is one, and test it. It will keep testing expressions and will execute the first list of statements that corresponds to a TRUE expression. If no expression is TRUE, then the list of statements following the `else` statement is executed. After the appropriate list of statements is executed, the program will go to the statement following the `endif` and continue on.

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

`if` statements can be nested.

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

9.8.3 Unconditional Branching

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

`goto`

A `goto` is an unconditional jump to a label with no return:

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

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

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

`gosub`

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

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

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

Since a subroutine is only an address marked by a label, there can be subroutines inside of procedures. The variables used in these subroutines are the same variables that are known inside the procedure. They will not be unique to the subroutine, but they may be locals that are unique to the procedure that the subroutine is in. (For details, see `gosub` in the GAUSS LANGUAGE REFERENCE.)

9.9 Functions

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

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

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

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

9.10 Rules of Syntax

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

9.10.1 Statements

A **GAUSS** program consists of a series of statements. A statement is a complete expression or command. Statements in **GAUSS** end with a semicolon 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 as follows.

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

9.10.2 Case

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

9.10.3 Comments

```
// This comments out all text between the '//' and the end
// of the line
/* This kind of comment can be nested */
@ We consider this kind of comment to be obsolete, but it
is supported for backwards compatibility. We do, however,
recommend replacing them with one of the other types of
comments @
```

9.10.4 Extraneous Spaces

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

```
print x y z;
```

In `print` and `lprint` statements, spaces can be used in expressions that are in parentheses:

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

Language Fundamentals

9.10.5 Symbol Names

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

9.10.6 Labels

A label is used as the target of a goto or a gosub. The rules for naming labels are 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;
```

9.10.7 Assignment Statements

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

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

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

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

is incorrect. It should be:

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

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

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

9.10.8 Function Arguments

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

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

9.10.9 Indexing Matrices

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

```
x = { 1 2 3,
 3 7 5,
 3 7 4,
 8 9 5,
 6 1 8 };
y = x[3,3];
z = x[1 2:4,1 3];
```

Vectors can be indexed with either 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.

9.10.10 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*:

```
//Create two global variables
height = { 64, 69, 72, 58, 68 };
weight = { 134, 177, 209, 123, 98 };

//Create a string array of variable names
mvec = "height" $| "weight";

//Calculate the maximum of 'weight'
i = 2;
max_wt = maxc(varget(mvec[i]));

//Calculate the maximum of 'height'
```

```
    i = 1;
    max_ht = maxc(varget(mvec[i]));
```

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

```
    proc (1) = 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 (1) = get(var_list,i);
        retp(varget(var_list[i]));
    endp;
```

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

```
    proc (1) = put(x,var_list,i);
        retp(varput(x,var_list[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.

9.10.11 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 pro-
cedures. *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 *i*th pointer, which
is just a number, is assigned to *f*. Then the second `local` statement tells the compiler
to treat *f* as a procedure from this point on; thus the subsequent statement **f**(x) is inter-
preted as a procedure call.

9.11 Formula String Notation

Formula strings allow you to represent a model or collection of variables in a compact
and intuitive manner, using the variable names in the dataset. In a model with a depend-
ent (or response variable), the dependent variable will be listed first, followed by a tilde
"~" and then the independent variables.

Simple linear regression

For example, to represent the model:

$$\text{weight} = \alpha + \beta_1 \times \text{height} + \epsilon$$

The correct formula string would be:

The formula: "weight ~ height"

Operators

Next we will show an example of multi-variate regression, but in order to do that we
must list the operators that are allowed in formula strings, they include:

OPERATOR	Meaning
~	Separates independent from dependent variables.
.	Include all variables, except for the dependent variable. Only variable names to the left of the "~" will be considered dependent variables.
+	Add a variable
-	Subtract a variable
1	The intercept term

Examples

For the following examples, we will assume that we are using a dataset, containing the following variables: admit, gre, gpa and rank.

9.11.1 Multiple variable regression

Model	Formula string
$\mathbf{admit} = \alpha + \mathbf{gre} \times \beta_1 + \mathbf{gpa} \times \beta_2 + \mathbf{rank} \times \beta_3 + \epsilon$	"admit ~ ."
	"admit ~ gre + gpa + rank"
$\mathbf{admit} = \alpha + \mathbf{gre} \times \beta_1 + \mathbf{gpa} \times \beta_2 + \epsilon$	"admit ~ . - rank"

"admit ~
gre +
gpa "

9.11.2 Model intercepts

As indicated in the operator table above, the "1" represents the intercept term in the model. If a formula string does not contain an intercept term, GAUSS will assume that one should be included in the model, by default. We can remove the intercept term by adding a "-1" to the formula string. The table below shows all of the formula strings from the previous table, with the intercept removed:

Model	Formula string
$\mathbf{admit} = \mathbf{gre} \times \beta_1 + \mathbf{gpa} \times \beta_2 + \mathbf{rank} \times \beta_3 + \epsilon$	"admit ~ . - 1"
	"admit ~ -1 + gre + gpa + rank"
$\mathbf{admit} = \mathbf{gre} \times \beta_1 + \mathbf{gpa} \times \beta_2 + \epsilon$	"admit ~ . - 1 - rank"
	"admit ~ -1 + gre + gpa "

9.11.3 Formulas without independent variables

Some computations, such as computing descriptive statistics, do not have a dependent variable. In these cases, the formula string will look much like the formula strings above after the tilde operator.

As above, these examples will assume that our dataset contains the variables: admit, gre, gpa and rank.

Variables to include	Formula string
admit, gre, gpa, rank	"."
	"admit + gre + gpa + rank"
gre, rank	". - admit - gpa"
	"gre + gpa "

9.11.4 Files without variable names (headers)

Some data files that can be used with formula strings do not have variable names. GAUSS matrix files (.fmt) that are created with the 'save' command are one example. In these cases, you may refer to columns 1, 2, 3, … , P, as variables X1, X2, X3, …, XP.

9.11.5 Fully functioning examples

For full examples that you can run, see the Command Reference section for one of the following functions: **dstatmt**, **glm**, **momentd**, **ols**, **olsmt** or **quantiled**.

9.11.6 Formula string rules

- If the dot operator '.' is used, it must be the first element in a formula without an independent variable, or the first element after the tilde operator in an equation with an independent variable.

```
//Correct
formula = "admit ~ . - gpa";
//INCORRECT!
formula = "admit ~ - gpa + .";
```

```
//Correct
formula = ". - gpa";
//INCORRECT!
formula = "- gpa + .";
```

- It is illegal to remove a variable with the minus operator '-', that does not exist in the dataset. This will cause an error.
- Variable names in formula strings are case sensitive.

10 Operators

Operators

10.1 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 con-
formable 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:

```
        1 3 2
    x = 4 5 1
        3 7 4

        2 4 3
    y = 3 1 4
        6 1 2

        3 7 6
    z = 7 6 5
        9 8 6
```

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

every element of x:

```
        1 3 2
x  =  4 5 1
        3 7 4


y  =  2


        3 5 4
z  =  6 7 3
        5 9 6
```

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

```
vector    matrix         result
1    →    2   4   3       3   5   4
4    →    3   1   4   =   7   5   8
3    →    6   1   2       9   4   5
```

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

```
vector   2   4   3


         ↓   ↓   ↓


matrix   7   2   4
         3   0   1
         5   3   2


         ↓   ↓   ↓


result   9   6   7
         5   4   4
         7   7   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.

10.2 Matrix Operators

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

Operators

10.2.1 Numeric Operators

For details on how matrix conformability is defined for element-by-element operators, see **Element-by-Element Operators**, Section 10.1 .

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, this 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 equation:

$$Ax = b$$

Linear equation solution is performed in the following cases:

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

$$x = \frac{A\prime b}{A\prime A}$$

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 comformability rules, see **Element-by-Element Operators**, Section 10.1 .)

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;

       4  5  6  8  10  12
y =    7  8  9 14  16  18
      12 15 18 16  20  24
      21 24 27 28  32  36
```

Horizontal direct product: *~

```
x =   1   2
      3   4

y =   5   6
      7   8

z = x *~ y;

z =   5   6 10 12
     21 24 28 32
```

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

10.2.2 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 $y = x.'*x$.

Horizontal concatenation: ~

```
z = x ~ y;

x =   1   2   3
      4   5   6

y =   7   8   9
      6   5   4

z = x ~ y;

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

Vertical concatenation: |

```
z = x | y;

x =   1   2   3
```

Operators

```
        4   5   6

y  =    7   8   9

z  =  x  |  y;

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

10.3 Relational Operators

For details on how matrix conformability is defined for element-by-element operators, see **Element-by-Element Operators**, Section 10.1 .

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 are valid. These operators are sign-insensitive for missings, NaN's, and zeros.

The string '$\$$' versions of these operators can also be used to test missings, NaN's and zeros. Because they do a strict byte-to-byte comparison, they are sensitive to the sign bit. Missings, NaN's, and zeros can all have the sign bit set to 0 or 1, depending on how they were generated and have been used in a program.

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

By this definition, then:

```
if x /= y;
```

is interpreted as: "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;
z = x $/= y;
z = x $!= y;
```

- Greater than or equal to

```
z = x >= y;
z = x ge y;
z = x $>= y;
```

- Greater than

```
z = x > y;
z = x gt y;
z = x $> y;
```

If the relational operator IS preceded by a dot '.', then the result will be a matrix of 1's and 0's, based upon an element-by-element comparison of x and y.

- Element-by-element less than

```
z = x .< y;
z = x .lt y;
z = x .$< y;
```

- Element-by-element less than or equal to

```
z = x .<= y;
z = x .le y;
z = x .$<= y;
```

- Element-by-element equal to

```
z = x .== y;
z = x .eq y;
z = x .$== y;
```

- Element-by-element not equal to

```
z = x ./= y;
z = x .ne y;
z = x .!= y;
z = x .$/= y;
z = x .$!= 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;
```

10.4 Logical Operators

The logical operators perform logical or Boolean operations on numeric values. On input a nonzero value is considered TRUE and a zero value is considered FALSE. The logical operators return a 1 if TRUE and a 0 if FALSE. Decisions are based on the following truth 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.

10.4.1 Scalar Operators

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;

Operators

- Equivalence

```
z = x eqv y;
```

10.4.2 Element-by-element Operators

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;
```

10.5 Other Operators

Assignment Operator

Assignments are done with one equal sign:

```
y = 3;
```

Comma

Commas are used to delimit lists:

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

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`), function (`fn`), or structure (`struct`). It is used when passing procedures or functions to other functions, when indexing procedures, and when initializing structure pointers. (For more information, see **Indexing Procedures**, Section 11.5 or **Structure Pointers**, Section 16.2 .)

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 used the same k to create z as follows:

```
z = "dog" $+ k[1,1];
```

The resulting z will be a string containing 'dogcatfish'.

String Array Concatenation

$ Vertical string array concatenation

```
x = "dog";
y = "fish";
k = x $| y;
print k;
    dog
    fish
```

$~ Horizontal string array concatenation

```
x = "dog";
y = "fish";
k = x $~ y;
print k;
    dog fish
```

String Variable Substitution

In a command like the following:

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

by default **GAUSS** will interpret **olsdat** as the literal name of the **GAUSS** data file 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 (^) in the current version.

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

10.6 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;
```

not

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

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

10.7 Operator Precedence

The order in which an expression is evaluated is determined by the precedence of the operators involved and the order in which they are used. For example, the $*$ and $/$ operators have a higher precedence than the $+$ and - operators. In expressions 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 table below. PLEASE NOTE: The transpose operator is listed with a precedence of 90. This applies for the transpose operation only. It does NOT apply for the expression:

```
X'X
```

Which is shorthand for:

```
X'*X
```

For a compound statement such as:

```
Z = (X'X)/(Y'X);
```

the parentheses are required.

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		

Operators

. $>   65 $>   55

11 Procedures and Keywords

Procedures are multiple-line, recursive functions that can have either local or global variables. Procedures allow a large computing task to be written as a collection of smaller tasks. These smaller tasks are easier to work with and keep the details of their operation from the other parts of the program that do not need to know them. This makes programs easier to understand and easier to maintain.

A procedure in **GAUSS** is basically a user-defined function that can be used as if it were an intrinsic part of the language. 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.

11.1 Defining a Procedure

A procedure definition consists of five parts, four of which are denoted by explicit
GAUSS commands:

1. Procedure declaration	`proc` statement	
2. Local variable declaration	`local` statement	
3. Body of procedure		
4. Return from procedure	`retp` statement	

5. End of procedure definition   `endp` statement

There is always one `proc` statement and one `endp` statement in a procedure definition. Any statements that come between these two statements are part of the procedure. Procedure definitions cannot be nested. `local` and `retp` statements are optional. There can be multiple `local` and `retp` statements in a procedure definition. Here is a basic example:

```
proc (1) = square(a);  //Procedure declaration for procedure
named, 'square'
                       //with 1 input and 1 return

    local s;           //Local variable declaration

    s = a .* a;        //Procedure body

    retp(s);           //Return from procedure

endp;                  //End of procedure definition
```

The procedure above, takes one input, a. The procedure squares the input a and assigns this value to a local variable called s and then returns s. Once defined, it could be called like any **GAUSS** function:

```
a = 2;
b = square(a);
```

Next let's look at a procedure with two inputs, three return values and a more complicated procedure body:

```
proc (3) = regress(x, y);
   local xxi,b,ymxb,sse,sd,t;
   xxi = invpd(x'x);
   b = xxi * (x'y);
   ymxb = y - x * b;
```

```
    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: is:

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

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

11.1.1 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.

11.1.2 Local Variable Declarations

The `local` statement is used to declare local variables. Local variables are variables known only to the procedure being defined. The names used in the argument list of the `proc` statement are always local. The format of the `local` statement is:

```
local x, y, f:proc, g:fn, z, h:keyword;
```

Local variables can be matrices or strings. If **:proc**, **:fn**, or **:keyword** follows the variable name in the `local` statement, the compiler will treat the symbol as if it were a procedure, function, or keyword respectively. This allows passing procedures, functions, and keywords to other procedures. (For more information, see **Passing Procedures to Procedures**, Section 11.4 .)

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**, Section 11.5 .)

11.1.3 Body of Procedure

The body of the procedure can have any **GAUSS** statements necessary to perform the task the procedure is being written for. Other user-defined functions and other procedures can be referenced as well as any global matrices and strings.

GAUSS procedures are recursive, so the procedure can call itself as long as there is logic in the procedure to prevent an infinite recursion. The process would otherwise terminate with either an **Insufficient workspace memory** message or a **Procedure calls too deep** message, depending on the space necessary to store the locals for each separate invocation of the procedure.

11.1.4 Returning from the Procedure

The return from the procedure is accomplished with the retp statement:

```
retp;
retp(expression1, expression2, arg1, arg2, argN,
    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 was 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.

11.1.5 End of Procedure Definition

The `endp` statement marks the end of the procedure definition:

```
endp;
```

An implicit `retp` statement that returns nothing is always generated here so it is impossible to run off the end of a procedure without returning. If the procedure was defined to return one or more objects, executing this implicit return will result in a **Wrong number of returns** error message and the program will terminate.

11.2 Calling a Procedure

Procedures are called like this:

```
dog(i, j, k);                 // no returns

y = cat(i, j, k);             // one return

{ x, y, z } = bat(i, j, k); // multiple returns
call bat(i, j, k);            // ignore any returns
```

Procedures are called in the same way that intrinsic functions are called. The procedure name is followed by a list of arguments in parentheses. The arguments must be separated by commas.

If there is to be no return value, 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, 2, 1);
```

is legal.

11.3 Keywords

A keyword, like a procedure, is a subroutine that can be called interactively or from within a **GAUSS** program. A keyword differs from a procedure in that a keyword accepts exactly one string argument, and returns nothing. Keywords can perform many tasks not as easily accomplished with procedures.

Procedures and Keywords

11.3.1 Defining a Keyword

A keyword definition is much like a procedure definition. Keywords always are defined
with 0 returns and 1 argument. The beginning of a keyword definition is the `keyword`
statement:

```
keyword
name(strarg);
```

name	Name of the keyword, up to 32 alphanumeric characters or an underscore, beginning with an alpha or an underscore.
strarg	Name that will be used inside of the keyword for the argument that is passed to the keyword when it is called. There is always 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**, Section 11.1 . 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.

11.3.2 Calling a Keyword

When a keyword is called, every character up to the end of the statement, excluding the leading spaces, is passed to the keyword as one string argument. 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
```

11.4 Passing Procedures to Procedures

Procedures and functions can be passed to procedures in the following way:

```
proc max(x,y);  // procedure to return maximum
   if x>y;
     retp(x);
   else;
     retp(y);
   endif;
endp;

proc min(x,y);  // procedure to return minimum
   if x<y;
     retp(x);
   else;
     retp(y);
   endif;
endp;

fn lgsqrt(x) = ln(sqrt(x));  /* function to return
                                log of square root */
proc myproc(&f1, &f2, x, y);
  local f1:proc, f2:fn, z;
  z = f1(x, y);
  retp(f2(z));
endp;
```

The procedure **myproc** takes 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.

11.5 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 to 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.

11.6 Multiple Returns from Procedures

Procedures can return multiple items, up to 1023. The procedure is defined like this example of a complex inverse:

```
proc (2) = cminv(xr,xi); /* (2) specifies number of
                                  return values */
   local ixy, zr, zi;
   ixy = inv(xr)*xi;
   zr = inv(xr+xi*ixy);   //real part of inverse
   zi = -ixy*zr;          //imaginary part of inverse
   retp(zr, zi);          //return: real part, imaginary part
endp;
```

It can then be called like this:

```
{ zr, zi } = cminv(xr, xi);
```

To make the assignment, the list of targets must be enclosed in braces.

Also, a procedure that returns more than one argument can be used as input to another procedure or function that takes more than one argument:

```
proc (2) = cminv(xr, xi);
   local ixy, zr, zi;
   ixy = inv(xr)*xi;
   zr = inv(xr+xi*ixy);   //real part of inverse
   zi = -ixy*zr;          //imaginary part of inverse
retp(zr, zi);
endp;

proc (2) = cmmult(xr, xi, yr, yi);
   local zr,zi;
   zr = xr*yr-xi*yi;
   zi = xr*yi+xi*yr;
  retp(zr, zi);
endp;
```

```
{ zr, zi } = cminv(cmmult(xr, xi, yr, yi));
```

The two returned matrices from **cmmult**() are passed directly to **cminv**() in the statement above. This is equivalent to the following statements:

```
{ tr, ti } = cmmult(xr, xi, yr, yi);
{ zr, zi } = cminv(tr, ti);
```

This is completely general so the following program is legal:

```
proc (2) = cmcplx(x);
   local r,c;
   r = rows(x);
   c = cols(x);
  retp(x, zeros(r,c));
endp;

proc (2) = cminv(xr,xi);
   local ixy, zr, zi;
   ixy = inv(xr)*xi;
   zr = inv(xr+xi*ixy);  //real part of inverse
   zi = -ixy*zr;         //imaginary part of inverse
retp(zr,zi);
endp;

proc (2) = cmmult(xr, xi, yr, yi);
   local zr, zi;
   zr = xr*yr-xi*yi;
   zi = xr*yi+xi*yr;
  retp(zr, zi);
endp;

{ xr, xi } = cmcplx(rndn(3,3));
{ yr, yi } = cmcplx(rndn(3,3));
{ zr, zi } = cmmult(cminv(xr, xi), cminv(yr, yi));
```

```
{ qr, qi } = cmmult(yr, yi, cminv(yr, yi));
{ wr, wi } = cmmult(yr, yi, cminv(cmmult(cminv(xr, xi), yr,
yi)));
```

11.7 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.

12 Random Number Generation in GAUSS

GAUSS provides a powerful suite of functionality for random number generation. The key features include:

1. Availability of several modern random number generators

2. Sampling of many distributions

3. Thread-safe random number generators

4. Parallel random number generation

Random Number Generation

12.1 Available Random Number Generators

1. KISS-Monster

2. MRG32k3a

3. MT19937: Mersenne-Twister 19937

4. MT2203: Mersenne-Twister 2203

5. Niederreiter

6. SFMT19937: Optimized Mersenne-Twister 19937

7. Sobol

8. WH: Wichmann-Hill

12.1.1 Choosing a Random Number Generator

Quality

What is a high quality random number generator? Pierre L'Ecuyer stated that "The difference between the good and bad random number generators, in a nutshell, is that the bad ones fail very simple tests whereas the good ones fail only very complicated tests that are hard to figure out or impractical to run." The most prominent testsuites for testing random number generators (RNGs) are Marsaglia's DIEHARD, DIEHARDER (an extended version of DIEHARD maintained by Brown), and L'Ecuyer's TestU01. These are all good testsuites, with BigCrush from TestU01 being the most expansive and stringent. Below is a summary of the performance of each of the RNGs included in **GAUSS** on the BigCrush test.

RNG	Number of streams	Period Length	Diehard	TestU01 BigCrush

KISS-Monster	1	10^8859	PASS	PASS
MRG32k3a	1	2^191	PASS	PASS
MT19937	1	2^19937	PASS*	2
MT2203	6024	2^2203	PASS*	4
Wichmann-Hill	273	2^42.7	PASS*	22

*The column under Test U01 indicates the number of test failures.

As we can see from the table above, all of the available RNGs (with the possible exception of Wichmann-Hill) provide very good to excellent performance. It is beyond the scope of this document to discuss the implications of particular test failures.

Speed

All of the RNGs in **GAUSS** are very fast. However, some are faster than others. Below is a chart of the relative speed of each random number generator in **GAUSS**.

1. SFMT19937
2. MT19937
3. MRG32K3A
4. Wichmann-Hill
5. Kiss-Monster

Beyond simply choosing the fastest available generators, two coding techniques can also cause your programs to run faster. The first is creating larger numbers of random deviates at a time. For example, creating one million random numbers in one call will be much faster than creating the same one million numbers through 100,000 calls that each create 10 numbers. Creating too many numbers at once can also slow down your program's performance if the resulting matrix takes up too much of your available RAM. The key point is to avoid creating large random matrices one or two numbers at a time. The second coding technique is incorporate multi-threading into your random number generation. This is covered in the next section of this chapter.

12.2 Thread-safe Random Number Generators

Each successive number in a pseudo-random sequence is computed based upon the generator's current state. The RNGs in **GAUSS** that take and return a state, such as **rndKMu** and **rndGamma**, are inherently thread-safe. These functions can be used independently in separate threads as long as the same state variable is not written to in more than one concurrent thread. Functions that do not take or return a state should not be used inside concurrent thread sets. For example, the following is not a legal program:

```
threadBegin;
    x = rndn(500, 1);
    y = myFunction(x);
threadEnd;
threadBegin;
    x2 = rndn(500, 1);
    y2 = myFunction(x2);
threadEnd;
threadJoin;
```

The problem with the above example is that implicit in each call to **rndn** is a write to the same global state. Note that **rndn**, **rndGam** and **rndu** all share the same global state. So replacing one of the above calls to **rndn** with a call to **rndu** would still result in an illegal program. To solve this problem, either move the call to **rndn** above the first threadBegin, like this:

```
x = rndn(500, 1);
x2 = rndn(500, 1);
threadBegin;
    y = myFunction(x);
threadEnd;
threadBegin;
    y2 = myFunction(x2);
threadEnd;
threadJoin;
```

Or pass in and return the state vector:

```
seed1 = 723193;
seed2 = 94493;
threadBegin;
    { x1, state1 } = rndn(500, 1, seed1);
    y1 = myFunction(x1);
threadEnd;
threadBegin;
    { x2, state2 } = rndn(500,1,seed2);
    y2 = myFunction(x2);
threadEnd;
threadJoin;
```

12.3 Parallel Random Number Generation

12.3.1 Multiple Stream Generators

Some of the RNGs in **GAUSS** are a collection of more than one independent stream of random numbers. A stream is an independent series of numbers with a length that is the period length of the generator. Each of these sequences or streams has the same statistical properties. However, not all of the RNGs in **GAUSS** offer this ability. The MT2203 and the Wichman-Hill are examples of RNGs with multiple streams. The MT2203 is a set of 6024 different streams; the Wichmann-Hill contains 273 streams. You can initialize any of these streams with the rndCreateState function, like this:

```
// Initialize two Wichmann-Hill streams
seed = 192938;
whState3 = rndCreateState("wh-3", seed);
whState117 = rndCreateState("wh-117", seed);

// Initialize two MT2203 streams
```

```
seed = 192938;
mt2203State9 = rndCreateState("mt2203-3", seed);
mt2203State21 = rndCreateState("mt2203-21", seed);
```

Once created, these states can be passed into any of the new random number functions to create random numbers. To reiterate, from each of these newly created states you can create a full period length of random numbers.

12.3.2 Block-skipping

Creating random numbers from multiple streams provides multiple sets of independent random numbers with similar statistical properties that can be used in different threads. However, these separate streams do not form a contiguous set of random numbers. If you would like to process a contiguous stream of random numbers in multiple threads, you can use block-splitting. Block-splitting involves splitting a large chunk of random numbers into smaller contiguous blocks which can each be processed by different threads. This can be accomplished with the rndSkip function.

Example

```
// Set seed and create new state vector
seed = 23423;
state1 = rndCreateState("mrg32k3a", seed);

// Advance state by 2
state2 = rndSkip(2, state1);
{ r, state1 } = rndn(4, 1, state1);

// Same sequence as after skipping 2
{ r2, state2 } = rndn(2, 1, state2);

          0.19971499
r =       0.14053340
         -0.53132702
          0.48660530
```

```
r2 = -0.53132702
        0.48660530
```

For example if your program required four hundred million random numbers, you could split it into four blocks of one hundred million numbers each. This can be accomplished in **GAUSS** using the function **rndStateSkip**.

Each successive thread in the above program will process a different chunk of the sequence of random numbers.

```
// Set seed and create initial state vector
seed = 2342343;
state1 = rndCreateState("mrg32k3a", seed);

// Create 3 additional state vectors each starting 1e8
// numbers forward from the state just above it
state2 = rndStateSkip(1e8, state1);
state3 = rndStateSkip(1e8, state2);
state4 = rndStateSkip(1e8, state3);

// Use each state vector in a separate, concurrent thread
threadBegin;
    { x, state1 } = rndGamma(1e8, 1, 2, 2, state1);
    y = myfunc(x);
threadEnd;
threadBegin;
    { x2, state2 } = rndGamma(1e8, 1, 2, 2, state2);
    y2 = myfunc(x2);
threadEnd;
threadBegin;
    { x, state3 } = rndGamma(1e8, 1, 2, 2, state3);
    y3 = myfunc(x3);
threadEnd;
threadBegin;
    { x, state4 } = rndGamma(1e8, 1, 2, 2, state4);
    y4 = myfunc(x);
```

```
threadEnd;
threadJoin;
```

References

1. (MRG32k3a) L'Ecuyer, P. "Good Parameter Sets for Combined Multiple Recurs-
    ive Random Number Generators." Operations Research, 47, 1, 159-164, 1999.

2. (TestU01) L'Ecuyer, P. and Simard, R. "TestU01: A C Library for Empirical Test-
    ing of Random Number Generators." ACM Trans. Math. Softw. 33, 4, Article 22
    , 2007.

3. (Wichmann-Hill) MacLaren, N.M. "The Generation of Multiple Independent
    Sequences of Pseudorandom Numbers." Applied Statistics, 38, 351-359, 1989.

13 Sparse Matrices

The sparse matrix data type stores only the non-zero values of a 2-dimensional sparse matrix, which makes working with sparse matrices use less memory. Sparse matrix calculations may be faster or slower than those on an equivalent dense matrix, depending upon several factors.

Sparse matrix computations require extra complexity. This makes them slower than the same operation on an equivalent dense matrix. If the matrix is sufficiently sparse, the reduced element count can make up for this and provide better performance than dense operations. However, it is generally recommended to use dense matrices if your problem will fit into memory.

13.1 Defining Sparse Matrices

The sparse matrix data type is strongly typed in **GAUSS**, which means that a variable must be defined as a sparse matrix variable before it may be used as such. Once a

variable has been defined as a sparse matrix, it may not be used as another data type. Similarly, once a variable has been used as a matrix, array, or other non-sparse data type, it may not be redefined as a sparse matrix.

To define a global sparse matrix, you may use either the `declare` or the `let` command:

```
declare sparse matrix sm1;
let sparse matrix sm1;
```

or the following implicit `let` statement:

```
sparse matrix sm1;
```

`declare` may be used to define multiple sparse matrices in a single statement:

```
declare sparse matrix sm1, sm2, sm3;
```

To define a local sparse matrix inside of a procedure, use an implicit let statement:

```
sparse matrix lsm1;
```

As neither `let` nor `declare` support the initialization of a sparse matrix at this time, you must initialize a sparse matrix with an assignment after defining it.

13.2 Creating and Using Sparse Matrices

Several new functions have been added to allow you to create and manipulate sparse matrices. These functions are:

denseToSp	Converts a dense matrix to a sparse matrix.
denseToSpRE	Converts a dense matrix to a sparse matrix, using a relative epsilon.
packedToSp	Creates a sparse matrix from a packed matrix of non-zero values and row and column indices.
spBiconjGradSol	Solves the system of linear equations Ax=b

	using the biconjugate gradient method.
`spConjGradSol`	Solves the system of linear equations Ax=b for symmetric matrices using the conjugate gradient method.
`spCreate`	Creates a sparse matrix from vectors of non-zero values, row indices, and column indices.
`spDenseSubmat`	Returns a dense submatrix of sparse matrix.
`spDiagRvMat`	Inserts submatrices along the diagonal of a sparse matrix.
`spEigv`	Computes a specified number of eigenvalues and eigenvectors of a square, sparse matrix.
`spEye`	Creates a sparse identity matrix.
`spGetNZE`	Returns the non-zero values in a sparse matrix, as well as their corresponding row and column indices.
`spLDL`	Computes the LDL decomposition of a symmetric sparse matrix.
`spLU`	Computes the LU decomposition of a sparse matrix with partial pivoting.
`spNumNZE`	Returns the number of non-zero elements in a sparse matrix.
`spOnes`	Generates a sparse matrix containing only ones and zeros
`spSubmat`	Returns a sparse submatrix of sparse matrix.
`spToDense`	Converts a sparse matrix to a dense matrix.
`spTrTDense`	Multiplies a sparse matrix transposed by a dense matrix.
`spTScalar`	Multiplies a sparse matrix by a scalar.
`spZeros`	Creates a sparse matrix containing no non-zero

values.

See COMMAND REFERENCE for detailed information on each command.

13.3 Sparse Support in Matrix Functions and Operators

Support for the sparse matrix data type has also been added to many matrix functions and operators. The following is a complete list of the matrix functions and operators that currently support the new sparse matrix type:

`...`	`/=`	`.>=`	`minc`
`~`	`!=`	`<`	`print`
`|`	`./=`	`.<`	`rows`
`*`	`.!=`	`.<`	`scalerr`
`.*`	`==`	`<=`	`show`
`+`	`.==`	`.<=`	`type`
`-`	`>`	`abs`	
`/`	`.>`	`cols`	
`./`	`>=`	`maxc`	

Indexing is also supported for sparse matrices, using the same syntax as matrix indexing.

Note that `print`ing a sparse matrix results in a table of the non-zero values contained in the sparse matrix, followed by their corresponding row and column indices, respectively.

13.3.1 Return Types for Dyadic Operators

The types of the returns for the dyadic operators were decided on a case-by-case basis, using the following general principles:

1. The return type for dyadic operations on two dense arguments is always dense.

2. The return type for dyadic operations on two sparse arguments is always sparse unless the result is likely to be significantly less sparse than the sparse arguments.

3. The return type for dyadic operations on a dense argument and a sparse argument (regardless of order) is dense unless the return is likely to be at least as sparse as the sparse argument.

These general principles have led to the following decisions regarding return types (note that only the cases that are displayed in these tables have been implemented at this point):

Element-by-Element Numeric Operators

Element-by-Element Addition

Result	=	Left	Operator	Right
dense	=	sparse	+	dense
dense	=	dense	+	dense
sparse	=	sparse	+	sparse
dense	=	dense	+	sparse

Element-by-Element Subtraction

Result	=	Left	Operator	Right
dense	=	sparse	-	dense
dense	=	dense	-	dense
sparse	=	sparse	-	sparse
dense	=	dense	-	sparse

Element-by-Element Multiplication

Result	=	Left	Operator	Right
sparse	=	sparse	.*	dense
dense	=	dense	.*	dense
sparse	=	sparse	.*	sparse
sparse	=	dense	.*	sparse

Element-by-Element Division

Result	=	Left	Operator	Right
sparse	=	sparse	./	dense
dense	=	dense	./	dense
dense	=	sparse	./	sparse
dense	=	dense	./	sparse

Other Numeric Operators

Matrix Multiplication

Result	=	Left	Operator	Right
dense	=	sparse	*	dense
dense	=	dense	*	dense
sparse	=	sparse	*	sparse

Linear Solve

Result	=	Left	Operator	Right
dense	=	dense	/	dense
dense	=	dense	/	sparse

Note that at this time, the dense = dense **/** sparse case is defined only for real data.

When either of its arguments are sparse, the **/** operator uses a tolerance to determine the result, which may be read or set using the **sysstate** function, case 39. The default tolerance is 1e-14.

Relational Operators

Since the results of element-by-element 'dot' comparison operators depend largely on the kind of data inputted, there are both both dense-returning and sparse-returning versions of the dot comparison operators when one or both arguments is a sparse matrix. The regular dot comparison operators and their alphabetic counterparts always return dense matrices, and there is a new set of alphabetic dot comparison operators that all return sparse matrices:

Element-by-Element Dot Comparison Operators

Operation	Dense-Returning		Sparse-Returning
Equal to	.==	.eq	.speq
Not equal to	./=	.ne	.spne
Less than	.<	.lt	.splt
Less than or equal to	.<=	.le	.sple
Greater than	.>	.gt	.spgt
Greater than or equal to	.>=	.ge	.spge

Since the element-by-element 'non-dot' comparison operators (==, /=, <, <=, >, >=) and their alphabetic counterparts (eq, ne, lt, le, gt, ge) all return scalars, there are no sparse-returning versions of them.

Other Matrix Operators

Horizontal Concatenation

Result	=	Left	Operator	Right
dense	=	dense	~	dense
sparse	=	sparse	~	sparse

Vertical Concatenation

Result	=	Left	Operator	Right

dense	=	dense	|	dense
sparse	=	sparse	|	sparse

14 N-Dimensional Arrays

In **GAUSS**, internally, matrices and arrays are separate data types. Matrices, which are 2-dimensional objects, are stored in memory in row major order. Therefore, a 3x2 matrix is stored as follows:

```
[1,1]  [1,2]  [2,1]  [2,2]  [3,1]  [3,2]
```

The slowest moving dimension in memory is indexed on the left, and the fastest moving dimension is indexed on the right. This is true of N-dimensional arrays as well. A 4x3x2 array is stored in the following way:

```
[1,1,1]  [1,1,2]  [1,2,1]  [1,2,2]  [1,3,1]  [1,3,2]
[2,1,1]  [2,1,2]  [2,2,1]  [2,2,2]  [2,3,1]  [2,3,2]
[3,1,1]  [3,1,2]  [3,2,1]  [3,2,2]  [3,3,1]  [3,3,2]
[4,1,1]  [4,1,2]  [4,2,1]  [4,2,2]  [4,3,1]  [4,3,2]
```

A complex N-dimensional array is stored in memory in the same way. Like complex matrices, complex arrays are stored with the entire real part first, followed by the entire imaginary part.

Every N-dimensional array has a corresponding Nx1 vector of orders that contains the sizes of each dimension of the array. This is stored with the array and can be accessed with **getorders**. The first element of the vector of orders corresponds to the slowest

moving dimension, and the last element corresponds to the fastest moving dimension (refer to the Glossary of Terms at the end of the chapter for clear definitions of these terms). The vector of orders for a 6x5x4x3x2 array, which has 5 dimensions, is the following 5x1 vector:

```
6
5
4
3
2
```

Two terms that are important in working with N-dimensional arrays are "dimension index" and "dimension number." A dimension index specifies a dimension based on indexing the vector of orders. It is a scalar, 1-to-N, where 1 corresponds to the dimension indicated by the first element of the vector of orders of the array (the slowest moving dimension) and N corresponds to the dimension indicated by the last element of the vector of orders (the fastest moving dimension).

A dimension number specifies dimensions by numbering them in the same order that one would add dimensions to an array. In other words, the dimensions of an N-dimensional array are numbered such that the fastest moving dimension has a dimension number of 1, and the slowest moving dimension has a dimension number of N.

A 6x5x4x3x2 array has 5 dimensions, so the first element of the vector of orders (in this case, 6) refers to the size of dimension number 5. Since the index of this element in the vector of orders is 1, the dimension index of the corresponding dimension (dimension number 5) is also 1.

You will find references to both dimension index and dimension number in the documentation for the functions that manipulate arrays.

There are a number of functions that have been designed to manipulate arrays. These functions allow you to manipulate a subarray within the array by passing in a locator vector to index any subarray that comprises a contiguous block of memory within the larger block. A vector of indices of an N-dimensional array is a [1-to-N]x1 vector of base 1 indices into the array, where the first element corresponds to the first element in a vector of orders. An Nx1 vector of indices locates the scalar whose position is indicated by the indices. For a 4x3x2 array *x*, the 3x1 vector of indices:

```
3
2
1
```

indexes the [3,2,1] element of x. A 2x1 vector of indices for this 3-dimensional example, references the 1-dimensional array whose starting location is given by the indices.

Because the elements of the vector of indices are always in the same order (the first element of the vector of indices corresponds to the slowest moving dimension of the array, the second element to the second slowest moving dimension, and so on), each unique vector of indices locates a unique subarray.

In general, an [N-K]x1 vector of indices locates a K-dimensional subarray that begins at the position indicated by the indices. The sizes of the dimensions of the K-dimensional subarray correspond to the last K elements of the vector of orders of the N-dimensional array. For a 6x5x4x3x2 array y, the 2x1 vector of indices:

```
2
5
```

locates the 4x3x2 subarray in y that begins at [2,5,1,1,1] and ends at [2,5,4,3,2].

14.1 Bracketed Indexing

Brackets '[ ]' can be used to index N-dimensional arrays in virtually the same way that they are used to index matrices. Bracketed indexing is slower than the convenience array functions, such as `getarray` and `setarray`; however, it can be used to index non-contiguous elements. In order to index an N-dimensional array with brackets, there must be N indices located within the brackets, where the first index corresponds to the

slowest moving dimension of the array and the last index corresponds to the fastest mov-
ing dimension.

For a 2x3x4 array x, such that:

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

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

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

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

x[1,2,3] returns a 1x1x1 array containing the [1,2,3] element of *x*.

```
7
```

x[.,3,2] returns a 2x1x1 array containing:

```
10
22
```

x[2,.,1 4] returns a 1x3x2 array containing

```
13  16
17  20
21  24
```

x[.,2,1:3] returns a 2x1x3 array containing

```
 5  6  7
17 18 19
```

14.2 ExE Conformability

The following describes rules for ExE conformability of arrays for operators and functions with two or more arguments.

- Any N-dimensional array is conformable to a scalar.

- An array is conformable to a matrix only if the array has fewer than 3 dimensions, and the array and matrix follow the standard rules of ExE conformability.

- Two arrays are ExE conformable if they comply with one of the following requirements:
  - The two arrays have the same number of dimensions, and each dimension has the same size.
  - The two arrays have the same number of dimensions, and each of the N-2 slowest moving dimensions has the same size. In this case, the 2 fastest moving dimensions of the arrays must follow the ExE comformability rules that apply to matrices.
  - Both of the arrays have fewer than 3 dimensions, and they follow the ExE conformability rules that apply to matrices.

14.3 Glossary of Terms

dimensions	The number of dimensions of an object.
vector of orders	Nx1 vector of the sizes of the dimensions of an object, where N is the number of dimensions, and the first element corresponds to the slowest moving dimension.
vector of indices	[1-to-N]x1 vector of indices into an array, where the first element corresponds to the first element in a vector of orders.
dimension number	Scalar [1-to-N], where 1 corresponds to the fastest moving dimension and N to the slowest moving dimension.

dimension index	Scalar [1-to-N], where 1 corresponds to the first element of the vector of orders or vector of indices.
locator	[1-to-N]x1 vector of indices into an array used by array functions to locate a contiguous block of the array.

15 Working with Arrays

15.1 Initializing Arrays

The use of N-dimensional arrays in **GAUSS** is an additional tool for reducing development time and increasing execution speed of programs. There are multiple ways of handling N-dimensional arrays and using them to solve problems, and these ways sometimes have implications for a trade-off between speed of execution and development time. We will try to make this clear in this chapter.

The term "arrays" specifically refers to N-dimensional arrays and must not be confused with matrices. Matrices and arrays are distinct types even if in fact they contain

identical information. Functions for conversion from one to the other are described below.

There are five basic ways of creating an array depending on how the contents are specified:

areshape	Create array from specified matrix .
aconcat	Create array from matrices and arrays.
aeye	Create array of identity matrices.
arrayinit	Allocate array filled with specified scalar value.
arrayalloc	Allocate array with no specified contents.

15.1.1 areshape

areshape is a method for creating an array with specified contents. **arrayinit** creates an array filled with a selected scalar value: **areshape** will do the same, but with a matrix. For example, given a matrix, **areshape** will create an array containing multiple copies of that matrix:

```
x = reshape(seqa(1, 1, 4), 2, 2);
ord = 3 | 2 | 2;
a = areshape(x, ord);
print a;
  Plane [1,.,.]
```

```
1.0000 2.0000
3.0000 4.0000

Plane [2,.,.]
1.0000 2.0000
3.0000 4.0000

Plane [3,.,.]
1.0000 2.0000
3.0000 4.0000
```

Reading Data from the Disk into an Array

areshape is a fast way to re-dimension a matrix or array already in memory. For example, suppose we have a **GAUSS** data set containing panel data and that it's small enough to be read in all at once:

```
panel = areshape(loadd("panel"),5|100|10);
mn = amean(panel,2);  // 5x1x10 array of means of each panel
mm = moment(panel,0); // 5x10x10 array of moments of each
panel
/*
** vc is a 5x10x10 array of
** covariance matrices
*/

vc = mm / 100 - amult(atranspose(mn, 1|3|2), mn);
```

panel is a 5x100x10 array, and in this context is 5 panels of 100 cases measured on 10 variables.

Inserting Random Numbers into Arrays

A random array of any dimension or size can be quickly created using **areshape**. Thus, for a 10x10x5x3 array:

```
ord = { 10, 10, 5, 3 };
y = areshape(rndu(prodc(ord),1),ord);
```

Expanding a Matrix into an Array Vector of Matrices

For computing the log-likelihood of a variance components model of panel data, it is necessary to expand a TxT matrix into an NTxT array of these matrices. This is easily accomplished using **areshape**. For example:

```
m = { 1.0 0.3 0.2,
      0.3 1.0 0.1,
      0.2 0.1 1.0 };
r = areshape(m, 3|3|3);
print r;

  Plane [1,.,.]
  1.0000 0.3000 0.2000
  0.3000 1.0000 0.1000
  0.2000 0.1000 1.0000

  Plane [2,.,.]
  1.0000 0.3000 0.2000
  0.3000 1.0000 0.1000
  0.2000 0.1000 1.0000

  Plane [3,.,.]
  1.0000 0.3000 0.2000
  0.3000 1.0000 0.1000
  0.2000 0.1000 1.0000
```

15.1.2 aconcat

aconcat creates arrays from conformable sets of matrices or arrays. With this function, contents are completely specified by the user. This example tries three concatenations, one along each dimension:

```
rndseed 345678;
x1 = rndn(2,2);
x2 = arrayinit(2|2,1);

/*
** along the first dimension or rows
*/

a = aconcat(x1,x2,1);
print a;

   -0.4300 -0.2878 1.0000 1.0000
   -0.1327 -0.0573 1.0000 1.0000

/*
** along the second dimension or columns
*/

a = aconcat(x1,x2,2);
print a;

   -0.4300 -0.2878
   -0.1327 -0.0573
    1.0000  1.0000
    1.0000  1.0000

/*
** along the third dimension
*/

a = aconcat(x1,x2,3);
print a;

   Plane [1,.,.]
   -0.4300 -0.2878
   -0.1327 -0.0573
```

```
Plane [2,.,.]
1.0000 1.0000
1.0000 1.0000
```

15.1.3 aeye

aeye creates an array in which the principal diagonal of the two trailing dimensions is set to one. For example:

```
ord = 2 | 3 | 3;
a = aeye(ord);
print a;

    Plane [1,.,.]
    1.00000 0.00000 0.00000
    0.00000 1.00000 0.00000
    0.00000 0.00000 1.00000

    Plane [2,.,.]
    1.00000 0.00000 0.00000
    0.00000 1.00000 0.00000
    0.00000 0.00000 1.00000
```

15.1.4 arrayinit

arrayinit creates an array with all elements set to a specified value. For example:

```
ord = 3 | 2 | 3;
a = arrayinit(ord,1);
print a;

    Plane [1,.,.]
    1.0000 1.0000 1.0000
```

```
1.0000 1.0000 1.0000

Plane [2,.,.]
1.0000 1.0000 1.0000
1.0000 1.0000 1.0000

Plane [3,.,.]
1.0000 1.0000 1.0000
1.0000 1.0000 1.0000
```

15.1.5 arrayalloc

arrayalloc creates an array with specified number and size of dimensions without setting elements to any values. This requires a vector specifying the order of the array. The length of the vector determines the number of dimensions, and each element determines the size of the corresponding dimensions. The array will then have to be filled using any of several methods described later in this chapter.

For example, to allocate a 2x2x3 array:

```
rndseed 345678;
ord = 3 | 2 | 2;
a = arrayalloc(ord, 0);


for i(1,ord[1],1);
   a[i,.,.] = rndn(2, 3);
endfor;

print a;

  Plane [1,.,.]
 -0.4300 -0.2878 -0.1327
 -0.0573 -1.2900 0.2467
```

<div style="writing-mode: vertical-rl">Working with Arrays</div>

```
 Plane [2,.,.]
-1.4249 -0.0796 1.2693
-0.7530 -1.7906 -0.6103

 Plane [3,.,.]
 1.2586 -0.4773 0.7044
-1.2544 0.5002 0.3559
```

The second argument in the call to **arrayalloc** specifies whether the created array is real or complex. **arrayinit** creates only real arrays.

15.2 Assigning to Arrays

There are three methods used for assignment to an array:

index operator	The same method as matrices, generalized to arrays.
putarray	Put a subarray into an N-dimensional array and returns the result.
setarray	Set a subarray of an N-dimensional array in place.

And there are several ways to extract parts of arrays:

index operator	The same method as matrices, generalized to arrays.
getarray	Get a subarray from an array.
getmatrix	Get a matrix from an array.
getmatrix4D	Get a matrix from a 4-dimensional array.
getscalar4D	Get a scalar from a 4-dimensional array.

The index operator is the slowest way to extract parts of arrays. The specialized functions are the fastest when the circumstances are appropriate for their use.

15.2.1 index operator

The index operator will put a **subarray** into an array in a manner analogous to the use of index operators on matrices:

```
a = arrayinit(3|2|2, 0);
b = arrayinit(3|1|2, 1);
a[.,2,.] = b;
print a;

   Plane [1,.,.]
   0.0000 0.0000
   1.0000 1.0000

   Plane [2,.,.]
   0.0000 0.0000
   1.0000 1.0000

   Plane [3,.,.]
   0.0000 0.0000
   1.0000 1.0000
```

As this example illustrates, the assignment doesn't have to be contiguous. **putMatrix** and **setMatrix** require a contiguous assignment, but for that reason they are faster.

The right hand side of the assignment can also be a matrix:

```
a[1,.,.] = rndn(2, 2);
print a;

   Plane [1,.,.]
 -1.7906502 -0.61038103
  1.2586160 -0.47736360

   Plane [2,.,.]
  0.00000 0.00000
  1.00000 1.00000

   Plane [3,.,.]
  0.00000 0.00000
  1.00000 1.00000
```

The index operator will extract an array from a subarray in a manner analogous to the use of index operators on matrices:

```
a = areshape(seqa(1,1,12), 3|2|2);
b = a[.,1,.];
print a;

   Plane [1,.,.]
   1.0000 2.0000
   3.0000 4.0000

   Plane [2,.,.]
   5.0000 6.0000
   7.0000 8.0000

   Plane [3,.,.]
   9.0000 10.000
   11.000 12.000

print b;
```

```
Plane [1,.,.]
1.0000 2.0000

Plane [2,.,.]
5.0000 6.0000

Plane [3,.,.]
9.0000 10.000
```

It is important to note that the result is always an array even if it's a scalar value:

```
c = a[1,1,1];
print c;

    Plane [1,.,.]
    1.0000
```

If you require a matrix result, and if the result has one or two dimensions, use **arrayto-mat** to convert to a matrix, or use **getmatrix**, or **getmatrix4D**. Or, if the result is a scalar, use **getscalar3D** or **getscalar4D**.

15.2.2 getarray

getarray is an additional method for extracting arrays:

```
a = areshape(seqa(1,1,12), 3|2|2);
b = getarray(a, 2|1);
print a;

    Plane [1,.,.]
    1.0000 2.0000
    3.0000 4.0000

    Plane [2,.,.]
    5.0000 6.0000
```

```
    7.0000 8.0000

    Plane [3,.,.]
    9.0000 10.000
    11.000 12.000

print b;

    5.0000 6.0000
```

getarray can only extract a contiguous part of an array. To get non-contiguous parts you must use the index operator.

15.2.3 getmatrix

If the result is one or two dimensions, **getmatrix** returns a portion of an array converted to a matrix. **getmatrix** is about 20 percent faster than the index operator:

```
a = areshape(seqa(1,1,12), 3|2|2);
b = getmatrix(a, 2);
print b;

    5.0000 6.0000
    7.0000 8.0000
```

15.2.4 getmatrix4D

This is a specialized version of **getmatrix** for 4-dimensional arrays. It behaves just like **getmatrix** but is dramatically faster for that type of array. The following illustrates the difference in timing:

```
a = arrayinit(100|100|10|10, 1);
t0 = date;

for i(1, 100, 1);
```

```
        for j(1, 100, 1);
            b = a[i,j,.,.];
        endfor;
    endfor;

    t1 = date;
    e1 = ethsec(t0, t1);
    print e1;
    print;
    t2 = date;

    for i(1,100,1);
        for j(1,100,1);
            b = getmatrix4d(a, i, j);
        endfor;
    endfor;

    t3 = date;
    e2 = ethsec(t2, t3);
    print e2;
    print;
    print ftostrC(100*((e1-e2)/e1), "percent difference -
    %6.2lf%%");

     13.000000
     5.0000000
    percent difference - 61.54%
```

15.2.5 getscalar3D, getscalar4D

These are specialized versions of **getmatrix** for retrieving scalar elements of 3-dimensional and 4-dimensional arrays, respectively. They behave just like **getmatrix**, with scalar results, but are much faster. For example:

```
a = arrayinit(100|10|10, 1);
t0 = date;

for i(1,100,1);
   for j(1,10,1);
     for k(1,10,1);
        b = a[i,j,k];
     endfor;
   endfor;
endfor;

t1 = date;
e1 = ethsec(t0, t1);
print e1;
print;
t2=date;

for i(1,100,1);
   for j(1,10,1);
     for k(1,10,1);
        b = getscalar3d(a, i, j, k);
     endfor;
   endfor;
endfor;

t3 = date;
e2 = ethsec(t2, t3);
print e2;
print;
print ftostrC(100*((e1-e2)/e1), "percent difference -
%6.2lf%%");

7.0000000
2.0000000
percent difference - 71.43%
```

15.2.6 putarray

putarray enters a subarray, matrix, or scalar into an N-dimensional array and returns the result in an array. This function is much faster than the index operator, but it requires the part of the array being assigned to be contiguous:

```
a = arrayinit(3|2|2, 3);
b = putarray(a, 2, eye(2));
print b;

   Plane [1,.,.]
   3.0000 3.0000
   3.0000 3.0000

   Plane [2,.,.]
   1.0000 0.00000
   0.00000 1.0000

   Plane [3,.,.]
   3.0000 3.0000
   3.0000 3.0000
```

15.2.7 setarray

setarray enters a subarray, matrix, or scalar into an N-dimensional array in place:

```
a = arrayinit(3|2|2,3);
setarray a,2, eye(2);
print b;

   Plane [1,.,.]
   3.0000 3.0000
   3.0000 3.0000

   Plane [2,.,.]
```

```
1.0000 0.0000
0.0000 1.0000

Plane [3,.,.]
3.0000 3.0000
3.0000 3.0000
```

15.3 Looping with Arrays

When working with arrays, `for` loops and `do` loops may be used in the usual way. In the following, let *Y* be an Nx1xL array of L time series, *X* an Nx1xK array of K independent variables, *b* a KxL matrix of regression coefficients, *phi* a PxLxL array of garch coefficients, *theta* a QxLxL array of arch coefficients, and *omega* a LxL symmetric matrix of constants. The log-likelihood for a multivariate garch BEKK model can be computed using the index operator:

```
yord = getorders(Y);
xord = getorders(X);
gord = getorders(phi);
aord = getorders(theta);
N = yord[1]; // No. of observations
L = yord[3]; // No. of time series
K = xord[3]; // No. of independent variables // in mean equa-
tion
P = gord[1]; // order of garch parameters
Q = aord[1]; // order of arch parameters

r = maxc(P|Q);
E = Y - amult(X, areshape(B, N|K|L));
sigma = areshape(omega,N|L|L);

for i(r+1, N, 1);
   for j(1,Q,1);
      W = amult(theta[j,.,.],
      atranspose(E[i-j,.,.], 1|3|2));
```

```
        sigma[i,.,.] = sigma[i,.,.] + amult(W, atranspose
(W,1|3|2));
    endfor;
    for j(1, P, 1);
        sigma[i,.,.] = sigma[i,.,.] + amult(amult(phi[j,.,.],
sigma[i-j,.,.]),phi[j,.,.]);
    endfor;
endfor;

sigmai = invpd(sigma);
lndet = ln(det(sigma));
lnl = -0.5*( L*(N-R)*asum(ln(det(sigmai)),1) + asum(amult
(amult(E, sigmai), atranspose(E,1|3|2)), 3);
```

Instead of index operators, the above computation can be done using **getarray** and
setarray:

```
yord = getorders(Y);
xord = getorders(X);
gord = getorders(phi);
aord = getorders(theta);
N = yord[1]; // No. of observations
L = yord[3]; // No. of time series
K = xord[3]; // No. of independent variables in mean equation
P = gord[1]; // order of garch parameters
Q = aord[1]; // order of arch parameters

r = maxc(P|Q);
E = Y - amult(X, areshape(B,N|K|L));
sigma = areshape(omega, N|L|L);

for i(r+1, N, 1);
   for j(1,Q,1);
      W = amult(getarray(theta, j),
      atranspose(getarray(E, i-j), 2|1));
      setarray sigma,i, getarray(sigma, i)+
```

```
        amult(W, atranspose(W, 2|1));
    endfor;
    for j(1, P, 1);
        setarray sigma,i, getarray(sigma,i)+
        areshape(amult(amult(getarray(phi, j),
        getarray(sigma, i-j)), getarray(phi, j)), 3|3);
    endfor;
endfor;

sigmai = invpd(sigma);
lndet = ln(det(sigma));
lnl = -0.5*( L*(N-R)*asum(ln(det(sigmai)),1)+
 asum(amult(amult(E,sigmai), atranspose(E,1|3|2)), 3)
```

Putting the two code fragments above into loops that called them a hundred times and measuring the time, produced the following results:

index operator: 2.604 seconds

getarray, **setarray**: 1.092 seconds

Thus, the **getarray** and **setarray** methods are more than twice as fast.

15.3.1 loopnextindex

Several keyword functions are available in **GAUSS** for looping with arrays. The problem in the previous section, for example, can be written using these functions rather than with for loops:

```
sigind = r + 1;
sigloop:

sig0ind = sigind[1];
```

```
thetaind = 1;
thetaloop:

sig0ind = sig0ind - 1;
W = amult(getarray(theta,thetaind),
atranspose(getarray(E, sig0ind), 2|1));
setarray sigma,sigind, getarray(sigma, sigind)+
amult(W, atranspose(W, 2|1));

loopnextindex thetaloop,thetaind,aord;
sig0ind = sigind;
phiind = 1;
philoop:

sig0ind[1] = sig0ind[1] - 1;
setarray sigma,sigind, getarray(sigma, sigind)+
areshape(amult(amult(getarray(phi, phiind),
getArray(sigma, sig0ind)),
getArray(phi, phiind)), 3|3);

loopnextindex philoop,phiind,gord;
loopnextindex sigloop,sigind,sigord;
```

The `loopnextindex` function in this example isn't faster than the `for` loop used in the previous section primarily because the code is looping only through the first dimension in each loop. The advantages of `loopnextindex`, **previousindex**, **nextindex**, and **walkindex** are when the code is looping through the higher dimensions of a highly dimensioned array. In this case, looping through an array can be very complicated and difficult to manage using `for` loops. `loopnextindex` can be faster and more useful.

The next example compares two ways of extracting a subarray from a 5-dimensional array:

```
ord = 3|3|3|3|3;
a = areshape(seqa(1, 1, prodc(ord)) ,ord);
```

```
b = eye(3);

for i(1, 3, 1);
   for j(1, 3, 1);
      for k(1, 3, 1);
         setarray a,i|j|k,b;
      endfor;
   endfor;
endfor;

ind = { 1,1,1 };
loopi:

setarray a,ind,b;
loopnextindex loopi,ind,ord;
```

Calling each loop 10,000 times and measuring the time each takes, we get

for loop: 1.171 seconds

loopnextindex: .321 seconds

In other words, loopnextindex is about four times faster, a very significant difference.

15.4 Miscellaneous Array Functions

This section discusses miscellaneous array functions.

15.4.1 atranspose

This function changes the order of the dimensions. For example:

```
a = areshape(seqa(1, 1, 12), 2|3|2);
print a;

   Plane [1,.,.]
   1.0000 2.0000
   3.0000 4.0000
   5.0000 6.0000

   Plane [2,.,.]
   7.0000 8.0000
   9.0000 10.000
   11.000 12.000

/*
** swap 2nd and 3rd dimension
*/
print atranspose(a,1|3|2);

   Plane [1,.,.]
   1.0000 3.0000 5.0000
   2.0000 4.0000 6.0000

   Plane [2,.,.]
   7.0000 9.0000 11.000
   8.0000 10.000 12.000

/*
```

```
** swap 1st and 3rd dimension
*/
print atranspose(a,3|2|1);

   Plane [1,.,.]
   1.0000 7.0000
   3.0000 9.0000
   5.0000 11.000

   Plane [2,.,.]
   2.0000 8.0000
   4.0000 10.000
   6.0000 12.000

/*
** move 3rd into the front
*/
print atranspose(a,3|1|2);

   Plane [1,.,.]
   1.0000 3.0000 5.0000
   7.0000 9.0000 11.000

   Plane [2,.,.]
   2.0000 4.0000 6.0000
   8.0000 10.000 12.000
```

15.4.2 amult

This function performs a matrix multiplication on the last two trailing dimensions of an array. The leading dimensions must be strictly conformable, and the last two trailing dimensions must be conformable in the matrix product sense. For example:

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

```
c = amult(a, b);
print a;

    Plane [1,.,.]
    1.0000 2.0000
    3.0000 4.0000
    5.0000 6.0000

    Plane [2,.,.]
    7.0000 8.0000
    9.0000 10.000
    11.000 12.000

    print b;

    Plane [1,.,.]
    1.0000 2.0000 3.0000 4.0000
    5.0000 6.0000 7.0000 8.0000

    Plane [2,.,.]
    9.0000 10.000 11.000 12.000
    13.000 14.000 15.000 16.000

    print c;

    Plane [1,.,.]
    11.000 14.000 17.000 20.000
    23.000 30.000 37.000 44.000
    35.000 46.000 57.000 68.000

    Plane [2,.,.]
    167.00 182.00 197.00 212.00
    211.00 230.00 249.00 268.00
    255.00 278.00 301.00 324.00
```

Suppose we have a matrix of data sets, a 2x2 matrix of 100x5 data sets that we've stored in a 2x2x100x5 array called x. The moment matrices of these data sets can easily and quickly be computed using **atranspose** and **amult**:

```
vc = amult(atranspose(x, 1|2|4|3), x);
```

15.4.3 amean, amin, amax

These functions compute the means, minimums, and maximums, respectively, across a dimension of an array. The size of the selected dimension of the resulting array is shrunk to one and contains the means, minimums, or maximums depending on the function called. For example:

```
a = areshape(seqa(1,1,12), 2|3|2);
print a;

   Plane [1,.,.]
   1.0000 2.0000
   3.0000 4.0000
   5.0000 6.0000

   Plane [2,.,.]
   7.0000 8.0000
   9.0000 10.000
   11.000 12.000

/*
** compute means along third dimension
*/
print amean(a, 3);

   Plane [1,.,.]
   4.0000 5.0000
   6.0000 7.0000
   8.0000 9.0000
```

```
/*
** print means along the second dimension, i.e.,
** down the columns
*/
print amean(a, 2);

    Plane [1,.,.]
    3.0000 4.0000

    Plane [2,.,.]
    9.0000 10.000

/*
** print the minimums down the columns
*/
print amin(a, 2);

    Plane [1,.,.]
    1.0000 2.0000

    Plane [2,.,.]
    7.0000 8.0000

/*
** print the maximums along the third dimension
*/
print amax(a, 3);

    Plane [1,.,.]
    7.0000 8.0000
    9.0000 10.000
    11.000 12.000
```

15.4.4 getdims

This function returns the number of dimensions of an array:

```
a = arrayinit(4|4|5|2, 0);
print getdims(a);

    4.00
```

15.4.5 getorders

This function returns the sizes of each dimension of an array. The length of the vector returned by **getorders** is the dimension of the array:

```
a = arrayinit(4|4|5|2,0);
print getorders(a);

    4.00
    4.00
    5.00
    2.00
```

15.4.6 arraytomat

This function converts an array with two or fewer dimensions to a matrix:

```
a = arrayinit(2|2, 0);
b = arraytomat(a);
type(a);

    21.000

type(b);
```

```
   6.0000
```

15.4.7 mattoarray

This function converts a matrix to an array:

```
b = rndn(2,2);
a = mattoarray(b);
type(b);

    6.0000

type(a);

 21.000
```

15.5 Using Arrays with GAUSS functions

Many of the **GAUSS** functions have been re-designed to work with arrays. There are two general approaches to this implementation. There are exceptions, however, and you are urged to refer to the documention if you are not sure how a particular **GAUSS** function handles array input.

In the first approach, the function returns an element-by-element result that is strictly conformable to the input. For example, **cdfNc** returns an array of identical size and shape to the input array:

```
a = areshape(seqa(-2, .5, 12), 2|3|2);
b = cdfnc(a);
print b;

    Plane [1,.,.]
    0.9772 0.9331
```

Working with Arrays

```
0.8413 0.6914
0.5000 0.3085

Plane [2,.,.]
0.1586 0.0668
0.0227 0.0062
0.0013 0.0002
```

In the second approach, which applies generally to **GAUSS** matrix functions, the function operates on the matrix defined by the last two trailing dimensions of the array. Thus, given a 5x10x3 array, **moment** returns a 5x3x3 array of five moment matrices computed from the five 10x3 matrices in the input array.

Only the last two trailing dimensions matter; i.e., given a 2x3x4x5x10x6 array, **moment** returns a 2x3x4x5x6x6 array of moment matrices.

For example, in the following the result is a 2x3 array of 3x1 vectors of singular values of a 2x3 array of 6x3 matrices:

```
a = areshape(seqa(1,1,108), 2|3|6|3);
b=svds(a);
print b;

 Plane [1,1,.,.]
 45.894532
 1.6407053
 1.2063156e-015

 Plane [1,2,.,.]
 118.72909
 0.63421188
 5.8652600e-015

 Plane [1,3,.,.]
 194.29063
 0.38756064
```

```
1.7162751e-014

Plane [2,1,.,.]
270.30524
0.27857175
1.9012118e-014

Plane [2,2,.,.]
346.47504
0.21732995
1.4501098e-014

Plane [2,3,.,.]
422.71618
0.17813229
1.6612287e-014
```

It might be tempting to conclude from this example that, in general, a **GAUSS** function's behavior on the last two trailing dimensions of an array is strictly analogous to the **GAUSS** function's behavior on a matrix. This may be true with some of the functions, but not all. For example, the **GAUSSmeanc** function returns a column result for matrix input. However, the behavior for the **GAUSSamean** function is not analogous. This function takes a second argument that specifies on which dimension the mean is to be taken. That dimension is then collapsed to a size of 1. Thus:

```
a = areshape(seqa(1, 1, 24), 2|3|4);
print a;

   Plane [1,.,.]
   1.000 2.000 3.000 4.000
   5.000 6.000 7.000 8.000
   9.000 10.000 11.000 12.000

   Plane [2,.,.]
   13.000 14.000 15.000 16.000
```

```
   17.000 18.000 19.000 20.000
   21.000 22.000 23.000 24.000

/*
** means computed across rows
*/

b = amean(a,1);
print b;

   Plane [1,.,.]
   2.500
   6.500
   10.500

   Plane [2,.,.]
   14.500
   18.500
   22.500

/*
** means computed down columns
*/
c = amean(a,2);
print c;

   Plane [1,.,.]
   5.000 6.000 7.000 8.000
   Plane [2,.,.]
   17.000 18.000 19.000 20.000

/*
** means computed along 3rd dimension
*/

d = amean(a,3);
print d;
```

```
Plane [1,.,.]
7.000 8.000 9.000 10.000
11.000 12.000 13.000 14.000
15.000 16.000 17.000 18.000
```

15.6 A Panel Data Model

Suppose we have N cases observed at T times. Let y_{it} be an observation on a dependent variable for the ith case at time t, X_{it} an observation of k independent variables for the ith case at time t, B, a Kx1 vector of coefficients. Then

```
Yit = XitB + μi + εit
```

is a variance components model where μ_i is a random error term uncorrelated with ε_{it}, but which is correlated within cases. This implies an NTxNT residual moment matrix that is block diagonal with N TxT moment matrices with the following form:

$$
\begin{bmatrix}
\sigma_\mu^2 + \sigma_\epsilon^2 & \sigma^2 & \cdots & \sigma_\mu^2 \\
\sigma_\mu^2 & \sigma_\mu^2 + \sigma_\epsilon^2 & \cdots & \sigma_\mu^2 \\
\vdots & \vdots & \vdots & \vdots \\
\sigma_\mu^2 & \sigma_\mu^2 & \cdots & \sigma_\mu^2 + \sigma_\epsilon^2
\end{bmatrix}
$$

The log-likelihood for this model is

```
lnL = -0.5(NT(ln(2π)) - ln|Ω| + (Y - XB)'Ω⁻¹(Y - XB))
```

where Ω is the block-diagonal moment matrix of the residuals.

Computing the Log-likelihood

Using **GAUSS** arrays, we can compute the log-likelihood of this model without resorting to do loops. Let Y be a 100x3x1 array of observations on the dependent variable, and X a 100x3x5 array of observations on the independent variables. Further let b be a 5x1 vector of coefficients, and *sigu* and *sige* be the residual variances of μ and ε respectively. Then, in explicit steps we compute

```
N = 100;
T = 3;
K = 5;
sigma = sigu * ones(T, T) + sige * eye(T); // TxT sigma
sigmai = invpd(sigma); // sigma inverse
lndet = N*ln(detl);
E = Y - amult(X, areshape(B, N|K|1)); // residuals
Omegai = areshape(sigmai, N|T|T);   // diagonal blocks //
stacked in a vector array
R1 = amult(atranspose(E, 1|3|2), Omegai); // E'Omegai
R2 = amult(R1, E);          // R1*E
lnL = -0.5*(N*T*ln(2*pi) - lndet + asum(R2, 3)); // log-likel-
hood
```

All of this can be made more efficient by nesting statements, which eliminates copying of temporary intervening arrays to local arrays. It is also useful to add a check for the positive definiteness of *sigma*:

```
N = 100;
T = 3;
K = 5;
const = -0.5*N*T*ln(2*pi);
oldt = trapchk(1);
trap 1,1;
sigmai = invpd(sigu*ones(T, T)+sige*eye(T));
trap oldt,1;

if not scalmiss(sigmai);
   E = Y - amult(X, areshape(B,N|K|1));
```

```
   lnl = const + 0.5*N*ln(detl)-
   0.5*asum(amult(amult(atranspose(E,1|3|2),
   areshape(sigmai,N|T|T)),E),3);
else;
   lnl = error(0);
endif;
```

15.7 Appendix

This is an incomplete list of special functions for working with arrays. Many **GAUSS** functions have been modified to handle arrays and are not listed here. For example, **cdfNc** computes the complement of the Normal cdf for each element of an array just as it would for a matrix. See the documentation for these **GAUSS** functions for information about their behavior with arrays.

aconcat	Concatenate conformable matrices and arrays in a user-specified dimension.
aeye	Create an array of identity matrices.
amax	Compute the maximum elements across a dimension of an array.
amean	Compute the mean along one dimension of an array.
amin	Compute the minimum elements across a dimension of an array.
amult	Perform a matrix multiplication on the last two trailing dimensions of an array.
areshape	Reshape a scalar, matrix, or array into an array of user-specified size.
arrayalloc	Create an N-dimensional array with unspecified contents.
arrayinit	Create an N-dimensional array with a specified fill value.

arraytomat	Change an array to type matrix.
asum	Compute the sum across one dimension of an array.
atranspose	Transpose an N-dimensional array.
getarray	Get a contiguous subarray from an N-dimensional array.
getdims	Get the number of dimensions in an array.
getmatrix	Get a contiguous matrix from an N-dimensional array.
getmatrix4D	Get a contiguous matrix from a 4-dimensional array.
getorders	Get the vector of orders corresponding to an array.
getscalar3D	Get a scalar from a 3-dimensional array.
getscalar4D	Get a scalar form a 4-dimensional array.
loopnextindex	Increment an index vector to the next logical index and jump to the specified label if the index did not wrap to the beginning.
mattoarray	Change a matrix to a type array.
nextindex	Return the index of the next element or subarray in an array.
previousindex	Return the index of the previous element or subarray in an array.
putarray	Put a contiguous subarray into an N-dimensional array and return the resulting array.
setarray	Set a contiguous subarray of an N-dimensional array.
walkindex	Walk the index of an array forward or backward through a specified dimension.

16 Structures

Structures

16.1 Basic Structures

Structures

16.1.1 Structure Definition

The syntax for a structure definition is

```
struct A { /* list of members */ };
```

The list of members can include scalars, arrays, matrices, strings, and string arrays, as well as other structures. As a type, scalars are unique to structures and don't otherwise exist.

For example, the following defines a structure containing the possible contents:

```
struct generic_example {
    scalar x;
    matrix y;
    string s1;
    string array s2;
    struct other_example t;
};
```

Defining a structure in a program

A useful convention is to save the structure definition into a file with a `.sdf` extension. After the structure definition has been created and saved in a file, the next step is to make the structure definition available for your program. This may be accomplished by either executing the structure definition file or by adding the structure definition to a library.

Executing a structure definition file

You can execute a structure definition just like any other GAUSS program file. However, any time that the **new** command is entered, the structure definition will be cleared from your workspace. To ensure that the structure definition is always available for your program, you can add an `#include` statement like this:

```
#include  filename.sdf
```

to the top of your program file. This will ensure that the structure definition is always available for your program.

Adding a structure definition to a library

Starting in GAUSS version 13.1, you can also add structure definition files to a GAUSS library. You can add structure definitions to any GAUSS library with the `lib` command or using the Library Tool in the user interface. If a structure definition is in a library file, you can make all procedures and structure definitions available with one statement. For example:

```
library  mylibrary;
```

instead of:

```
library  mylibrary;
#include  mystruct1.sdf
#include  mystruct2.sdf
```

Also if a structure is defined in an active library, autocomplete will be available for the structure members.

The next section describes how to create an instance of a structure.

16.1.2 Declaring an Instance

To use a structure, it is necessary to declare an instance. The syntax for this is

```
struct structure_type structure_name;
```

For example:

```
#include  example.sdf
struct generic_example p0;
```

16.1.3 Initializing an Instance

Members of structures are referenced using a "dot" syntax:

```
p0.x = rndn(20,3);
```

The same syntax applies when referred to on the right-hand side:

```
mn = meanc(p0.x);
```

Initialization of Global Structures

Global structures are initialized at compile time. Each member of the structure is initialized according to the following schedule:

scalar	0, a scalar zero
matrix	{}, an empty matrix with zero rows and zero columns
array	0, a 1-dimensional array set to zero
string	"", a null string
string array	"", a 1x1 string array set to null
sparse matrix	{}, an empty sparse matrix with zero rows and zero columns

If a global already exists in memory, it will not be reinitialized. It may be the case in your program that when it is rerun, the global variables may need to be reset to default values. That is, your program may depend on certain members of a structure being set to default values that are set to some other value later in the program. When you rerun this program, you will want to reinitialize the global structure. To do this, make an assignment to at least one of the members. This can be made convenient by writing a procedure that declares a structure and initializes one of its members to a default value, and then returns it. For example:

```
//ds.src
#include  ds.sdf

proc dsCreate;
   struct DS d0;
```

```
        d0.dataMatrix = 0;
        retp(d0);
endp;
```

Calling this function after declaring an instance of the structure will ensure initialization to default values each time your program is run:

```
    struct DS d0;
    d0 = dsCreate;
```

Initializing Local Structures

Local structures, which are structures defined inside procedures, are initialized at the first assignment. The procedure may have been written in such a way that a subset of structures are used an any one call, and in that case time is saved by not initializing the unused structures. They will be initialized to default values only when the first assignment is made to one of its members.

16.1.4 Arrays of Structures

To create a matrix of instances, use the **reshape** command:

```
    #include  ds.sdf
    struct DS p0;
    p0 = reshape(dsCreate,5,1);
```

This creates a 5x1 vector of instances of **DS** structures, with all of the members initialized to default values.

When the instance members have been set to some other values, **reshape** will produce multiple copies of that instance set to those values.

Matrices or vectors of instances can also be created by concatenation:

```
    #include  trade.sdf
    struct option p0,p1,p2;
```

Structures

```
p0 = optionCreate;
p1 = optionCreate;
p2 = p1 | p0;
```

16.1.5 Structure Indexing

Structure indexing may be used to reference a particular element in a structure array. The syntax follows that of matrix indexing. For example, given the following structure definition:

```
struct example1 {
   matrix x;
   matrix y;
   string str;
};
```

you could create an array of *example1* structures and index it as follows:

```
struct example1 e1a;
struct example1 e1b;

e1a = e1a | e1b;
e1a[2,1].y = rndn(25,10);
```

In this example, *e1a* and *e1b* are concatenated to create a 2x1 array of *example1* structures that is assigned back to *e1a*. Then the *y* member of the [2,1] element of *e1a* is set to a random matrix.

Indexing of structure arrays can occur on multiple levels. For example, let's define the following structures:

```
struct example3 {
   matrix w;
   string array sa;
};
```

```
struct example2 {
   matrix z;
   struct example3 e3;
};
```

and let's redefine *example1* to include an instance of an *example2* structure:

```
struct example1 {
   matrix x;
   matrix y;
   string str;
   struct example2 e2;
};
```
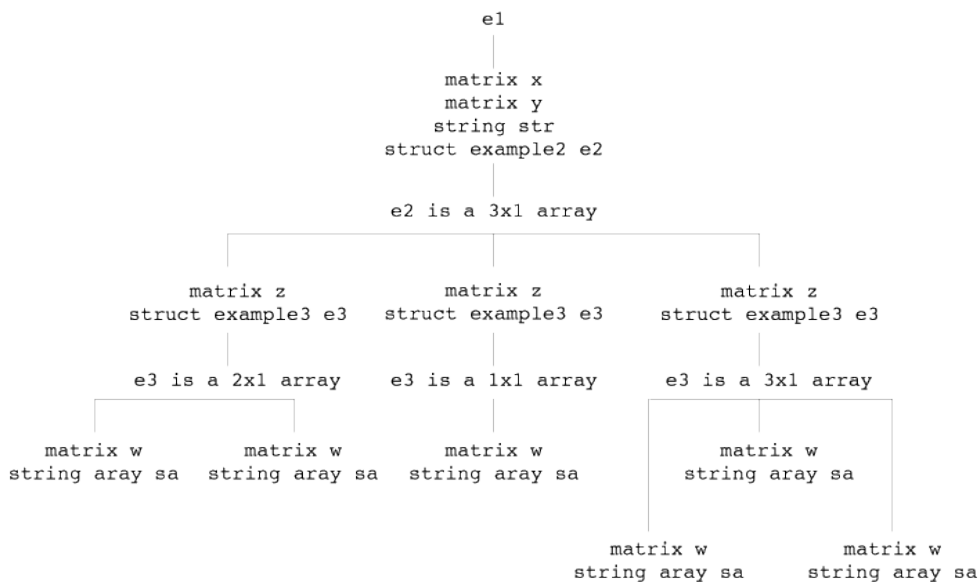
Figure 16.1: Structure tree for *e1*

Let's assume that we have an *example1* structure *e1* like the one displayed in the figure above. We could then index the structure as follows:

```
r = e1.e2[3,1].e3[2,1].w
```

You can also use indexing to reference the structure itself, rather than a member of that structure:

```
struct example3 e3tmp;
e3tmp = e1.e2[3,1].e3[2,1];
```

Or you can use indexing to reference a subarray of structures:

```
e3tmp = e1.e2[3,1].e3[.,1];
```

In this case, *e3tmp* would be an array of 3x1 *example3* structures, since the [3,1] member of *e1.e2* contains a 3x1 array of *example3* structures.

It is important to remember, however, that when indexing a structure array on multiple levels, only the final index may resolve to an array of structures. For example:

```
e3tmp = e1.e2[.,1].e3[2,1];
```

would be invalid, since *e1.e2* [.,1] resolves to a 3x1 array of *example2* structures.

16.1.6 Saving an Instance to the Disk

Instances and vectors or matrices of instances of structures can be saved in a file on the disk, and later loaded from the file onto the disk. The syntax for saving an instance to the disk is

```
ret = savestruct(instance, filename);
```

The file on the disk will have an .fsr extension.

For example:

```
#include  ds.sdf
struct DS p0;
p0 = reshape(dsCreate,2,1);
retc = saveStruct(p2, "p2");
```

This saves the vector of instances in a file called `p2.fsr`. The variable *retc* will be zero if the save was successful; otherwise, nonzero.

16.1.7 Loading an Instance from the Disk

The syntax for loading a file containing an instance or matrix of instances is

```
{ instance, retc } = loadstruct(file_name, structure_name);
```

For example:

```
#include  ds.sdf;
struct  DS p3;
{ p3, retc } = loadstruct("p2", "ds");
```

16.1.8 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;
```

16.2 Structure Pointers

A structure pointer is a separate data type that contains the address of a structure and is used to reference that structure.

16.2.1 Creating and Assigning Structure Pointers

Given the following structure type definition:

```
struct example_struct {
   matrix x;
```

```
    matrix y;
};
```

a pointer to an *example_struct* structure can be created with the following syntax:

```
    struct example_struct *esp;
```

However, at this point, *esp* is not yet pointing at anything. It has only been defined to be the kind of pointer that points at *example_struct* structures. To set it to point at a particular structure instance, we must first create the structure instance:

```
    struct example_struct es;
```

and then we can set *esp* to point at *es* by setting *esp* to the address of *es*:

```
    esp = &es;
```

The following code:

```
    struct example_struct es2;
    es2 = *esp;
```

copies the contents of the structure that *esp* is pointing at (i.e., the contents of *es*) to *es2*. It is the same as

```
    struct example_struct es2;
    es2 = es;
```

16.2.2 Structure Pointer References

To reference a member of a structure, we use a "dot" syntax. For example, we might use the following code to set the *x* member of *es*.

```
    es.x = rndn(3,3);
```

To reference a member of a structure using a pointer to that structure, we use an "arrow" syntax. For example, we might use the following code to set the x member of *es* using the pointer *esp*:

```
esp->x = rndn(10,5);
```

This code will modify *es*, since *esp* is merely a pointer to *es*.

Structure pointers cannot be members of a structure. The following is illegal:

```
struct example_struct_2 {
   matrix z;
   struct example_struct *ep;
};
```

Therefore, since a structure pointer will never be a member of a structure, neither

```
sp1->sp2->x;
```

 nor

```
s.sp1->x;
```

will ever be valid (*sp1* and *sp2* are assumed to be structure pointers, *s* a structure instance, and *x* a matrix). The "arrow" (`->`) will only be valid if it is used for the first (or furthest left) dereference, as in:

```
sp1->st.x;
```

At this point we do not support indexing of structure pointers. Thus, a structure pointer should point at a scalar structure instance, not a matrix of structures. However, you may index members of that scalar structure instance. So, for example, let us suppose that you defined the following structure types:

```
struct sb {
   matrix y;
   matrix z;
```

```
   };
   struct sa {
      matrix x;
      struct structb s;
   };
```

and then created an instance of an *sa* structure, *a0*, setting *a0.s* to a 3x2 matrix of *sb* structures. The following would be legal:

```
   struct sa *sap
   sap = &a0;
   sap->s[3,1].y = rndn(3,3);
```

16.2.3 Using Structure Pointers in Procedures

Structure pointers are especially useful in cases where structures are passed into and out of procedures. If a procedure takes a structure as an argument and modifies any members of that structure, then it makes a local copy of the entire structure before modifying it. Thus if you want to have the modified copy of the structure after running the procedure, you need to pass the structure out of the procedure as one of its return arguments. For example:

```
   struct example_struct {
      matrix x;
      matrix y;
      matrix z;
   };

   proc product(struct example_struct es);
      es.z = (es.x).*(es.y);
      retp(es);
   endp;

   struct example_struct es1;
   es1.x = rndn(1000,100);
```

```
es1.y = rndn(1000,1);
es1 = product(es1);
```

In this example, the structure *es1* is passed into the procedure, copied and modified. The modified structure is then passed out of the procedure and assigned back to *es1*.

Structure pointers allow you to avoid such excessive data copying and eliminate the need to pass a structure back out of a procedure in cases like this. When you pass a structure pointer into a procedure and then modify a member of the structure that it references, the actual structure is modified rather than a local copy of it. Thus there is no need to pass the modifed structure back out of the procedure. For example, the above example could be accomplished using structure pointers as follows:

```
struct example_struct {
   matrix x;
   matrix y;
   matrix z;
};

proc(0) = product(struct example_struct *esp);
   esp->z = (esp->x).*(esp->y);
endp;

struct example_struct es1;
struct example_struct *es1p;
es1p = &es1;
es1.x = rndn(1000,100);
es1.y = rndn(1000,1);
product(es1p);
```

In this case, the procedure modifies the structure *es1*, which *es1p* is pointing at, instead of a local copy of the structure.

16.3 Special Structures

There are three common types of structures that will be found in the **GAUSS Run-Time Library** and applications.

The **DS** and **PV** structures are defined in the **GAUSS Run-Time Library**. Their definitions are found in ds.sdf and pv.sdf, respectively, in the src source code subdirectory.

Before structures, many procedures in the **Run-Time Library** and all applications had global variables serving a variety of purposes, such as setting and altering defaults. Currently, these variables are being entered as members of "control" structures.

Structures

16.3.1 The DS Structure

The **DS** structure, or "data" structure, is a very simple structure. It contains a member for each **GAUSS** data type. The following is found in ds.sdf:

```
//DS Structure definition
 struct DS {
   scalar type;
   matrix dataMatrix;
   array dataArray;
   string dname;
   string array vnames;
 };
```

This structure was designed for use by the various optimization functions in **GAUSS**, in particular, **sqpSolvemt**, as well as a set of gradient procedures, **gradmt**, **hessmt**, et al.

These procedures all required that the user provide a procedure computing a function (to be optimized or take the derivative of, etc.), which takes the **DS** structure as an argument before **GAUSS** version 16. These functions will still work with a **DS** structure, but you can now pass one or more matrices directly instead of placing them in a **DS** structure.

The **Run-Time Library** procedures such as **sqpSolvemt** take the extra data needed by the user provided objective or likelihood function (whether in a **DS** structure or as separate matrices) as argument(s) and pass it on to the user-provided procedure without modification.

To initialize an instance of a **DS** structure, the procedure **dsCreate** is defined in ds.src:

```
//Declare 'd0' to be an instance of a 'DS' structure
struct DS d0;

//Initialize structure
d0 = dsCreate();
```

16.3.2 The PV Structure

The **PV** structure, or **parameter vector** structure, is used by various optimization, modeling, and gradient procedures, in particular **sqpSolvemt**, for handling the parameter vector. The **GAUSS Run-Time Library** contains special functions that work with this structure. They are prefixed by "pv" and defined in pv.src. These functions store matrices and arrays with parameters in the structure, and retrieve various kinds of information about the parameters and parameter vector from it.

"Packing" into a PV Structure

The various procedures in the **Run-Time Library** and applications for optimization, modelling, derivatives, etc., all require a parameter vector. Parameters in complex models, however, often come in matrices of various types, and it has been the responsibility of the programmer to generate the parameter vector from the matrices and vice versa. The **PV** procedures make this problem much more convenient to solve.

The typical situation involves two parts: first, "packing" the parameters into the **PV** structure, which is then passed to the **Run-Time Library** procedure or application; and second, "unpacking" the **PV** structure in the user-provided procedure for use in computing the objective function. For example, to pack parameters into a **PV** structure:

```
#include sqpsolvemt.sdf

/* starting values */

b0 = 1;              //constant in mean equation
garch = { .1, .1 }; //garch parameters
arch = { .1, .1 };  //arch parameters
omega = .1;          //constant in variance equation
struct PV p0;
p0 = pvPack(pvCreate,b0, "b0");
p0 = pvPack(p0,garch, "garch");
p0 = pvPack(p0,arch, "arch");
p0 = pvPack(p0,omega, "omega");

/* data */

z = loadd("tseries");
struct DS d0;
d0.dataMatrix = z;
```

Next, in the user-provided procedure for computing the objective function, in this case minus the log-likelihood, the parameter vector is unpacked:

Structures

```
proc ll(struct PV p0, struct DS d0);
   local b0,garch,arch,omega,p,q,h,u,vc,w;
   b0 = pvUnpack(p0, "b0");
   garch = pvUnpack(p0, "garch");
   arch = pvUnpack(p0, "arch");
   omega = pvUnpack(p0, "omega");
   p = rows(garch);
   q = rows(arch);
   u = d0.dataMatrix - b0;
   vc = moment(u,0)/rows(u);
   w = omega + (zeros(q,q) | shiftr((u.*ones(1,q))',
   seqa(q-1,-1,q))) * arch;
   h = recserar(w,vc*ones(p,1),garch);
   logl = -0.5 * ((u.*u)./h + ln(2*pi) + ln(h));
   retp(logl);
endp;
```

Masked Matrices

The **pvUnpack** function unpacks parameters into matrices or arrays for use in computations. The first argument is a **PV** structure containing the parameter vector. Sometimes the matrix or vector is partly parameters to be estimated (that is, a parameter to be entered in the parameter vector) and partly fixed parameters. To distinguish between estimated and fixed parameters, an additional argument is used in the packing function called a "mask", which is strictly conformable to the input matrix. Its elements are set to 1 for an estimated parameter and 0 for a fixed parameter. For example:

```
p0 = pvPackm(p0,.1*eye(3), "theta", eye(3));
```

Here just the diagonal of a 3x3 matrix is added to the parameter vector.

When this matrix is unpacked, the entire matrix is returned with current values of the parameters on the diagonal:

```
print
pvUnpack(p0, "theta");
```

```
0.1000   0.0000   0.0000
0.0000   0.1000   0.0000
0.0000   0.0000   0.1000
```

Symmetric Matrices

Symmetric matrices are a special case because even if the entire matrix is to be estim-
ated, only the nonredundant portion is to be put into the parameter vector. Thus, for them
there are special procedures. For example:

```
vc = { 1 .6 .4, .6 1 .2, .4 .2 1 };
p0 = pvPacks(p0,vc, "vc");
```

There is also a procedure for masking in case only a subset of the nonredundant ele-
ments are to be included in the parameter vector:

```
vc = { 1 .6 .4, .6 1 .2, .4 .2 1 };
mask = { 1 1 0, 1 1 0, 0 0 1 };
p0 = pvPacksm(p0,vc, "vc",mask);
```

Fast Unpacking

When unpacking matrices using a matrix name, **pvUnpack** has to make a search
through a list of names, which is relatively time-consuming. This can be alleviated by
using an index rather than a name in unpacking. To do this, though, requires using a spe-
cial pack procedure that establishes the index:

```
p0 = pvPacki(p0,b0, "b0",1);
p0 = pvPacki(p0,garch, "garch",2);
p0 = pvPacki(p0,arch, "arch",3);
p0 = pvPacki(p0,omega, "omega",4);

//Now they may be unpacked using the index number:
b0 = pvUnpack(p0,1);
garch = pvUnpack(p0,2);
```

```
arch = pvUnpack(p0,3);
omega = pvUnpack(p0,4);
```

When packed with an index number, they may be unpacked either by index or by name, but unpacking by index is faster.

16.3.3 Miscellaneous PV Procedures

pvList

This procedure generates a list of the matrices or arrays packed into the structure:

```
p0 = pvPack(p0,b0, "b0");
p0 = pvPack(p0,garch, "garch");
p0 = pvPack(p0,arch, "arch");
p0 = pvPack(p0,omega, "omega");
print pvList(p0);

   b0
   garch
   arch
   omega
```

pvLength

This procedure returns the length of the parameter vector:

```
print
pvLength(p0);

   6.0000
```

pvGetParNames

This procedure generates a list of parameter names:

```
print
pvGetParNames(p0);

   b0[1,1]
   garch[1,1]
   garch[2,1]
   arch[1,1]
   arch[2,1]
   omega[1,1]
```

pvGetParVector

This procedure returns the parameter vector itself:

```
print
pvGetParVector(p0);

   1.0000
   0.1000
   0.1000
   0.1000
   0.1000
   1.0000
```

pvPutParVector

This procedure replaces the parameter vector with the one in the argument:

```
newp = { 1.5, .2, .2, .3, .3, .8 };
p0 = pvPutParVector(newp,p0);
print pvGetParVector(p0);
 1.5000
 0.2000
 0.2000
 0.3000
 0.3000
 0.8000
```

Structures

pvGetIndex

This procedure returns the indices in the parameter vector of the parameters in a matrix. These indices are useful when setting linear constraints or bounds in **sqpSolvemt**. Bounds, for example, are set by specifying a Kx2 matrix where K is the length of the parameter vector and the first column are the lower bounds and the second the upper bounds. To set the bounds for a particular parameter, then, requires knowing where that parameter is in the parameter vector. This information can be found using **pvGetIndex**. For example:

```
// get indices of lambda parameters in parameter vector
lind = pvGetIndex(par0, "lambda");

// set bounds constraint matrix to unconstrained default
c0.bounds = ones(pvLength(par0),1).*(-1e250~1e250);

// set bounds for lambda parameters to be positive
c0.bounds[lind,1] = zeros(rows(lind),1);
```

16.3.4 Control Structures

Another important class of structures is the "control" structure. Applications developed before structures were introduced into **GAUSS** typically handled some program specifications by the use of global variables which had some disadvantages, in particular, preventing the nesting of calls to procedures.

Currently, the purposes served by global variables are now served by the use of a control structure. For example, for **sqpSolvemt**:

```
struct
sqpSolvemtControl {
   matrix A;
   matrix B;
   matrix C;
   matrix D;
   scalar eqProc;
```

```
        scalar ineqProc;
        matrix bounds;
        scalar gradProc;
        scalar hessProc;
        scalar maxIters;
        scalar dirTol;
        scalar CovType;
        scalar feasibleTest;
        scalar maxTries;
        scalar randRadius;
        scalar trustRadius;
        scalar seed;
        scalar output;
        scalar printIters;
        matrix weights;
    };
```

The members of this structure determine optional behaviors of **sqpSolvemt**.

16.4 sqpSolveMT

sqpSolveMT is a procedure in the **GAUSS Run-Time Library** that solves the general
nonlinear programming problem using a Sequential Quadratic Programming descent
method, that is, it solves

   min f(Θ)

subject to

$A\Theta = B$	linear equality
$C\Theta >= D$	linear equality
$H(\Theta) = 0$	nonlinear equality
$G(\Theta) >= 0$	nonlinear equality
$\Theta_{lb} <= \Theta <= \Theta_{ub}$	bounds

The linear and bounds constraints are redundant with respect to the nonlinear constraints, but are treated separately for computational convenience.

The call to **sqpSolveMT** has four input arguments and one output argument:

```
out = sqpSolveMT(&fct, P, D, C);
```

16.4.1 Input Arguments

The first input argument is a pointer to the objective function to be minimized. The procedure computing this objective function has two arguments: a **PV** structure containing the start values, and a **DS** structure containing data, if any. For example:

```
proc fct(struct PV p0, struct DS d0);
   local y, x, b0, b, e, s;
   y = d0[1].dataMatrix;
   x = d0[2].dataMatrix;
   b0 = pvUnpack(p0, "constant");
   b = pvUnpack(p0, "coefficients");
   e = y - b0 - x * b;
   s = sqrt(e'e/rows(e));
   retp(-pdfn(e/s));
endp;
```

Note that this procedure returns a vector rather than a scalar. When the objective function is a properly defined log-likelihood, returning a vector of minus log-probabilities permits the calculation of a QML covariance matrix of the parameters.

The remaining input arguments are structures:

P	a **PV** structure containing starting values of the parameters
D	a **DS** structure containing data, if any
C	an **sqpSolvemtControl** structure

The **DS** structure is optional. **sqpSolvemt** passes this argument on to the user-provided procedure that `&fct` is pointing to without modification. If there is no data, a default structure can be passed to it.

sqpSolvemtControl Structure

A default **sqpSolvemtControl** structure can be passed in the fourth argument for an unconstrained problem. The members of this structure are as follows:

A	MxK matrix, linear equality constraint coefficients: $A\Theta = B$, where p is a vector of the parameters.
B	Mx1 vector, linear equality constraint constants: $A\Theta = B$, where p is a vector of the parameters.
C	MxK matrix, linear inequality constraint coefficients: $C\Theta = D$, where p is a vector of the parameters.
D	Mx1 vector, linear inequality constraint constants: $C\Theta = D$, where p is a vector of the parameters.
eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, instances of **PV** and **DS** structures, and one output argument, a vector of computed inequality constraints.

Default = .; i.e., no inequality procedure. |
| **IneqProc** | scalar, pointer to a procedure that computes the |

nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, instances of **PV** and **DS** structures, and one output argument, a vector of computed inequality constraints.

Default = .; i.e., no inequality procedure.

Bounds   1x2 or Kx2 matrix, bounds on parameters. If 1x2 all parameters have same bounds.

Default = -1e256 1e256.

GradProc   scalar, pointer to a procedure that computes the gradient of the function with respect to the parameters. When such a procedure has been provided, it has two input arguments, instances of **PV** and **DS** structures, and one output argument, the derivatives. If the function procedure returns a scalar, the gradient procedure returns a 1xK row vector of derivatives. If function procedure turns an Nx1 vector, the gradient procedure returns an NxK matrix of derivatives.

This procedure may compute a subset of the derivatives. **sqpSolvemt** will compute numerical derivatives for all those elements set to missing values in the return vector or matrix.

Default = .; i.e., no gradient procedure has been provided.

HessProc   scalar, pointer to a procedure that computes the Hessian; i.e., the matrix of second order partial derivatives of the function with respect to the parameters. When such a procedure has been provided, it has two input arguments, instances of

PV and DS structures, and one output argument, a vector of computed inequality constraints.

Default = .; i.e., no Hessian procedure has been provided.

Whether the objective function procedure returns a scalar or vector, the Hessian procedure must return a KxK matrix. Elements set to missing values will be computed numerically by **sqpSolvemt**.

MaxIters	scalar, maximum number of iterations. Default = 1e+5.
MaxTries	scalar, maximum number of attemps in random search. Default = 100.
DirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisifed, **sqpSolvemt** exits the iterations.
CovType	scalar, if 2, QML covariance matrix, else if 0, no covariance matrix is computed, else ML covariance matrix is computed. For a QML covariance matrix, the objective function procedure must return an Nx1 vector of minus log-probabilities.
FeasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = .001.
seed	scalar, if nonzero, seeds random number generator for random search, otherwise time in seconds from

	midnight is used.
trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = .001.
output	scalar, if nonzero, results are printed. Default = 0.
PrintIters	scalar, if nonzero, prints iteration information. Default = 0.
weights	vector, weights for objective function returning a vector. Default = 1.

16.4.2 Output Argument

The single output argument is an **sqpSolvemtOut** structure. Its definition is:

```
struct SQPsolveMTOut {
    struct PV par;
    scalar fct;
    struct SQPsolveMTLagrange lagr;
    scalar retcode;
    matrix moment;
    matrix hessian;
    matrix xproduct;
};
```

The members of this structure are:

par	instance of a **PV** structure containing the parameter estimates are placed in the matrix member *par*.
fct	scalar, function evaluated at final parameter estimates.

lagr	an instance of an **SQPLagrange** structure containing the Lagrangeans for the constraints. For an instance named **lagr**, the members are:	
	lagr.lineq	Mx1 vector, Lagrangeans of linear equality constraints
	lagr.nlineq	Nx1 vector, Lagrangeans of nonlinear equality constraints
	lagr.linineq	Px1 vector, Lagrangeans of linear inequality constraints
	lagr.nlinineq	Qx1 vector, Lagrangeans of nonlinear inequality constraints
	lagr.bounds	Kx2 matrix, Lagrangeans of bounds

Whenever a constraint is active, its associated Lagrangean will be nonzero. For any constraint that is inactive throughout the iterations as well as at convergence, the corresponding Lagrangean matrix will be set to a scalar missing value.

retcode	return	
	0	normal convergence
	1	forced exit
	2	maximum number of iterations exceeded
	3	function calculation failed
	4	gradient calculation failed
	5	Hessian calculation failed
	6	line search failed
	7	error with constraints
	8	function complex
	9	feasible direction couldn't be

found

16.4.3 Example

Define

$$Y = \Lambda\eta + \Theta$$

where Λ is a KxL matrix of `loadings`, η an Lx1 vector of unobserved "latent" variables, and Θ a Kx1 vector of unobserved errors. Then

$$\Sigma = \Lambda\Phi\Lambda'\Psi$$

where Φ is the LxL covariance matrix of the latent variables, and Ψ is the KxK covariance matrix of the errors.

The log-likelihood of the `ith` observation is

$$logP(i) = -\tfrac{1}{2}\,[\,Kln(2\pi) + ln|\,\pi\,| + Y(i)\Sigma Y(i)'\,]$$

Not all elements of Λ, Φ, and Ψ can be estimated. At least one element of each column of Λ must be fixed to 1, and Ψ is usually a diagonal matrix.

Constraints

To ensure a well-defined log-likelihood, constraints on the parameters are required to guarantee positive definite covariance matrices. To do this, a procedure is written that returns the eigenvalues of Σ and Φ minus a small number. **sqpSolvemt** then finds parameters such that these eigenvalues are greater than or equal to that small number.

16.4.4 The Command File

This command file can be found in the file sqpfact.e in the examples subdirectory:

```
#include sqpsolvemt.sdf
lambda = { 1.0 0.0,
           0.5 0.0,
           0.0 1.0,
           0.0 0.5 };
lmask = { 0 0,
          1 0,
          0 0,
          0 1 };
phi = { 1.0 0.3,
        0.3 1.0 };
psi = { 0.6 0.0 0.0 0.0,
        0.0 0.6 0.0 0.0,
        0.0 0.0 0.6 0.0,
        0.0 0.0 0.0 0.6 };
tmask = { 1 0 0 0,
          0 1 0 0,
          0 0 1 0,
          0 0 0 1 };

struct PV par0;
par0 = pvCreate;
par0 = pvPackm(par0,lambda, "lambda",lmask);
par0 = pvPacks(par0,phi, "phi");
par0 = pvPacksm(par0,psi, "psi",tmask);

struct SQPsolveMTControl c0;
c0 = sqpSolveMTcontrolCreate;
//get indices of lambda; parameters in parameter vector
lind = pvGetIndex(par0, "lambda");
//get indices of psi; parameters in parameter vector
tind = pvGetIndex(par0, "psi");


c0.bounds = ones(pvLength(par0),1).*(-1e250~1e250);
c0.bounds[lind,1] = zeros(rows(lind),1);
```

Structures

```
c0.bounds[lind,2] = 10*ones(rows(lind),1);
c0.bounds[tind,1] = .001*ones(rows(tind),1);
c0.bounds[tind,2] = 100*ones(rows(tind),1);
c0.output = 1;
c0.printIters = 1;
c0.trustRadius = 1;
c0.ineqProc = &ineq;
c0.covType = 1;

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = loadd("maxfact");
output file = sqpfact.out reset;

struct SQPsolveMTOut out0;
out0 = SQPsolveMT(&lpr,par0,d0,c0);

lambdahat = pvUnpack(out0.par, "lambda");
phihat = pvUnpack(out0.par, "phi");
psihat = pvUnpack(out0.par, "psi");

print "estimates";
print;
print "lambda" lambdahat;
print;
print "phi" phihat;
print;
print "psi" psihat;

struct PV stderr;
stderr = out0.par;

if not scalmiss(out0.moment);
   stderr = pvPutParVector(stderr,sqrt(diag(out0.moment)));
   lambdase = pvUnpack(stderr, "lambda");
   phise = pvUnpack(stderr, "phi");
```

```
    psise = pvUnpack(stderr, "psi");
    print "standard errors";
    print;
    print "lambda" lambdase;
    print;
    print "phi" phise;
    print;
    print "psi" psise;
endif;

output off;

proc lpr(struct PV par1, struct DS data1);
    local lambda,phi,psi,sigma,logl;
    lambda = pvUnpack(par1, "lambda");
    phi = pvUnpack(par1, "phi");
    psi = pvUnpack(par1, "psi");
    sigma = lambda*phi*lambda' + psi;
    logl = -lnpdfmvn(data1.dataMatrix,sigma);
  retp(logl);
endp;

proc ineq(struct PV par1, struct DS data1);
    local lambda,phi,psi,sigma,e;
    lambda = pvUnpack(par1, "lambda");
    phi = pvUnpack(par1, "phi");
    psi = pvUnpack(par1, "psi");
    sigma = lambda*phi*lambda' + psi;
    e = eigh(sigma) - .001;   //eigenvalues of sigma
    e = e | eigh(phi) - .001;  //eigenvalues of phi
    retp(e);
endp;
```

Structures

17 Run-Time Library Structures

Two structures are used by several **GAUSSRun-Time Library** functions for handling parameter vectors and data: the **PV** parameter structure and the **DS** data structure.

17.1 The PV Parameter Structure

The members of an instance of structure of type **PV** are all "private," that is, not accessible directly to the user. It is designed to handle parameter vectors for thread-safe optimization functions. Entering and receiving parameter vectors, and accessing properties of this vector, are accomplished using special functions.

Suppose you are optimizing a function containing a KxL matrix of coefficients. The optimization function requires a parameter vector but your function uses a KxL matrix. Your needs and the needs of the optimization function can be both satisfied by an instance of the structure of type **PV**. For example:

Run-Time Library
Structures

```
struct PV p1;
p1 = pvCreate;
//on input contains start values
//on exit contains estimates
x = zeros(4,3);
p1 = pvPack(p1,x, "coefficients");
```

The **pvCreate** function initializes **p1** to default values. **pvPack** enters the 4x3 matrix stored row-wise as a 12x1 parameter vector for the optimization function. The optimization program will pass the instance of the structure of type **PV** to your objective function.

By calling **pvUnpack** your 4x3 coefficient matrix is retrieved from the parameter vector. For example, in your procedure you have

```
x = pvUnpack(p1, "coefficients");
```

and now *x* is a 4x3 matrix of coefficients for your use in calculating the object function.

Suppose that your objective function has parameters to be estimated in a covariance matrix. The covariance matrix is a symmetric matrix where only the lower left portion contains unique values for estimation. To handle this, use **pvPacks**. For example:

```
struct PV p1;
p1 = pvCreate;
cov = { 1.0 0.1 0.1,
        0.1 1.0 0.1,
        0.1 0.1 1.0 };
p1 = pvPacks(p1,cov, "covariance");
```

Only the lower left portion of cov will be stored in the parameter vector. When the covariance matrix is unpacked, the parameters in the parameter vector will be entered into both the lower and upper portions of the matrix.

There may be cases where only a portion of a matrix being used to compute the objective function are parameters to be estimated. In this case use **pvPackm** with a "mask"

matrix that contains ones where parameters are to be estimated and zeros otherwise. For example,

```
struct PV p1;
p1 = pvCreate;
cov = { 1.0 0.5,
        0.5  1.0 };
mask = { 0 1,
         1 0 };
p1 = pvPacksm(p1,cov, "correlation",mask);
```

Here only the one element in the lower left of **cov** is stored in the parameter vector. Suppose the optimization program sends a trial value for that parameter of, say, .45. When the matrix is unpacked in your procedure it will contain the fixed values associated with the zeros in the mask as well as the trial value in that part of the matrix associated with the ones. Thus,

```
print unpack(p1, "correlation");
1.0000 .4500
.4500  1.0000
```

A mask may also be used with general matrices to store a portion of a matrix in the parameter vector.

```
struct PV p1;
p1 = pvCreate;
m = { 0.0 0.5 1.0,
      0.5 0.0 0.3 };
mask = { 0 1 1,
         1 0 0};
p1 = pvPackm(p1,m, "coefficients",mask);
```

A **PV** instance can, of course, hold parameters from all these types of matrices: symmetric, masked symmetric, rectangular, and masked rectangular. For example:

```
lambda = { 1.0 0.0,
           0.5 0.0,
           0.0 1.0,
           0.0 0.5 };

lmask = { 0 0,
          1 0,
          0 0,
          0 1 };
phi = { 1.0 0.3,
        0.3 1.0 };
theta = { 0.6 0.0 0.0 0.0,
          0.0 0.6 0.0 0.0,
          0.0 0.0 0.6 0.0,
          0.0 0.0 0.0 0.6 };
tmask = { 1 0 0 0,
          0 1 0 0,
          0 0 1 0,
          0 0 0 1 };
struct PV par0;
par0 = pvCreate;
par0 = pvPackm(par0,lambda, "lambda",lmask);
par0 = pvPacks(par0,phi, "phi");
par0 = pvPacksm(par0,theta, "theta",tmask);
```

It isn't necessary to know where in the parameter vector the parameters are located in order to use them in your procedure calculating the objective function. Thus:

```
lambda = pvUnpack(par1, "lambda");
phi = pvUnpack(par1, "phi");
theta = pvUnpack(par1, "theta");
sigma = lambda*phi*lambda' + theta;
```

Additional functions are available to retrieve information on the properties of the parameter vector. **pvGetParVector** and **pvPutParVector** get and put parameter vector from and into the **PV** instance, **pvGetParNames** retrieves names for the elements of the

parameter vector, **pvList** returns the list of matrix names in the **PV** instance, **pvLength**
the length of the parameter vector.

```
struct PV p1;
p1 = pvCreate;
cov = { 0.1 0.5,
        0.5 1.0 };
mask = { 0 1,
         1 0 };

p1 = pvPacksm(p1,cov, "correlation",mask);

print pvGetParVector(p1);
    0.5000

p1 = pvPutParVector(p1,.8);

print pvGetParVector(p1);
    0.8000
print pvUnpack(p1, "correlation");
    1.0000 .8000
    0.8000 1.0000
print pvGetParNames(p1);

correlation[2,1]

print pvLength(p1);
    1.0000
```

Also, **pvTest** tests an instance to make sure it is properly constructed. **pvCreate** gen-
erates an initialized instance, and **pvGetIndex** returns the indices of the parameters
of an input matrix in the parameter vector. This last function is most useful when con-
structing linear constraint indices for the optimization programs.

17.2 Fast Pack Functions

Unpacking matrices using matrix names is slow because it requires a string search through a string array of names. A set of special packing functions are provided that avoid the search altogether. These functions use a "table" of indices that you specify to find the matrix in the **PV** instance. For example:

```
struct PV p1;
p1 = pvCreate;
y = rndn(4,1);
x = rndn(4,4);
p1 = pvPacki(p1,y, "Y",1);
p1 = pvPacki(p1,x, "X",2);

print pvUnpack(p1,1);

  .3422
  .0407
  .5611
  .0953

print pvUnpack(p1, "Y");

  .3422
  .0407
  .5611
  .0953
```

The call to **pvPacki** puts an entry in the table associating the matrix in its second argument with the index 1. As indicated above the matrix can be unpacked either by index or by name. Unpacking by index, however, is much faster than by name.

Note that the matrix can be unpacked using either the index or the matrix name.

There are index versions of all four of the packing functions, **pvPacki**, **pvPackmi**, **pvPacksi**, and **pvPacksmi**.

17.3 The DS Data Structure

An instance of the **DS** data structure contains the following members:

```
struct  DS d0;
```

d0.*dataMatrix*	MxK matrix, data
d0.*dataArray*	N-dimensional array, data
d0.*type*	scalar
d0.*dname*	string
d0.*vnames*	string array

The definition and use of the elements of **d0** are determined by the particular application and are mostly up to the user. A typical use might use a vector of structures. For example, suppose the objective function requires a vector of observations on a dependent variable as well as on K independent variables. Then:

```
struct DS d0;
d0 = dsCreate;

y = rndn(20,1);
x = rndn(20,5);

d0 = reshape(d0,2,1);
d0[1].dataMatrix = y;
d0[2].dataMatrix = X;
```

The **d0** instance would be passed to the optimization program which then passes it to your procedure computing the objective function. For example:

```
 proc
   lpr(struct PV p1, struct DS d1);
   local u;
   u = d0[1].dataMatrix - d0[2].dataMatrix * pvUnpack(p1,
"beta");
  retp(u'u);
endp;
```

A particular application may require setting other members of the **DS** instance for particular purposes, but in general you may use them for your own purposes. For example, **d0.dname** could be set to a **GAUSS** dataset name from which you read the data in the objective function procedure, or **d0.vnames** could be set to the variable names of the columns of the data stored in **d0.dataMatrix**, or **d0.type** could be an indicator variable for the elements of a vector of **DS** instances.

The following are complete examples of the use of the **PV** and **DS** structures. The first example fits a set of data to the Micherlitz model. It illustrates packing and unpacking by index.

```
#include sqpsolvemt.sdf
struct DS Y;
Y = dsCreate;
Y.dataMatrix = 3.183|
               3.059|
               2.871|
               2.622|
               2.541|
               2.184|
               2.110|
               2.075|
               2.018|
               1.903|
               1.770|
               1.762|
               1.550;

struct DS X;
X = dsCreate;
X.dataMatrix = seqa(1,1,13);

struct DS Z;
Z = reshape(Z,2,1);
```

```
Z[1] = Y;
Z[2] = X;

struct SQPsolveMTControl c1;
c1 = sqpSolveMTcontrolCreate; // initializes to default val-
ues

c1.bounds = 0~100;              // constrains parameters to be
positive
c1.CovType = 1;
c1.output = 1;
c1.printIters = 0;
c1.gradProc = &grad;

struct PV par1;
par1 = pvCreate;
start = { 2, 4, 2 };
par1 = pvPacki(par1,start, "Parameters",1);

struct SQPsolveMTout out1;
out1 = SQPsolveMT(&Micherlitz,par1,Z,c1);
estimates = pvGetParVector(out1.par);

print " parameter estimates ";
print estimates;
print;
print " standard errors ";
print sqrt(diag(out1.moment));

proc Micherlitz(struct PV par1,struct DS Z);
   local p0,e,s2;
   p0 = pvUnpack(par1,1);
   e = Z[1].dataMatrix - p0[1] - p0[2]*exp(-p0[3]
   *Z[2].dataMatrix);
   s2 = moment(e,0)/(rows(e)-1);
  retp( (2/rows(e))*(e.*e/s2 + ln(2*pi*s2)));
endp;
```

```
proc grad(struct PV par1, struct DS Z);
   local p0,e,e1,e2,e3,w,g,s2;
   p0 = pvUnpack(par1,1);
   w = exp(-p0[3]*Z[2].dataMatrix);
   e = z[1].dataMatrix - p0[1] - p0[2] * w;
   s2 = moment(e,0) / rows(e);
   e1 = -ones(rows(e),1);
   e2 = -w;
   e3 = p0[2]*Z[2].dataMatrix.*w;
   w =(1 - e.*e / s2) / rows(e);
   g = e.*e1 + w*(e'e1);
   g = g ~ (e.*e2 + w*(e'e2));
   g = g ~ (e.*e3 + w*(e'e3));
  retp(4*g/(rows(e)*s2));
endp;
```

This example estimates parameters of a "confirmatory factor analysis" model.

```
#include sqpsolvemt.sdf
lambda = { 1.0 0.0,
           0.5 0.0,
           0.0 1.0,
           0.0 0.5 };
lmask = { 0 0,
          1 0,
          0 0,
          0 1 };
phi = { 1.0 0.3,
        0.3 1.0 };
theta = { 0.6 0.0 0.0 0.0,
          0.0 0.6 0.0 0.0,
          0.0 0.0 0.6 0.0,
          0.0 0.0 0.0 0.6 };
tmask = { 1 0 0 0,
          0 1 0 0,
```

```
          0 0 1 0,
          0 0 0 1 };

struct PV par0;
par0 = pvCreate;
par0 = pvPackm(par0,lambda, "lambda",lmask);
par0 = pvPacks(par0,phi, "phi");
par0 = pvPacksm(par0,theta, "theta",tmask);

struct SQPsolveMTControl c0;
c0 = sqpSolveMTcontrolCreate;

lind = pvGetIndex(par0, "lambda"); // get indices of lambda
parameters  // in parameter vector

tind = pvGetIndex(par0, "theta"); // get indices of theta
parameters // in parameter vector

c0.bounds = ones(pvLength(par0),1).*(-1e250~1e250);
c0.bounds[lind,1] = zeros(rows(lind),1);
c0.bounds[lind,2] = 10*ones(rows(lind),1);
c0.bounds[tind,1] = .001*ones(rows(tind),1);
c0.bounds[tind,2] = 100*ones(rows(tind),1);
c0.ineqProc = &ineq;
c0.covType = 1;

struct DS d0;
d0 = dsCreate;
d0.dataMatrix = loadd("maxfact");

struct SQPsolveMTOut out0;
out0 = SQPsolveMT(&lpr,par0,d0,c0);

lambdahat = pvUnpack(out0.par, "lambda");
phihat = pvUnpack(out0.par, "phi");
thetahat = pvUnpack(out0.par, "theta");
```

```
print "estimates";
print;
print "lambda" lambdahat;
print;
print "phi" phihat;
print;
print "theta" thetahat;

struct PV stderr;
stderr = out0.par;

if not scalmiss(out0.moment);
   stderr = pvPutParVector(stderr,sqrt(diag(out0.moment)));
   lambdase = pvUnpack(stderr, "lambda");
   phise = pvUnpack(stderr, "phi");
   thetase = pvUnpack(stderr, "theta");
   print"standard errors";
   print;
   print"lambda" lambdase;
   print;
   print"phi" phise;
   print;
   print"theta" thetase;
endif;

proc lpr(struct PV par1, struct DS data1);
   local lambda,phi,theta,sigma,logl;
   lambda = pvUnpack(par1, "lambda");
   phi = pvUnpack(par1, "phi");
   theta = pvUnpack(par1, "theta");
   sigma = lambda*phi*lambda' + theta;
   logl = -lnpdfmvn(data1.dataMatrix,sigma);
  retp(logl);
endp;

proc ineq(struct PV par1, struct DS data1);
```

```
      local lambda,phi,theta,sigma,e;
      lambda = pvUnpack(par1, "lambda");
      phi = pvUnpack(par1, "phi");
      theta = pvUnpack(par1, "theta");
      sigma = lambda*phi*lambda' + theta;
      e = eigh(sigma) - .001;   // eigenvalues of sigma
      e = e | eigh(phi) - .001; // eigenvalues of phi
    retp(e);
endp;
```

18 Multi-Threaded Programming in GAUSS

The term thread comes from the phrase "thread of execution"--simply, it denotes a section of code that you want to execute. A single-threaded program has only one thread of execution, i.e., the program itself. A multi-threaded program is one that can have multiple threads--sections of code--executing **simultaneously**. Since these threads are part of the same program, they share the same workspace, and see and operate on the same symbols. Threads allow you to take full advantage of the hardware processing resources available on hyper-threaded, multi-core, and multi-processor systems, executing independent calculations simultaneously, combining and using the results of their work when done.

18.1 The Functions

GAUSS includes four keywords for multi-threading your programs:

ThreadStat	Marks a single statement to be executed as a thread.
ThreadBegin	Marks the beginning of a block of code to be executed as a thread.
ThreadEnd	Marks the end of a block of code to be executed as a thread.

`threadfor, threadendfor`	Begins a parallel for loop.
`ThreadJoin`	Completes the definition of a set of threads, waits until they are done.

`ThreadStat` defines a single statement to be executed as a thread:

```
ThreadStat n = m'm;
```

`ThreadBegin` and `ThreadEnd` define a multi-line block of code to be executed as a thread:

```
ThreadBegin;
    y = x'x;
    z = y'y;
ThreadEnd;
```

Together these define **sets** of threads to be executed concurrently:

```
ThreadStat n = m'm;        // Thread 1
ThreadBegin;               // Thread 2
    y = x'x;
    z = y'y;
ThreadEnd;
ThreadBegin;               // Thread 3
    q = r'r;
    r = q'q;
ThreadEnd;
ThreadStat p = o'o;        // Thread 4
```

Finally, `ThreadJoin` completes the definition of a set of threads. It waits for the threads in a set to finish and rejoin the creating (the **parent**) thread, which can then continue, making use of their individual calculations:

```
ThreadBegin;               // Thread 1
    y = x'x;
```

```
    z = y'y;
ThreadEnd;
ThreadBegin;                    // Thread 2
    q = r'r;
    r = q'q;
ThreadEnd;
ThreadStat n = m'm;             // Thread 3
ThreadStat p = o'o;             // Thread 4
ThreadJoin;                     // waits for Threads 1-4 to finish
b = z + r + n'p;                // Using the results
```

18.2 GAUSS Threading Concepts

This is really the one and only thing you need to know about threads: threads are separate sections of the same program, executing simultaneously, operating on the same data. In fact, it's so fundamental it's worth saying again: threads are separate sections of code in a program, running at the same time, using the same workspace, referencing and operating on the same symbols.

This raises basic issues of workflow and data integrity. How do you manage the creation and execution of threads, and make use of the work they do? And how do you maintain data integrity? (You **do not** want two threads assigning to the same symbol at the same time.)

To handle thread workflow, **GAUSS** employs a **split-and-join** approach. At various points in your program (as many as you like), you define a set of threads that will be created and run as a group. When created, the threads in the set execute simultaneously, each doing useful work. The parent thread waits for the created threads to complete, then continues, the results of their work now available for further use.

To maintain data integrity, we introduce the **writer-must-isolate** (informally, the **any-thread-can-read-unless-some-thread-writes**) programming rule. That is to say, symbols that are read from but not assigned to can be referenced by as many threads in a set as you like. Symbols that are assigned to, however, must be **wholly** owned by a single thread. No other thread in the set can reference that symbol. They cannot assign to it, nor can they read from it. They cannot refer to it at all.

Note: the **writer-must-isolate** rule only applies to the threads within a given set (including any child thread sets they may create). It does not apply between thread sets that have no chance of running simultaneously.

For threads defined in the main code, the **writer-must-isolate** rule applies to the global symbols. For threads defined in procedures or keywords, it applies to the global symbols, local symbols, and the procedure/keyword arguments.

18.3 Coding With Threads

There are two main points to coding with threads.

1. **You can define threads anywhere.**

   You can define them in the main code, you can define them in `proc`'s and `keyword`'s, and yes, you can define them inside other threads.

2. **You can call `proc`'s and `keyword`'s from threads.**

   This is what really ties everything together. You can call a `proc` from a thread, and that `proc` can create threads, and any of those threads can call `proc`'s, and any of those `proc`'s can create threads, and ... you get the picture.

So--you can do things like this:

```
q = chol(b);
ThreadBegin;
    x = q + m;
    ThreadBegin;
        y = x'x;
        z = q'm;
    ThreadEnd;
    ThreadBegin;
        a = b + x;
        c = a + m;
    ThreadEnd;
    ThreadJoin;
q = m'c;
ThreadEnd;
```

```
ThreadBegin;
   ThreadStat r = m'm;
   ThreadStat s = m + inv(b);
   ThreadJoin;
t = r's;
ThreadEnd;
ThreadJoin;
x = r+s+q+z-t;
```

More importantly, you can do things like this:

```
 proc
   bef(x);
   local y,t;
   ThreadStat y = nof(x);
   ThreadStat t = dof(x'x);
   ThreadJoin;
   t = t+y;
  retp(t);
endp;

proc abr(m);
   local x,y,z,a,b;
   a = m'm;
   ThreadStat x = inv(m);
   ThreadStat y = bef(m);
   ThreadStat z = dne(a);
   ThreadJoin;
   b = chut(x,y,z,a);
  retp(inv(b));
endp;

s = rndn(500,500);
ThreadStat t = abr(s);
ThreadStat q = abr(s^2);
ThreadStat r = che(s);
ThreadJoin;
```

```
w = del(t,q,r);
print w[1:10,1:10];
```

This means you can multi-thread anything you want, and call it from anywhere. You can multi-thread all the proc's and keyword's in your libraries, and call them freely anywhere in your multi-threaded programs.

18.4 Coding Restrictions

A few points on coding restrictions. First, you can't interlace thread definition statements and regular statements. You can't do this:

```
ThreadStat a = b'b;
n = q;
ThreadStat c = d'd;
ThreadJoin;
```

Or this:

```
if k == 1;
    ThreadStat a = b'b;
elseif k == 2;
    ThreadStat a = c'c;
endif;
if j == 1;
    ThreadStat d = e'e;
elseif j == 2;
    ThreadStat d = f'f;
endif;
ThreadJoin;
```

Each set of threads is defined as a group, and always completed by a ThreadJoin, like this:

```
n = q;
ThreadStat a = b'b;
```

```
ThreadStat c = d'd;
ThreadJoin;
```

And this:

```
ThreadBegin;
   if k == 1;
      a = b'b;
   elseif k == 2;
      a = c'c;
   endif;
ThreadEnd;
ThreadBegin;
   if j == 1;
      d = e'e;
   elseif j == 2;
      d = f'f;
   endif;
ThreadEnd;
ThreadJoin;
```

Second--as stated above, you can reference read-only symbols in as many threads within a set as you like, but any symbols that are assigned to must be **wholly** owned by a single thread. A symbol that is assigned to by a thread cannot be written **or read** by any other thread in that set. This is the **writer-must-isolate** rule.

So, you can do this:

```
ThreadStat x = y'y;
ThreadStat z = y+y;
ThreadStat a = b-y;
ThreadJoin;
```

You cannot do this:

```
ThreadStat x = y'y;
Threadstat z = x'x;
```

```
ThreadStat a = b-y;
ThreadJoin;
```

This is because the threads within a set run simultaneously. Thus, there is no way of knowing when an assignment to a symbol has taken place, no way of knowing in one thread the "state" of a symbol in another.

Let's revisit the nested thread example for a minute and see how the **writer-must-isolate** rule applies to it:

```
q = chol(b);              // main code, no threads yet
ThreadBegin;              // Th1: isolates x,y,z,a,c,q from
Th2
   x = q + m;
   ThreadBegin;           // Th1.1: isolates y,z from 1.2
      y = x'x;
      z = q'm;
   ThreadEnd;
   ThreadBegin;           // Th1.2: isolates a,c from 1.1
      a = b + x;
      c = a + m;
   ThreadEnd;
   ThreadJoin;            // Joins 1.1, 1.2
   q = m'c;
ThreadEnd;
ThreadBegin;              // Th2: isolates r,s,t from Th1
ThreadStat r = m'm;       // Th2.1: isolates r from 2.2
ThreadStat                // Th2.2: isolates s from 2.1
   s = m + inv(b);
   ThreadJoin;            // Joins 2.1, 2.1
   t = r's;
ThreadEnd;
ThreadJoin;               // Joins Th1, Th2
x = r+s+q+z-t;
```

The main point here is that any symbols a thread **or its children** assign to must be isolated from all the other threads (and their children) of the same nesting level in that set.

On the other hand, the children of a thread can freely read/write symbols that are read/written by their parent, because there is no risk of simultaneity; they must only isolate written symbols from their siblings and siblings' offspring.

If you break the **writer-must-isolate** rule, your program (and probably **GAUSS**) will crash. Worse, until it crashes, it will be happily producing indeterminate results.

Finally--the ThreadEnd command is what tells a thread to terminate, so you mustn't write code that keeps a thread from reaching it. For example, don't retp from the middle of a thread:

```
ThreadStat m = imt(9);
ThreadBegin;
    x = q[1];
    if x = 1;
      retp(z);
    else;
       r = z + 2;
    endif;
ThreadEnd;
ThreadJoin;
```

And don't use goto to jump into or out of the middle of a thread:

```
retry:
ThreadBegin;
    { err, x } = fna(q);
    if err;
       goto badidea;
    endif;
    x = fnb(x);
ThreadEnd;
ThreadStat y = fnb(y);
ThreadJoin;

z = fnc(x,y);
save z;
end;
```

```
badidea:
errorlog "Error computing fna(q)";
q = fnd(q);

goto retry;
```

Basically, don't do anything that will keep a thread from reaching its `ThreadEnd` command. That's the only way it knows its work is done. `end` and `stop` are okay to call, though--they will bring the program to an end as usual, and terminate all running threads in the process.

(You **can** use `goto` and labels to jump around within a thread--that is, within the confines of a `ThreadBegin`/`ThreadEnd` pair.)

18.5 Parallel for Loops

The GAUSS `threadFor` keyword allows you to create compact code that will run in parallel. In order for a for loop construct to execute correctly concurrently in separate threads, the `threadFor` loop has a few special rules. In this section, we will explain these rules by working through a simple example.

18.5.1 Basic syntax

The syntax of a `threadFor` loop is the same as a standard for loop:

```
threadFor loop_counter(start, stop, step);
    //loop body
threadEndFor;
```

Below is one of the most simple, specific examples of a `threadFor` loop. It will print out the loop counter from one to ten:

```
threadFor i(1, 10, 1);
    print i;
threadEndFor;
```

18.5.2 Single threaded example

As we learn how to use `threadFor` loops, we will illustrate the concepts by converting the following single-threaded for loop to a `threadFor` loop:

```
//Using the trapezoidal rule to integrate//the cosine func-
tion from 0 to 0.5
n_slices = 1000;
a = 0;
b = 0.5;

h = (a - b) ./ n_slices;

approx = zeros(n_slices + 1, 1);
approx[1] = (cos(a) + cos(b)) ./ 2;

for i(1, n_slices, 1);
    x_tmp = a + i*h;
    approx[i+1] = cos(x_tmp);
endfor;

final = sumc(approx) .* h;
```

18.5.3 threadFor variables

A `threadFor` loop contains three types of variables: temporary variables, broadcast variables and slice variables. The next sections will explain how they are defined and how they operate. Before we start it is important to point out that the assignment of a

variable to a category is permanent for the remainder of the `threadFor` loop. Variables cannot change to different categories during a `threadFor` loop.

18.5.4 Temporary variables

One of the most important rules of parallel programming is that you should not assign to the same global variable from multiple threads that are running at the same time. This can cause a "race condition" in which the output from your program will depend upon which thread "wins the race" and assigns to the variable first. It can also cause your program to crash if they try to assign at the same time.

With that in mind, it may appear that we have a problem when we try to convert the first line inside the for loop from our program above to a `threadFor` loop:

```
threadFor i(1, n_slices, 1);
    x_tmp = a + i*h;
    approx[i+1] = cos(x_tmp);
threadEndFor;
```

In the sequential version, every iteration of the for loop is certainly assigning to the same global variable, x_tmp. While we might expect that this is not allowed in the `threadFor` loop, it is actually acceptable.

The reason that this works is because GAUSS will make a separate temporary version of x_tmp for each thread. When a variable in a `threadFor` loop is assigned to in it's entirety, not simply a portion of it by index, it becomes a temporary variable.

The rules for temporary variables are:

1. A temporary variable may have any legal variable name

   In our example above, our variable name ended in $\_tmp$, but this is not required.

2. A temporary variable is not the same as a variable with the same name outside the 'threadFor' loop.

Temporary variables exist only in the scope of the `threadFor` loop. They do not start the loop with any value and their value is not available outside the `threadFor` loop. For example:

```
x = 100;
threadFor i(1, 4, 1);
    x = i;
threadEndFor;
print "x = " x;
```

will print out:

```
x = 100
```

3. The first reference to a temporary variable must be an assignment.

Since temporary variables do not exist outside the loop and are not related to variables outside the loop with the same name, they will not have a value until they are assigned to inside of the 'threadFor' loop. For example:

```
x = 100;
threadFor i(1, 4, 1);
    a = x; //'x' does not have a value until the next line
    x = i; //This line makes 'x' a temporary
variablethreadEndfor;
```

18.5.5 Broadcast variables

```
threadFor i(1, n_slices, 1);
    x_tmp = a + i*h; //'a' and 'h' are broadcast variables
    approx[i+1] = cos(x_tmp);
threadEndFor;
```

Besides x_tmp which we discussed above, were two other variables on the first line of our loop, 'h' and 'a' which were only referenced. They were not assigned to on that line or anywhere else in the loop. These variables are called broadcast variables, because the same value is "broadcast" to every thread. If a variable is not assigned to and it is not the loop counter, it will be, by default, a broadcast variable.

18.5.6 Slice variables

Next we will look at the indexed assignment of $approx$ on the second line inside our loop.

```
threadFor i(1, n_slices, 1);
    x_tmp = a + i*h; //'a' and 'h' are broadcast variables
    approx[i+1] = cos(x_tmp);
threadEndFor;
```

Earlier we discussed that you are not allowed to assign to the same global variable from more than one thread at one time. It is allowable to assign to portions of a variable, however, as long as multiple threads never assign to the same location.

If a variable is assigned to by index it becomes a slice variable. In order to ensure correct multi-threaded programs, the rules for 'slice variables' are:

1. The dimension that is assigned to using the threadFor loop counter defines the slice. The index into this dimension may contain, in addition to the loop counter, a commutative operator, such as +, - or .*, a constant or a variable that is constant during the threadFor loop. For example, assuming that 'i' is the threadFor loop counter and that $approx$ is an nx1 vector, any of these statements would be correct:

    ```
    approx[i] = cos(x_tmp);
    approx[i+1] = cos(x_tmp);
    approx[n-i] = cos(x_tmp);
    ```

    These statements, however, would be incorrect:

```
approx[maxc(idx)] = cos(x_tmp); //function call cannot be
an index
approx[8] = cos(x_tmp); //each iteration would assign to
the same location
approx[i:i+5] = cos(x_tmp); //ranges are not allowed to
define the slice
```

2. The dimension or dimensions that do not define the slice may be assigned to freely. These statements would all be legal:

```
approx[i, maxc(idx) = cos(x_tmp);
approx[i+1, .] = cos(x_tmp);
approx[i+3, 1:10] = cos(x_tmp);
```

18.5.7 Reduction variables

A reduction variable is a variable that accumulates the result of a commutative operator over all the iterations of a threadFor loop. For example, if we changed our example loop to this:

```
approx = (cos(a) + cos(b)) ./ 2;

for i(1, n_slices, 1);
    x_tmp = a + i*h;
    approx = approx + cos(x_tmp);
endfor;
```

then 'approx' would be a an reduction variable. Reduction variables are not yet supported in GAUSS threadFor statements as of the release of this version. To perform a reduction, instead assign the result from each iteration to an element of an array and perform a 'sumc' or 'prodc' on this final vector as is shown in this example.

18.5.8 Categorizing the variables in our example

Take a moment and look through the loop below. Can you identify which classification each of the variables belongs to? Based upon what we have covered, is this a legal `threadFor` loop?

```
for i(1, n_slices, 1);
    x_tmp = a + i*h;
    approx[i+1] = cos(x_tmp);
endfor;
```

After you have tried to categorize the variables, look at the table below to compare your answers and see if your reasoning is correct:

i	The loop counter.
x_tmp	Temporary variable, it is assigned to in its entirety.
a	Broadcast variable, it is never assigned to.
h	Broadcast variable, it is never assigned to.
approx	Slice variable, it is assigned by index, using the loop counter.

To the second question, yes the code above is a legal `threadFor` loop. Below is the entire program converted to use `threadFor`.

```
//Using the trapezoidal rule to integrate//the cosine func-
tion from 0 to 0.5
n_slices = 1000;
a = 0;
b = 0.5;

h = (a - b) ./ n_slices;

approx = zeros(n_slices + 1, 1);
```

```
approx[1] = (cos(a) + cos(b)) ./ 2;

threadFor i(1, n_slices, 1);
    x_tmp = a + i*h;
    approx[i+1] = cos(x_tmp);
threadEndFor;

final = sumc(approx) .* h;
```

18.6 Controlling thread count

By default when dividing up the iterations of a threadFor loop, **GAUSS** will create one thread for each core on your machine. This is a good setting in many cases. You may, however, override this behavior if you like with the **sysstate** command, case 42. Here is an example of setting the number of threads that threadFor will create to 16:

```
new_num = 16;
old_num = sysstate(42,new_num);
```

After the above code, *old_num* will be a 2x1 vector. The first element will be the old thread count used by threadFor and the second element is reserved for future use.

18.7 Threading Performance Considerations

- Oversubscription
- Thread affinity
- Excessive memory utilization
- Insufficient work
- Other issues

18.7.1 Oversubscription

At any given time, many different processes are all running on your machine. In fact, if you look at your 'Task Manager' on Windows or 'Activity Monitor' on Mac, you may see that there are more running processes on your computer than you have cores.

To handle this, your operating system gives CPU 'time slices' to the threads that are attempting to run. This thread swapping can cause a few problems. The most obvious is that if you create a program that creates a thread for each core, not all of your threads will be able to run without interruption. If the remainder of your background processes are mostly idle, this should not be a significant problem. However, as the number of threads actively seeking CPU time increases, the time spent waiting for CPU resources will increase. This will obviously slow down your algorithm.

As we will see, there are many factors related to the performance of threaded programs. Keeping the number of threads at or below the number of cores is not always best for performance. However, the costs of context switching should be kept in mind when trying to get the most performance from your program.

18.7.2 Thread affinity

Another side effect of the operating system swapping threads between cores is 'cache thrashing'. When your thread is restarted, it may be on a different core than it was previously and the data it was working with may have been evicted from cache.

Cache reuse is one of the most important aspects of computing performance on modern computers. The difference between an algorithm that efficiently uses cache versus one that does not, can easily be 10x or more.

The solution to this problem is thread pinning or thread affinity. These settings will depend upon your operating system. On Windows, you can (as of this writing) right-click on a process in the task manager and select 'set Affinity'. On Linux some options include 'numactl' and the environment variable KMP_SET_AFFINITY.

18.7.3 Excessive memory utilization

If each thread in your code creates large amounts of data, then the total memory requirements can grow very rapidly. Sometimes this will cause a failure to allocate enough memory. Other times, however, the operating system will continue to provide what is called 'virtual memory'. This is when the operating system uses part of the hard drive for extra memory space. Due to the poor performance of disk reads and writes, this will slow the program substantially. You may even see CPU usage as low as 30%, or slowdowns of 100x or more, as the threads are waiting for data to be read from or written to disk.

18.7.4 Insufficient work

There is some overhead in the creation, destruction and scheduling of threads. Therefore, for best performance, the threaded code must have enough work to do to overcome this added overhead.

18.7.5 Other issues

- The bandwidth of the memory system does not increase as you use more cores, so the memory bandwidth available to each core is smaller than for a single threaded program. This can cause threaded programs to spend more time waiting for data from memory.
- Many **GAUSS** functions are already internally threaded. If one of these functions dominates the time of the program run, there may be little or no opportunity to improve the performance of the program compared to the version without **threadfor**, or **threadbegin/end**.
- The usage of variables from the scope of the calling function makes the performance of **fn**'s poor inside of threadfor loops. You should always use procs instead of fn's in this case.

19 Libraries

The **GAUSS** library system allows for the creation and maintenance of modular programs. The user can create "libraries" of frequently used functions that the **GAUSS** system will automatically find and compile whenever they are referenced in a program.

19.1 Autoloader

The autoloader resolves references to procedures, keywords, matrices, and strings that are not defined in the program from which they are referenced. The autoloader automatically locates and compiles the files containing the symbol definitions that are not resolved during the compilation of the main file. The search path used by the autoloader is first the current directory, and then the paths listed in the src_path configuration

variable in the order they appear. *src_path* can be defined in the **GAUSS** configuration file.

19.1.1 Forward References

When the compiler encounters a symbol that has not previously been defined, that is called a "forward reference." **GAUSS** handles forward references in two ways, depending on whether they are "left-hand side" or "right-hand side" references.

Left-Hand Side

A left-hand side reference is usually a reference to a symbol on the left-hand side of the equal sign in an expression.

```
x = 5;
```

Left-hand side references, since they are assignments, are assumed to be matrices. In the statement above, x is assumed to be a matrix and the code is compiled accordingly. If, at execution time, the expression actually returns a string, the assignment is made and the type of the symbol x is forced to string.

Some commands are implicit left-hand side assignments. There is an implicit left-hand side reference to x in each statement below:

```
clear x;
load x;
open x = myfile;
```

Right-Hand Side

A right-hand side reference is usually a reference to a symbol on the right-hand side of the equal sign in an expression 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.

19.1.2 The Autoloader Search Path

If the autoloader is OFF, no forward references are allowed. Every procedure, matrix, and string referenced by your program must be defined before it is referenced. An external statement can be used above the first reference to a symbol, but the definition of the symbol must be in the main file or in one of the files that are #include'd. No global symbols are deleted automatically.

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

```
x = rndn(10,10);
y = sym(x);      // Forward reference to sym

proc sym(x);
   retp(x+x');
endp;
```

Use an `external` statement for anything referenced above its definition if autodelete is OFF:

```
external proc sym;

x = rndn(10,10);
y = sym(x);

proc sym(x);
   retp(x+x');
endp;
```

When autodelete is OFF, symbols not found in an active library will not be added to the symbol table. This prevents the creation of uninitialized procedures in the global symbol table. No deletion of symbols from the global symbol table will take place.

Libraries

The first place **GAUSS** looks for a symbol definition is in the "active" libraries. A **GAUSS** library is a text file that serves as a dictionary to the source files that contain the symbol definitions. When a library is active, **GAUSS** will look in it whenever it is looking for a symbol it is trying to resolve. The `library` statement is used to make a library active. Library files should be located in the subdirectory listed in the *lib_ path* configuration variable. Library files have an `.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 auto-loader will be able to find the procedures defined in that file whenever they are called.

First, create the file that is to contain your desired procedure definitions. By convention, this file is usually named with a `.src` extension, but you may 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. It is called `norm.src`:

```
/*
** norm.src
**
** This is a file containing the definitions of three
** procedures which return the norm of a matrix x.
** The three norms calculated are the 1-norm, the
** inf-norm and the E-norm.
*/

proc onenorm(x);
  retp(maxc(sumc(abs(x))));
endp;

proc infnorm(x);
  retp(maxc(sumc(abs(x'))));
endp;
```

```
proc Enorm(x);
  retp(sumc(sumc(x.*x)));
endp;
```

Next, create a library file that contains the name of the file you want access to, and the list of symbols defined in it. This can be done with the `lib` command. (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.
`array`	N-dimensional array.
`string`	string.
`sparse matrix`	sparse matrix.
`struct`	structure.

A structure is always denoted by `struct` followed by the structure type name.

If the symbol type is missing, the colon must not be present and the symbol type is assumed to be `proc`. Both library files below are valid:

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, you should combine related procedures in the same file in order to minimize the compiling of procedures not needed by your program. In other words, you should 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 as follows:

```
lib user myfile.src
```

This will update the user library by adding a reference to `myfile.src`.

No user library is shipped with **GAUSS**. It will be created the first time you use the lib command to update it.

For details on 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.

19.2 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 and to declare global N-dimensional arrays, sparse matrices, and structures, which will be initialized as follows:

Variable Type	Initializes To
N-dimensional array	1-dimensional array of 1 containing 0
sparse matrix	empty sparse matrix
structure	1x1 structure containing empty and/or zeroed out members

In order to declare structures in a .dec file, you must #include the file(s) containing the definitions of the types of structures that you wish to declare at the top of your .dec file. For example, if you have the following structure type definition in a file called mystruct.sdf:

```
struct mystruct {
   matrix m;
   array a;
   scalar scal;
   string array sa;
};
```

You could declare an instance of that structure type, called **ms**, in a .dec file as follows:

```
#include mystruct.sdf
declare struct mystruct ms;
```

See declare in the COMMAND REFERENCE for more information.

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. This allows you to reference global variables across source files in an application.

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 oper-
ations
** 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;
```

19.3 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 : '/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 /gauss/src/tstprt.g(2)

The symbol *prtvec* could not be found. Check if the file containing *prtvec* is in the *src_path*. You may have not activated the library that contains your symbol definition. Do so in a `library` statement.

```
/gauss/src/prt.dec(3) : Redefinition of '__vnames'
(proc)__vnames being declared external matrix
```

You are trying to illegally force a symbol to another type. You probably have a name conflict that needs to be resolved by renaming one of the symbols.

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

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

19.3.1 Using `.dec` Files

Below is some advice you are encouraged to follow when constructing your own library system:

- Whenever possible, declare variables in a file that contains only `declare` statements. When your program is run again without clearing the workspace, the file containing the variable declarations will not be compiled and `declare` warnings will be prevented.

- Provide a function containing regular assignment statements to reinitialize the global variables in your program if they ever need to be reinitialized 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 any 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.

20 The Library Tool

Libraries are collections of **GAUSS** procedures that are grouped together. The **GAUSS** Library Tool makes it easy for users to create and manage **GAUSS** libraries. The Library Tool is available on the Source Page and may be opened from the application menu **View->Library Tool**.

20.1 Creating New Libraries
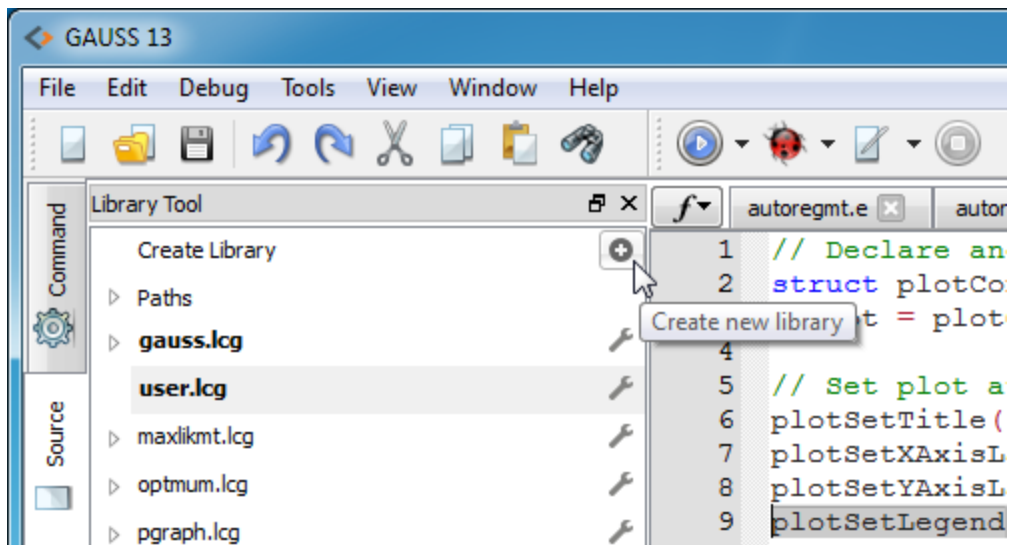
Figure 20.1: Create new library

At the top right corner of the Library Tool is a plus icon (+). Click this icon to create a new library. A file system window will open and ask you to enter a name for the library. Do not add a file extension as **GAUSS** will automatically add the correct file extension (`.lcg`).
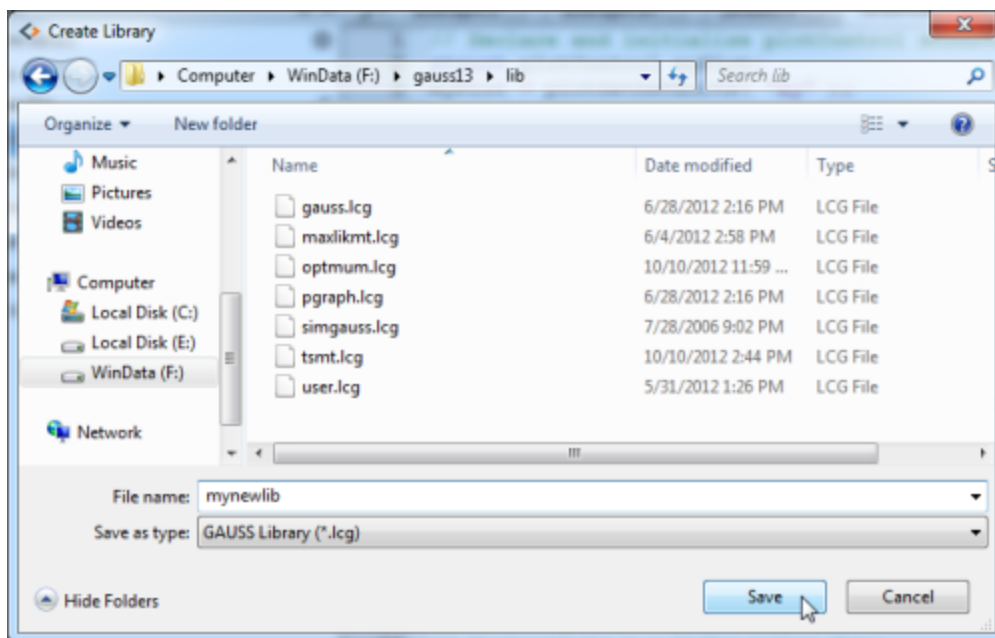
Figure 20.2: File browser

Once you have create a new empty library, you need to add some files. Click the wrench icon next to the library to which you would like to add a file and select 'Add Files' from the menu. This will open up a file browser window. Locate and select the file or files that you would like to add to the library.
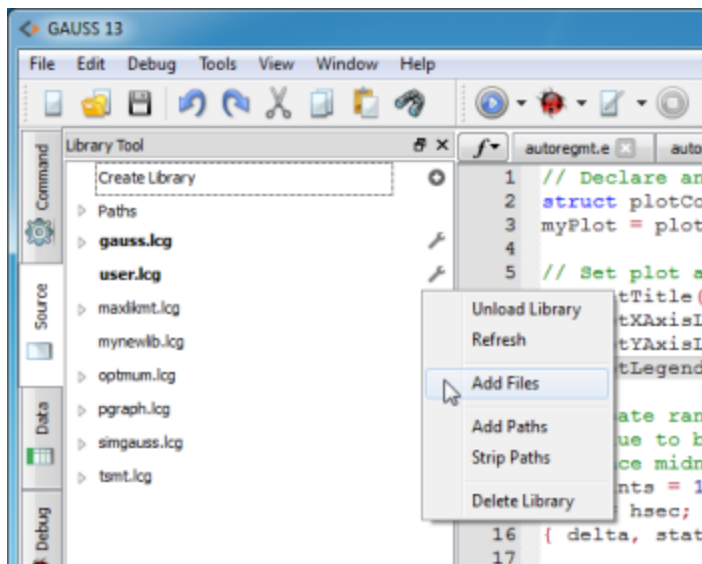
Figure 20.3: Add Files

20.2 Loading a Library

Loading a library makes all of its procedures available in **GAUSS**. It also allows you to navigate to the definition of a procedure by clicking on the name of a procedure and using the hotkey CTRL+F1, or selecting 'Open function definition' from the source editor's context menu.

To load the library, click the wrench icon and select 'Load Library' from the menu. The newly loaded library's name will be moved towards the top of the list of libraries and it will be bolded.

The library may be unloaded from the same menu accessible from the wrench icon.

20.3 Viewing Procedures

Expanding the node of any library in the tree will reveal a list of the included files. Double-clicking on one of the file names will open that file a source editor. Loaded libraries have an additional expandable node for each file. Expand this node to view each procedure stored in that file. Double-click the procedure name to open the file at the location of that procedure.
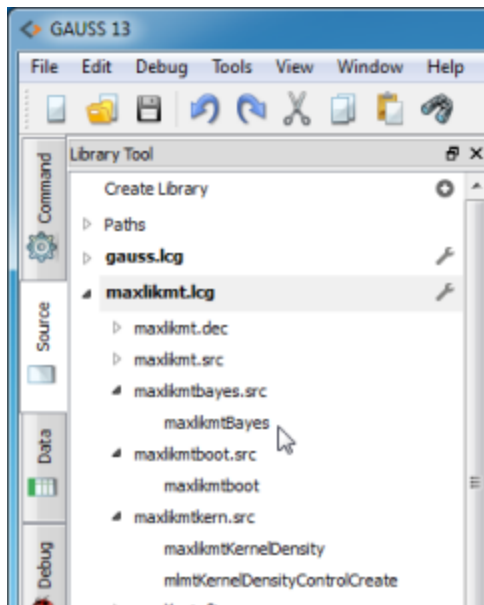
Figure 20.4: Library Tool Tree

20.4 Refreshing a Library

Changes made to files in a library such as adding procedures will be made available to **GAUSS**. However, to view and navigate correctly to new and changed library files

requires a library refresh. To perform this action, click on the corresponding wrench icon and select 'Refresh'.

21 Compiler

GAUSS allows you to compile your large, frequently used programs to a file that can be run over and over with no compile time. The compiled image is usually smaller than the uncompiled source. **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.

21.1 Compiling Programs

Programs are compiled with the `compile` command.

21.1.1 Compiling a File

Source code program files that can be run with the `run` command can be compiled to
`.gcg` files with the `compile` command:

```
compile qxy.e;
```

All procedures, global matrices, arrays, strings and string arrays, 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` com-
mand, 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 con-
figuration 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.

21.2 Saving the Current Workspace

The simplest way to create a compiled file containing a set of frequently used pro-
cedures is to use `saveall` and an `external` statement:

```
library pgraph;
external proc xy,logx,logy,loglog,hist;
saveall pgraph;
```

Just list the procedures you will be using in an `external` statement and follow it with
a `saveall` statement. It is not necessary to list procedures that you do not explicitly
call, but are called from another procedure, because the autoloader will automatically

Compiler

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.

21.3 Debugging

If you are using compiled code in a development situation in which 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.

22 Data Import/Export

GAUSS can read and write data in many different data formats such as:

- CSV and other delimited text files
- Undelimited text files
- Databases, such as MySQL, MS-SQL, Oracle and others
- Excel files
- GAUSS datasets
- GAUSS data archives
- GAUSS matrix files
- HDF5 files

For these file types, there are three main methods in which you can read data into GAUSS:

- Interactively, with the GAUSS Data Import Wizard.
- Programmatically, using a GAUSS function to read data directly into a GAUSS matrix or string array.
- Specifying a data file as a data source for a GAUSS procedure.

Concepts

- File schemas
    - CSV and delimited text file schemas
    - HDF5 file schemas
- File handles

- [CSV and delimited text](#)
- [GAUSS datasets](#)
- [GAUSS matrix files](#)
- [HDF5 files](#)

22.1 Data Import Wizard

Reading Text Files with the Data Import Dialog

The GAUSS Data Import Dialog can open text delimited files such as CSV or tab-separated files as well as Excel XLS and XLSX files. Select "File->Import Data" from the main GAUSS menu bar:
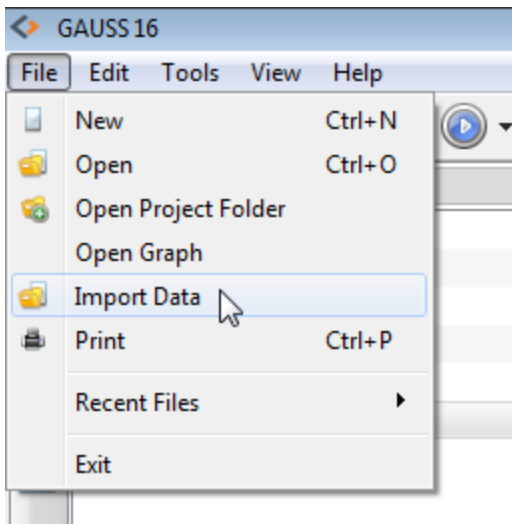
Or right-click in the Project Folder Window over a file you would like to import and select "Import data" from the context menu.
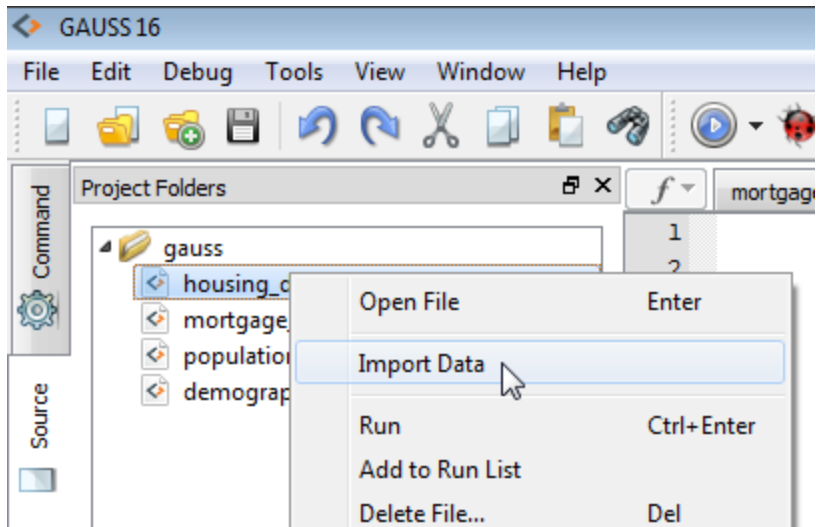
then locate the file that you would like to open and click the "Open" button.
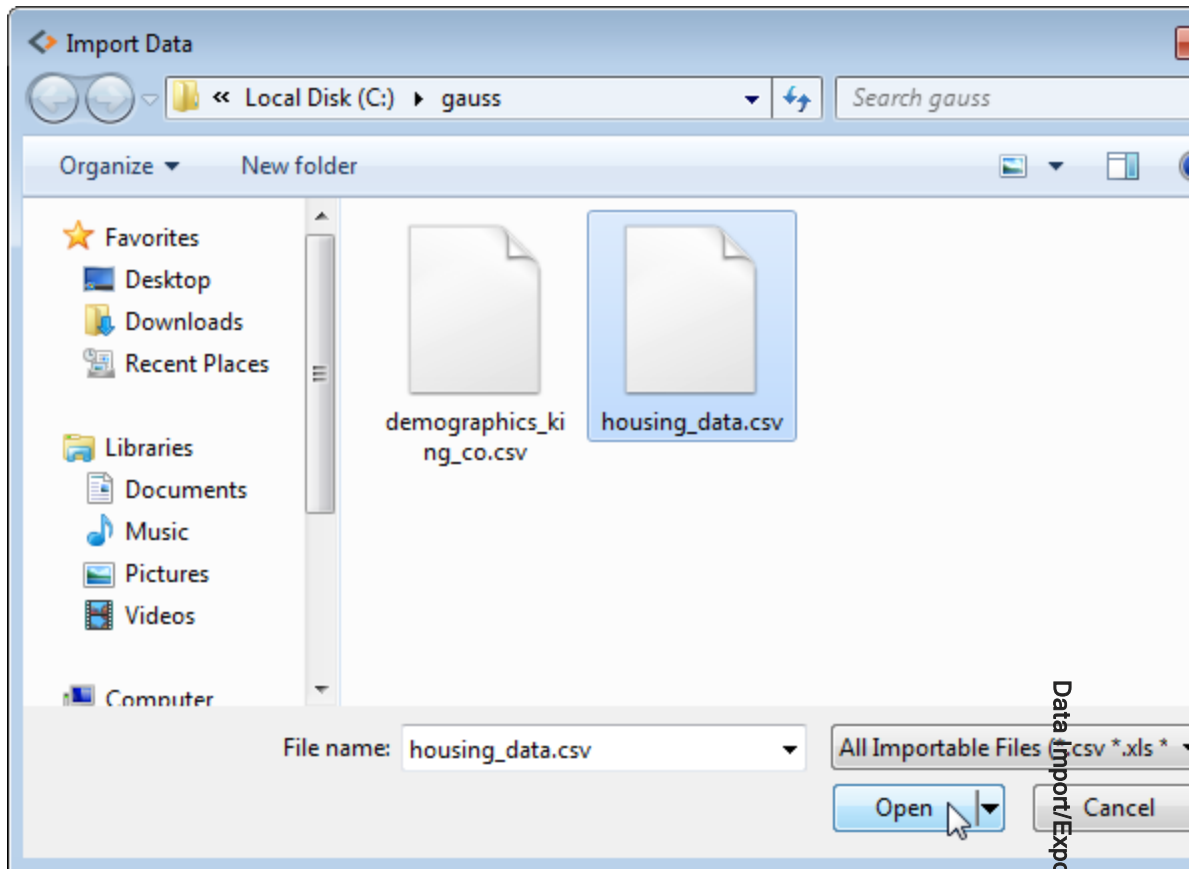
Data Import Dialog Overview

After you click the "Open" button, the Data Import Dialog Window will appear. The main area in the Data Import Dialog is a table with a preview of your data. This table contains the following elements:
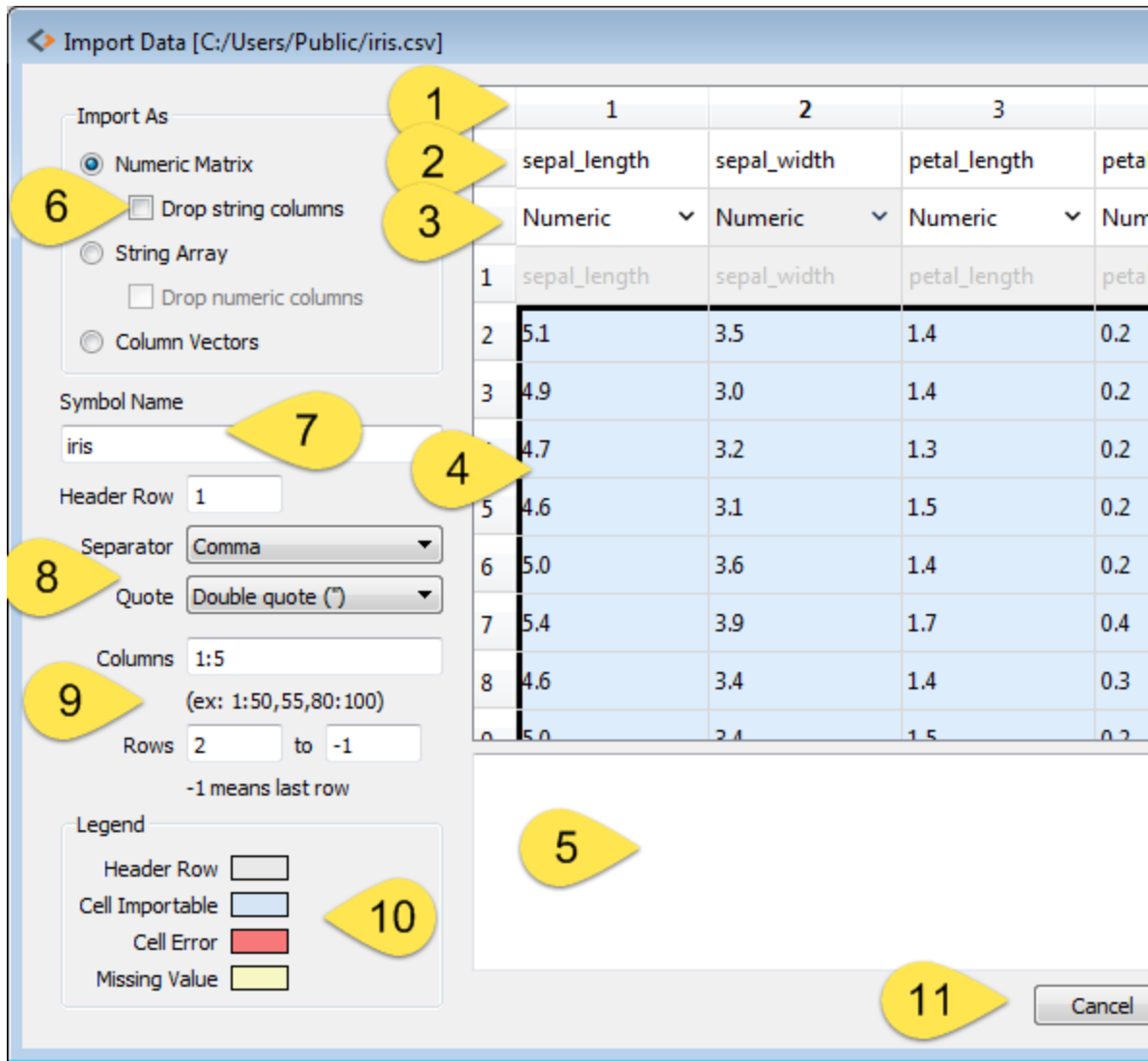
Figure 22.1: Data Import Dialog

1. **Column Index**

   A sequential index of all the columns in your data set.

2. **Column Header**

   If you select to import the data as separate column headers, each column will be imported as a separate GAUSS variable with it's column header as its name. After clicking "Import As 'Column Vectors'" (shown next to label 6) the column headers will be editable so that you can change their name prior to import. If the file does not contain file headers, GAUSS will insert X1, X2...XN.

3. **Column Type**

   A row containing the type of data for each column. These cells show the data type that GAUSS assumed each column to be. You may click in any of these cells to change the data type for the column. After doing so, the table will be updated to show the data when converted to the new type.

4. **Data Box**

   The first rows of your data. There will be a black border surrounding the cells that are selected for import. You can change the selected cells with the "Columns" and "Rows" text input boxes shown next to label 9.

   - Blue cells contain no problems and will be imported as seen in the preview.

   - Red Cells containing errors or unimportable content and will be imported as missing values.

   - Yellow background will be imported as missing values. Cells containing '.', 'NA' '?', or empty cells will be marked yellow. Also a cell that is selected to be imported as a "Numeric Matrix" (see label 10) that cannot be successfully converted to a double precision floating point value will be marked yellow.

5. **Message Window**

A window to display information about errors in your file or the result of your import.

6. **Desired Data Type**

The first section of the left side of the Data Import Dialog controls, with the title "Import As", controls the type of the data that is imported.
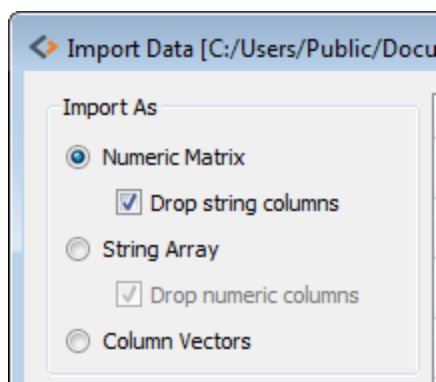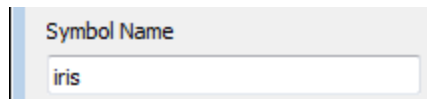
- If "Numeric Matrix" is selected, all selected data will be imported as a numeric matrix.

  - Any cell that cannot be converted to a double precision number will be imported as a missing value. These cells will be marked in yellow before import.

  - If "Drop string columns" is checked, any string columns inside the selected import range will be skipped over. These columns will be grayed out.

- If "String Array" is selected, all selected data will be imported as a string array.

- Any data, even numeric columns will be imported into a GAUSS string array.

  - If "Drop numeric columns" is checked, any numeric columns inside the selected import range will be skipped over. These columns will be grayed out.

- If "Column Vectors" is selected, each column will be imported separately. Numeric columns will be imported as numeric vectors and string columns will be imported as string vectors. Each vector will be a GAUSS global variable whose name will be the same as it's corresponding column header as shown in the Data Import Dialog.
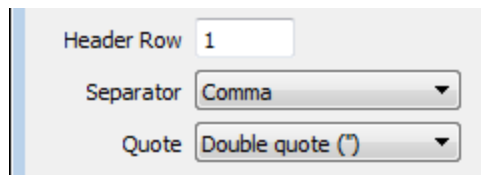
7. **Symbol Name**

You may change the text to any legal variable name in the input box for the name of the data matrix or string array created in your **GAUSS** workspace.

Symbol Name

iris
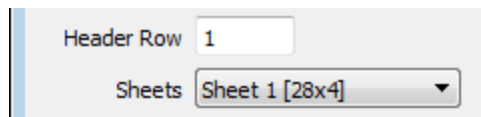
8. **Import control panel**

If you are importing a CSV file:

Header Row	1
Separator	Comma
Quote	Double quote (")

- The "Header Row" text input box specifies the row from which **GAUSS** will get the variable names to populate the "column headers" section of the import dialog (label 2). If the specified header row contains numeric data, the column headers will be filled with **X1, X2...**. The cells in the specified header row will have a gray background to mark them. If the header row is inside of the selected area, it will be imported. If you do not want this row imported make sure it is not included in your row range. Set "Header Row" to zero to indicate no header line.

- If the Import Dialog incorrectly specifies the field separator or quote character, you can change it here so that the file is imported correctly.

If you are importing an Excel XLS or XLSX file:

- The "Header Row" text input box specifies the row from which GAUSS will get the variable names to populate the "column headers" section of the import dialog (label 2). If the specified header row contains numeric data, the column headers will be filled with X1, X2.... The cells in the specified header row will have a gray background to mark them. If the header row is inside of the selected area, it will be imported. If you do not want this row imported make sure it is not included in your row range. Set "Header Row" to zero to indicate no header line.

- You can choose which sheet to import in the "sheets" option.

9. **Import Range**

Data Import/Export

The "Columns" and "Rows" text input boxes specify which rows and columns will be imported.

- For columns, you may specify a comma separated list of ranges or individual columns. For example 1:3 would specify columns 1, 2 and 3, whereas 1:3, 7:9, would specify columns 1,2,3,7,8,9. By default, all columns and all rows will be imported.

- The "Rows" input only accepts a starting row and ending row.

10. **Legend**

- "Header Row" indicates that the row is used to be as header in the preview window.

- "Cell Importable" indicates that the contents of a cell are importable as

shown in the preview window.

- "Cell Error" indicates that the field is invalid regardless of type (numeric or string).

- "Missing Value" indicates that the cell will be imported as a missing value. This may be because it is a specified missing value character (NA, ., ?) or because the data in the field is not valid for the import type (i.e. importing string data into a numeric matrix).

11. **Import Button**

- Click "Import and close", the desired data will be imported and Data Import Dialog will be closed.

- Click "Import", the desired data will be imported and Data Import Dialog will be open.

- Click "Cancel", Data Import Dialog will be closed.

22.2 Delimited Text Files

Delimited text files are files which contain data stored as human readable text separated by a character called the delimiter. Comma separated (CSV) files are one of the most common types of delimited text files. Other common delimiters include tabs, spaces and semi-colons.

Advantages	Disadvantages
Human readable.	Slow to read and write.
Easily portable to most platforms.	Does not contain meta-data (not self-describing).
	No formal file specification.
	Less space efficient for floating point data with high precision.

Reading and writing delimited text files

GAUSS can read and write delimited text files programmatically in chunks or all at once and can also read them interactively with the data import wizard. For more information and examples, regarding functions to read and write delimited text data, see the Command Reference entries for the functions in the table below:

csvReadM	Reads data from CSV file into a **GAUSS** matrix.
csvReadSA	Reads data from CSV file into a **GAUSS** string array.
csvWriteM	Write the contents of a **GAUSS** matrix to a CSV file.

Text Files as Data Sources

For convenience, some **GAUSS** functions, such as **dstatmt** for computing descriptive statistics, or **ols**, for computing least squares estimates, can accept the name of a delimited text file as a data source. As a trivial example, if you had a file named students.csv with the following contents:

```
Age,Grade
12,6
11,5
10,5
12,6
9,4
```

then the call:

```
//Calculate descriptive statistics on all variables in 'students.csv'
call dstatmt("students.csv")
```

will print out the following report:

```
----------------------------------------------------------
------------------------
Variable        Mean     Std
Dev      Variance   Minimum    Maximum    Valid   Missing
----------------------------------------------------------
------------------------

Age            10.8000    1.3038      1.7000     9.0000    12-
.0000        5         0

Grade           5.2000    0.8367      0.7000     4.0000     6-
.0000        5         0
```

Delimited text files do not always follow formal standards or contain meta-data to describe their internal layout. The GAUSS functions, csvReadM and csvReadSA provide the ability for users to specify the delimiter and skip a header row in a simple manner. However, for other functions that already have many other inputs to specify the computation, it is often too cumbersome to add other inputs for full data source specification. For this reason, to use a delimited text file as a data source input:

- The delimiter must be a comma.
- The file name must end in .csv OR a file schema must be added to the beginning of the file name.
- The first row will be assumed to contain the headers (or variable names).

File schemas

A file schema is made up of 2 or 3 letters followed by ".//"" at the start of the file name that allow you to give GAUSS necessary information in a compact manner. The schema to tell GAUSS to treat a file as CSV is: "csv://". For example, we could calculate descriptive statistics on all variables in the CSV file 'binary.csv' (located in your GAUSS examples directory), like this:

```
//No schema required if file ends in .csv
call dstatmt("binary.csv");
```

However, if our file had a different extension, such as 'binary.txt', we would need to use a schema, like this:

```
//Add schema to tell GAUSS that
//'binary.txt' contains CSV data
call dstatmt("csv://binary.txt");
```

With example full path for Windows:

```
call dstatmt("csv://C:/Users/MyUserName/Data/binary.txt");
```

With example full path for Mac:

```
call dstatmt("csv:///Users/MyUserName/Data/binary.txt");
```

With example full path for Linux:

```
call dstatmt("csv:///home/myusername/data/binary.txt");
```

The following is a partial list of functions that can accept a CSV file as a data source: dstatmt, glm, momentd, ols, quantiled.

File handles

A file handle is a reference to a data structure that allows you to keep a file open for multiple read and/or write operations and to move the cursor to specific locations in the file. File handles are less commonly used with delimited text files than **csvReadM/csvReadSA** or using the file as a data source. The following functions can be used with delimited text files:

close	Closes a GAUSS file.
fopen	Opens a file.
fgets	Reads a line of text from a file.
fputs	Writes strings to a file.
fputst	Writes strings to a file.
fseek	Positions the file pointer in a file.
fsterror	Returns an error message explaining the cause of the

most recent file I/O error.

ftell   Gets the position of the file pointer in a file.

For complete examples, see the Command Reference page for fputs.

22.3 Spreadsheets

Spreadsheets are popular choice for storing data in tabular form. **GAUSS** can read and write Microsoft Excel .XLS and .XLSX files on Windows, Mac and Linux.

Advantages	Disadvantages
Easily portable to most platforms.	Slow to read and write.
Widely used.	Incomplete meta-data does not allow for robust determination of the location of data vs. header names vs. comments.
Formal file specification.	Prone to cell formatting errors, such as extra columns coded as empty, which will not be shown when viewed inside of Excel.
Contains some meta-data about the type of data in individual cells.	Memory inefficient when trying to read small chunks of data from large files.

Reading and writing spreadsheet files

GAUSS can read Excel spreadsheet files programmatically in chunks or the entire file at once. GAUSS can also read them interactively with the data import wizard. For more information and examples see:

Data Import/Export

xlsGetSheetCount	Gets the number of sheets in an Excel® spreadsheet.
xlsGetSheetSize	Gets the size (rows and columns) of a specified sheet in an Excel® spreadsheet.
xlsGetSheetTypes	Gets the cell format types of a row in an Excel® spreadsheet.
xlsMakeRange	Builds an Excel® range string from a row/column pair.
xlsReadM	Reads from an Excel® spreadsheet into a **GAUSS** matrix.
xlsReadSA	Reads from an Excel® spreadsheet into a **GAUSS** string array or string.
xlsWrite	Writes a **GAUSS** matrix, string, or string array to an Excel® spreadsheet.
xlsWriteM	Writes a **GAUSS** matrix to an Excel® spreadsheet.
xlsWriteSA	Writes a **GAUSS** string or string array to an Excel® spreadsheet.

Excel files as data sources

For convenience, some **GAUSS** functions, such as **dstatmt** for computing descriptive statistics, or **ols**, for computing least squares estimates, can accept the name of an Excel file as a data source. As a trivial example, if you had a file named stu-dents.xlsx with the following contents:

Age	Grade
12	6
11	5
10	5
12	6
9	4

then the call:

```
//Calculate descriptive statistics on all variables in 'stu-
dents.xlsx'
call dstatmt("students.xlsx")
```

will print out the following report:

```
-----------------------------------------------------------
------------------------
Variable       Mean      Std
Dev      Variance   Minimum    Maximum     Valid    Missing
-----------------------------------------------------------
------------------------

Age          10.8000      1.3038      1.7000      9.0000     12-
.0000         5          0

Grade         5.2000      0.8367      0.7000      4.0000      6-
.0000         5          0
```

Since Excel files do not completely describe which cells contain data, or headers or comments, to use an XLS or XLSX file as a datasource input:

- The data must be on the first sheet.
- The first row will be assumed to contain the headers (variable names).
- All data must be read in.

File schemas

A file schema is made up of 2 or 3 letters followed by "://" at the start of the file name of a file source that allows you to give **GAUSS** some necessary information in a compact manner.

File schemas are not required in any cases for spreadsheet files in this **GAUSS** revision.

File handles

A file handle is a reference to a data structure that allows you to keep a file open for multiple read and/or write operations and to move the cursor to specific locations in the file.

There are no functions for creating or using file handles with spreadsheet files in this **GAUSS** revision.

22.4 HDF5 Files

The Hierarchical Data Format (HDF) is a data format specifically designed for scientific data. It was originally developed by the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign.

It has been used extensively at organizations such as NASA, who need data storage that is cross platform, high-performance, capable of handling very large datasets and can support complex data types.

HDF5 is the current version of this data format.

Advantages	Disadvantages
Easily portable to most platforms.	Can be more complicated to create for optimal performance.
Self-describing data format.	No official specification for including header names or specifying categorical variables.
No file size limit.	
Allows creation of multiple independent datasets in the same file.	
Fast read and write.	
Efficient data storage.	
Supports many data types, including integer types, floating point, strings	

and multi-dimensional arrays.

Allows addition of custom 'Attributes'
to datasets.

HDF5 File Layout

HDF5 files can contain one or more datasets and 'groups'. A 'group' in an HDF5 file is analogous to a folder in your computer's file system and an HDF5 dataset is analogous to a file.

At its most basic, an HDF5 file would have just one dataset at the root node of the HDF5 file. The root node is represented by a forward slash just like the file systems on Mac and Linux. For example, a file named `students.h5` with one dataset named, `math_scores` located at the root node:

File	Root Node	Dataset
students.h5	/	math_scores

You can add any number of datasets at the root node, for example, if you added a dataset to the root node named 'language_scores', the file layout would look like this:

File	Root Node	Datasets
students.h5	/	math_scores
		language_scores

Finally, the file could be further organized by adding a group to the root node named, 'academic' and placing both datasets from above in this group:

File	Root Node	Group	Datasets
students.h5	/	academic/	math_scores
			language_scores

Dataset Attributes

In an HDF5 file an 'Attribute' is like a dataset that is attached to another dataset. Attributes are usually used to provide information about the dataset. Attributes:

- Must be attached to a dataset.
- May be any supported datatype.
- Are read and written in an 'all-or-none' manner--no partial reading or writing.

When an HDF5 file is specified as a dataset, **GAUSS** will expect the variable names to be in an attribute named 'headers' attached to the specified dataset. If the dataset does not have an attached attribute named 'headers', then **GAUSS** will use 'X1, X2, X3...XP', where 'P' is the number of variables (columns) in the file.

See h5readAttribute and h5writeAttribute for more information and examples.

Related Functions

GAUSS has several functions for reading and writing of HDF5 files:

h5create	Create a HDF5 dataset (`.h5`). HDF5 is a data model, library, and file format for storing and managing data.
h5open	Open a HDF5 file. HDF5 is a data model, library, and file format for storing and managing data.
h5read	Reads data from a HDF5 file (`.h5`) into a **GAUSS** matrix. HDF5 is a data model, library, and file format for storing and managing data.
h5readAttribute	Read attributes from a HDF5 file into **GAUSS** .
h5write	Write a **GAUSS** matrix to a HDF5 file. HDF5 is a data model, library,

	and file format for storing and managing data.
h5WriteAttribute	Write a **GAUSS** string array as attributes into a HDF5 file.
dataopen	Opens a data set.
readr	Reads a specified number of rows of data from a **GAUSS** data set (`.dat`) file or a **GAUSS** matrix (`.fmt`) file.
writer	Writes a matrix to a **GAUSS** data set.
close	Closes a **GAUSS** file.

HDF5 files as data sources

For convenience, some **GAUSS** functions, such as **dstatmt** for computing descriptive statistics, or **ols**, for computing least squares estimates, can accept the name of an HDF5 file and dataset as a data source input. These **GAUSS** functions will expect the variable names to be located in an HDF5 'dataset attribute' named 'headers'. See h5writeAttribute for instructions on creating HDF5 dataset attributes. HDF5 files must be used with a file schema when specified as a data source.

File schemas

A file schema is made up of 2 or 3 letters followed by "://" at the start of the file name of a file source that allows you to give **GAUSS** some necessary information in a compact manner.

Since there may be more than one dataset in an HDF5 and they may be located inside of an arbitrary number of groups, an HDF5 data source must contain:

1. The file schema h5://
2. The file name with or without a path
3. The full path to the dataset WITHIN the file.

For example, to calculate descriptive statistics on all variables in a dataset named 'math_scores', located at the root node of a file named 'students.h5' you would enter:

```
call dstatmt("h5://students.h5/math_scores");
```

With example full path for Windows:

```
call dstatmt("h5://C:/Users/MyUserName/Data/students.h5/math_
scores");
```

With example full path for Mac:

```
call dstatmt("h5:///Users/MyUserName/Data/students.h5/math_
scores");
```

With example full path for Linux:

```
call dstatmt("h5:///home/myusername/data/students.h5/math_
scores");
```

To perform the same calculation on a dataset named 'language_scores' which resides in a 'group' located at the root node named 'academic', the correct string would be:

```
call dstatmt("h5://students.h5/academic/language_scores);
```

File handles

A file handle is a reference to a data structure that allows you to keep a file open for multiple read and/or write operations and to move the cursor to specific locations in the file.

The following functions can work with HDF5 file handles:

dataopen	Opens a data set.
readr	Reads a specified number of rows of data from a **GAUSS** data set (`.dat`) file or a **GAUSS** matrix (`.fmt`) file.

writer	Writes a matrix to a **GAUSS** data set.
close	Closes a **GAUSS** file.
h5open	Open a HDF5 file. HDF5 is a data model, library, and file format for storing and managing data.
getnamef	Reads the variable names from a dataset. (Stored in an HDF5 dataset attribute named 'headers')..

22.5 GAUSS Datasets

GAUSS Datasets (`.dat`) can store a **GAUSS** matrix along with variable names and keep track of which columns are categorical variables.

Advantages	Disadvantages
Fast read and write.	Categorical labels cannot have more than 8 characters.
Can be read all at once, or in specified chunks of rows.	
Can store header names and specify categorical variables.	
Rows may be added after file creation.	

Below is a list of the most commonly used functions for interacting with datasets.

loadd	Loads a data set.
saved	Writes a matrix in memory to a **GAUSS** data set on disk.
datacreate	Creates a real data set.
dataopen	Opens a data set.
readr	Reads a specified number of rows of data from a

	GAUSS data set (`.dat`) file or a **GAUSS** matrix (`.fmt`) file.
writer	Writes a matrix to a **GAUSS** data set.
close	Closes a **GAUSS** file.
getnamef	Returns a string array containing the names of the variables in a **GAUSS** data set.
vartypef	Returns a vector of ones and zeros that indicate whether variables in a data set are character or numeric.

GAUSS Matrix Files as Data Sources

For convenience, some **GAUSS** functions, such as `dstatmt` for computing descriptive statistics, or `ols`, for computing least squares estimates, can accept the name of a file as the data source for computations. **GAUSS** Datasets (`.dat`) may be used as a data source input.

File schemas

GAUSS Datasets (`.dat`) do not need to use file schemas, since they contain meta-data to fully describe their contents.

File handles

A file handle is a reference to a data structure that allows you to keep a file open for multiple read and/or write operations and to move the cursor to specific locations in the file.

The following functions can work with **GAUSS** Dataset file handles:

dataopen	Opens a data set.
readr	Reads a specified number of rows of data from a **GAUSS** data set (`.dat`) file or a **GAUSS** matrix (`.fmt`) file.

writer	Writes a matrix to a **GAUSS** data set.
close	Closes a **GAUSS** file.
colsf	Returns the number of columns in a **GAUSS** data (.dat) file or **GAUSS** matrix (.fmt) file.
rowsf	Returns the number of rows in a **GAUSS** data set (.dat) file or **GAUSS** matrix (.fmt) file.
getnamef	Returns a string array containing the names of the variables in a **GAUSS** data set.
vartypef	Returns a vector of ones and zeros that indicate whether variables in a data set are character or numeric.

22.6 GAUSS Matrix Files

GAUSS matrix files (.fmt) are a fast and convenient file type for storage of **GAUSS** matrices.

Advantages	Disadvantages
Easy to use with save and load.	Cannot store header names*.
Fast read and write.	Does not store meta-data to specify categorical variables.
Can be read all at once, or in specified chunks of rows.	Not human readable.
	Cannot add or delete rows after initial file creation.

- When used as a data source, .fmt files will return variable names X1, X2, X3…XP

save	Saves matrices, strings, or procedures to a disk file.
load	Loads from a disk file.

Data Import/Export

dataopen	Opens a data set.
readr	Reads a specified number of rows of data from a **GAUSS** data set (`.dat`) file or a **GAUSS** matrix (`.fmt`) file.
writer	Writes a matrix to a **GAUSS** data set.
seekr	Moves the pointer in a `.dat` or `.fmt` file to a particular row.
close	Closes a **GAUSS** file.

GAUSS Matrix Files as Data Sources

For convenience, some **GAUSS** functions, such as **dstatmt** for computing descriptive statistics, or **ols**, for computing least squares estimates, can accept the name of a file as the data source for computations. **GAUSS** Matrix Files (`.fmt`) may be used as a data source input. Since **GAUSS** Matrix Files (`.fmt`) do not contain header information (variable names), use X1, X2, X3…XP as the variable names.

File handles

A file handle is a reference to a data structure that allows you to keep a file open for multiple read and/or write operations and to move the cursor to specific rows in the file. This is useful when only a subset of rows are needed at one time.

The following functions can work with FMT file handles:

h5open	Open a HDF5 file. HDF5 is a data model, library, and file format for storing and managing data.
dataopen	Opens a data set.
readr	Reads a specified number of rows of data from a **GAUSS** data set (`.dat`) file or a **GAUSS** matrix (`.fmt`) file.
writer	Writes a matrix to a **GAUSS** data set.
close	Closes a GAUSS file.

colsf	Returns the number of columns in a **GAUSS** data (`.dat`) file or **GAUSS** matrix (`.fmt`) file.
rowsf	Returns the number of rows in a **GAUSS** data set (`.dat`) file or **GAUSS** matrix (`.fmt`) file.
getnamef	Returns a string array containing the names of the variables in a **GAUSS** data set.
vartypef	Returns a vector of ones and zeros that indicate whether variables in a data set are character or numeric.

22.7 GAUSS Data Archives

GAUSS Data Archives (`.gda`) provide the ability to store an unlimited number of GAUSS variables. They may hold any **GAUSS** data type, including matrix, sparse matrix, n-dimensional array, string or string array.

Advantages	Disadvantages
Can hold any **GAUSS** data type.	Cannot store header names*.
Fast read and write.	Does not store meta-data to specify categorical variables.
Unlimited file size.	Not human readable.
Can be read all at once, or in specified chunks of rows.	
Can hold more than one **GAUSS** variable (matrix, array, etc).	
Can be read and written on any platform.	

The main difference between a **GAUSS** Data Archive and a **GAUSS** Dataset is that a **GAUSS** Dataset will hold one matrix and variable names for each column of the matrix. **GAUSS** Data Archives, however, hold one or more **GAUSS** symbols (matrix, sparse matrix, structure, etc) and a name for each of these symbols.

```
//Create a GDA, overwrite the file
//if a file by this name exists
call gdaCreate("sample.gda", 1);
//Create a matrix
a = { 1.2 3.4,
      0.9 0.8,
      1.8 2.3 };

//Save the GAUSS matrix
//'a' to the GDA
//with the name 'sample_a'
call gdaWrite("sample.gda", a, "Sample A");

//Create a matrix
b = { 4.0 1.2,
      1.6 4.3,
      2.7 3.3 };

//Save the GAUSS matrix
//'b' to the GDA
//with the name 'Sample B'
call gdaWrite("sample.gda", b, "Sample B");

//Read in 'Sample A'
new_a = gdaRead("sample.gda", "Sample A");

//Read in 'Sample B'
new_b = gdaRead("sample.gda", "Sample B");
```

Notice in the above example that each 3x2 matrix had a name, but the columns of the matrices did not have a name. We can, of course, add a string array variable containing column (model variable) names. For example, let's suppose that the columns from 'Sample A' and 'Sample B' both represent measurements of the same quantity. We could create a string variable in the GDA file, containing the names of these measurements.

```
//Create a 2x1 string array
names = "height" $| "weight";

//Write the string data from 'names'
//to the GDA and call it 'headers'
call gdaWrite("sample.gda", names, "headers");
```

Then it is a simple matter to read in these variable names (headers) with '**gdaRead**', when required—maybe for printing a report.

```
//Read in data from variable 'headers' in 'sample.gda'
var_names = gdaRead("sample.gda", "headers");
```

The following functions are for use with **GAUSS** Data Archives:

gdaCreate	Creates a **GAUSS** Data Archive.
gdaDstat	Computes descriptive statistics on multiple Nx1 variables in a **GAUSS** Data Archive.
gdaGetIndex	Gets the index of a variable in a **GAUSS** Data Archive.
gdaGetName	Gets the name of a variable in a **GAUSS** Data Archive.
gdaGetNames	Gets the names of all the variables in a **GAUSS** Data Archive.
gdaGetOrders	Gets the orders of a variable in a **GAUSS** Data Archive.
gdaGetType	Gets the type of a variable in a **GAUSS** Data Archive.
gdaGetTypes	Gets the types of all the variables in a **GAUSS** Data Archive.
gdaGetVarInfo	Gets information about all of the variables in a**GAUSS** Data Archive and returns it in an array of **gdavartable** structures.
gdaLoad	Loads variables in a GDA into the workspace.
gdaPack	Packs the data in a **GAUSS** Data Archive, removing all empty bytes and truncating the file.

gdaRead	Gets a variable from a **GAUSS** Data Archive.
gdaReadByIndex	Gets a variable from a **GAUSS** Data Archive given a variable index.
gdaReadSome	Reads part of a variable from a **GAUSS** Data Archive.
gdaReadSparse	Gets a sparse matrix from a **GAUSS** Data Archive.
gdaReadStruct	Gets a structure from a **GAUSS** Data Archive.
gdaSave	Writes variables in a workspace to a GDA.
gdaUpdate	Updates a variable in a **GAUSS** Data Archive.
gdaWrite	Writes a variable to a **GAUSS** Data Archive.
gdaWriteSome	Overwrites part of a variable in a **GAUSS** Data Archive.

GAUSS Data Archives as Data Sources

GAUSS Data Archives are not yet supported as data sources for **GAUSS** functions.

22.8 File Format Specifications

This section discusses the **GAUSS** binary file formats.

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.

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.

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.

Data Import/Export

22.8.1 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:

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

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:

$$
\begin{array}{lcccccc}
(real\,part) & [1,1] & [1,2] & [1,3] & [2,1] & [2,2] & [2,3] \\
(imaginary\,part) & [1,1] & [1,2] & [1,3] & [2,1] & [2,2] & [2,3]
\end{array}
$$

22.8.2 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.

22.8.3 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

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.

22.8.4 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.

22.8.5 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:

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

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:

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

22.8.6 Extended Data Set v89 (Obsolete)

Data sets with more than 8175 columns are saved in an extended format that cannot be read by the 16-bit version. 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.

22.8.7 Matrix *v92* (Obsolete)

Offset	Description
0-3	always 0
4-7	always 0xEECDCDCD
8-11	reserved
12-15	reserved
16-19	reserved
20-23	0 - real matrix, 1 - complex matrix
24-27	number of dimensions 0 - scalar 1 - row vector 2 - column vector, matrix
28-31	header size, 128 + number of dimensions * 4, padded to 8-byte boundary
32-127	reserved

If the data is a scalar, the data will directly follow the header.

If the data is a row vector, an unsigned integer equaling the number of columns in the vector will precede the data, along with 4 padding bytes.

If the data is a column vector or a matrix, there will be two unsigned integers preceding the data. The first will represent the number of rows in the matrix and the second will represent the number of columns.

Data Import/Export

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.

22.8.8 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.

22.8.9 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.

22.8.10 Matrix *v96*

Offset	Description
0-3	always 0xFFFFFFFF
4-7	always 0
8-11	always 0xFFFFFFFF
12-15	always 0
16-19	always 0xFFFFFFFF
20-23	0xFFFFFFFF for forward byte order, 0 for backward byte order
24-27	0xFFFFFFFF for forward bit order, 0 for backward bit order
28-31	always 0xABCDEF01
32-35	currently 1

36-39	reserved
40-43	floating point type, 1 for IEEE 754
44-47	1008 (double precision data)
48-51	8, the size in bytes of a double matrix
52-55	0 - 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	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.

22.8.11 Data Set *v96*

Offset	Description
0-3	always 0xFFFFFFFF
4-7	always 0
8-11	always 0xFFFFFFFF
12-15	always 0
16-19	always 0xFFFFFFFF
20-23	0xFFFFFFFF for forward byte order, 0 for backward byte order
24-27	0xFFFFFFFF for forward bit order, 0 for backward bit order
28-31	0xABCDEF02
32-35	version, currently 1
36-39	reserved
40-43	floating point type, 1 for IEEE 754
44-47	12 - 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
64-67	1 for row major ordering of elements, 2 for column major
68-71	always 0

Data Import/Export

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.

22.8.12 GAUSS Data Archives

A **GAUSS** Data Archive consists of a header, followed by the variable data and, finally, an array of variable descriptors containing information about each variable.

Header

The header for a **GAUSS** Data Archive is laid out as follows:

Offset	Type	Description
0-3	32-bit unsigned integer	always 0xFFFFFFFF
4-7	32-bit unsigned integer	always 0
8-11	32-bit unsigned integer	always 0xFFFFFFFF
16-19	32-bit unsigned integer	always 0xFFFFFFFF
20-23	32-bit unsigned integer	0xFFFFFFFF for forward byte

		order, 0 for backward byte order
24-27	32-bit unsigned integer	always 0
28-31	32-bit unsigned integer	always 0xABCDEF08
32-35	32-bit unsigned integer	version, currently 1
36-39	32-bit unsigned integer	reserved
40-43	32-bit unsigned integer	floating point type, 1 for IEEE 754
44-55	32-bit unsigned integers	reserved
56-63	64-bit unsigned integer	number of variables
64-67	32-bit unsigned integer	header size, 128
68-95	32-bit unsigned integers	reserved
96-103	64-bit unsigned integer	offset of variable descriptor table from end of header
104-127	64-bit unsigned integers	reserved

Variable Data

After the header comes the variable data. Matrices are laid out in row-major order, and strings are written with a null-terminating byte.

For string arrays, an array of *rows*x *columns* struct **satable**'s is written out first, followed by the string array data in row-major order with each element null terminated. A struct **satable** consists of two members:

Member	Type	Description
off	*size_t*	offset of element data from beginning of string array data
len	*size_t*	length of element data, including null-terminating byte

On a 32-bit machine, a *size_t* is 4 bytes. On a 64-bit machine, it is 8 bytes.

Arrays are written with the orders (sizes) of each dimension followed by the array data. For example, the following 2x3x4 array:

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

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{matrix}$$

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

$$\begin{matrix} 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{matrix}$$

would be written out like this:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Variable Structures

The variable data is followed by an array of variable descriptors. For each variable in the GDA, there is a corresponding variable descriptor in this array. A variable descriptor is laid out as follows:

Offset	Type	Description
0-3	32-bit unsigned integer	variable type
4-7	32-bit unsigned integer	data type, 10 for 8 byte floating point
8-11	32-bit unsigned integer	dimensions, used only for arrays

12-15	32-bit unsigned integer	complex flag, 1 for real data, 0 for complex
16-19	32-bit unsigned integer	size of pointer, indicates whether the variable was written on a 32-bit or 64-bit platform
20-23	32-bit unsigned integer	huge flag, indicates whether the variable is larger than INT_ MAX
24-31	64-bit unsigned integer	rows for matrices and string arrays
32-39	64-bit unsigned integer	columns for matrices and string arrays, length for strings, including null-terminating byte
40-47	64-bit unsigned integer	index of the variable in the GDA
48-55	64-bit unsigned integer	offset of variable data from end of header
56-63	64-bit unsigned integer	length of variable data in bytes
64-143	string	name of variable, null-terminated

The variable type (bytes 0-3) may be any of the following:

20	array
30	matrix
40	string
50	string array

The size of pointer element (bytes 16-19) is the size of a pointer on the machine on which the variable was written to the GDA. It will be set to 4 on 32-bit machines and 8

on 64-bit machines. This element is used only for string array variables. If a GDA containing string arrays is created on a 32-bit machine and then read on a 64-bit machine, or vice versa, then the size of pointer element indicates how the members of the struct **satable**'s must be converted in order to be read on the current machine.

The huge flag (bytes 20-23) is set to 1 if the variable size is greater than INT_MAX, which is defined as 2147483647. A variable for which the huge flag is set to 1 may not be read into **GAUSS** on a 32-bit machine.

The variable index element (bytes 40-47) contains the index of the variable in the GDA. Although the variable data is not necessarily ordered by index (see **gdaUpdate**), the variable descriptors are. Therefore, the indices are always in ascending order.

23 Databases with GAUSS

The database functionality within **GAUSS** is designed with simplicity in mind. This chapter assumes a basic knowledge of SQL. You should be able to understand simple SELECT, INSERT, UPDATE, and DELETE statements.

23.1 Connecting to Databases

In order to operate on a database, a connection must first be created and then opened. Database connections are created using the **dbAddDatabase** method and identified by the id number returned. It is possible to have multiple connections to the same database.

Note that the creation of a connection is different from opening the connection. The database can not be accessed until the connection is opened after it is created. The following example shows how to create a connection and open it.

```
id = dbAddDatabase("MYSQL");
dbSetHostName(id, "sql.mycompany.net");
dbSetDatabaseName(id, "testdb");
dbSetUserName(id, "webuser");
dbSetPassword(id, "f2DAf3E");
ret = dbOpen(id);
```

The first line returns an identifier for a MYSQL connection, and the last line opens the actual connection. Between those actions we set up the connection properties, such as the host name, database name, username and password. There is an additional short-hand method available to initialize the properties, described in the help page for **dbAddDatabase**.

It is also possible to omit the **dbSetUserName** and **dbSetPassword** calls and specify both in the **dbOpen** call, as follows:

```
ret = dbOpen(id, "webuser", "f2DAf3E");
```

The following example shows how to specify all connection properties in a single string, as well as how to connect to the same database twice using the same properties.

```
url =
"mysql://webuser:f2DAf3E@sql.mycompany.net:3306/testdb";

id1 = dbAddDatabase(url);
id2 = dbAddDatabase(url);
```

id1 and *id2* are now separate connections to the same database. Since we've encapsulated all of the connection properties in the 'url' string, we can immediately call **dbOpen** on either *id1* or *id2*.

If **dbOpen** fails for any reason, it should be visible in the error output view. You can also call **dbGetLastErrorNum** or **dbGetLastErrorText** and pass in the appropriate id to get that information programmatically in **GAUSS**.

Once the connection has been established, it can be removed and all resources freed by calling **dbRemoveDatabase** with the corresponding connection id. If you'd like to only temporarily close the connection, you can also call **dbClose**, which allows for subsequent calls to **dbOpen** to re-open the connection.

23.2 Executing SQL Statements

GAUSS provides an intuitive interface for executing SQL statements and retrieving data either all at once or iteratively.

23.2.1 Creating a Query

A query can be created through two available methods. The first is direct-execution using the **dbExecQuery** method, assuming 'id' is a valid connection id:

```
qid = dbExecQuery(id, "SELECT symbol,
    price FROM stocks");
```

If you wish for your query to be more selective about the results being returned and the condition is user supplied, parameter binding is supported by **GAUSS** and easy to use:

```
print  "Enter stock symbol: ";;
symbol = cons();
qid = dbExecQuery(id, "SELECT symbol,
    price FROM stocks WHERE symbol = ?",
    symbol);
```

The above example will immediately execute the query. If you'd like to set up the query more carefully, you can prepare it first and execute at your discretion.

```
qid = dbCreateQuery(id, "SELECT symbol,
    price FROM stocks WHERE symbol = :sym");
print "Enter stock symbol: ";;
symbol = cons();
dbQueryBindValue(qid, ":sym", symbol);
dbQueryExecPrepared(qid);
```

Note the specific call to **dbQueryExecPrepared** that is required when using **dbCreateQuery** instead of **dbExecQuery**. **dbQueryExecPrepared** can also be used for re-execution of a query created using **dbExecQuery**.

If the query fails to execute, output should be displayed in the error view but can also be fetched programatically using either **dbQueryGetLastErrorNum** or **dbQueryGetLastErrorText**.

23.2.2 Fetching Query Results

Once the query has been executed, retrieving the result is trivial.

To fetch all results directly into a matrix:

```
results = dbQueryFetchAllM(qid);
```

To store the symbol names into a string array and the price into a matrix:

```
symbols = dbQueryFetchAllSA(qid, "symbol");
prices = dbQueryFetchAllM(qid, "price");
```

GAUSS also has methods to provide access to the results one row at a time. Following the execution of a query, its internal pointer is located one position before the first record. An initial call to **dbQuerySeekNext** is required to position on the first row. Additional calls to **dbQuerySeekNext** will increment the current row, until the function returns false and no more rows are available.

A typical iteration of the results is as follows:

```
do while
    dbQuerySeekNext(qid);
    symbol = dbQueryFetchOneSA(qid, "symbol");
    price = dbQueryFetchOneM(qid, "price");
endo;
```

Multiple functions are available for flexible navigation of the result set. You can iterate forward and backward with **dbQuerySeekNext** and **dbQuerySeekPrevious**. **dbQuerySeekFirst** and **dbQuerySeekLast** will navigate to the first and last record, respectively. **dbQueryGetPosition** returns the current row index. The total number of rows can be retrieved via the **dbQueryRows** method.

Note that a performance increase can be achieved if you are only using **dbQuerySeekNext**, and never navigating backwards. This is most noticeable on large result sets.

In order to take advantage of the performance improvement, the driver must support it, and you must call **dbQuerySetForwardOnly** before executing the query, as such:

```
qid = dbCreateQuery("SELECT * FROM stocks");
dbQuerySetForwardOnly(qid);
dbQueryExecPrepared(qid);
```

23.2.3 Inserting, Updating, and Deleting Records

While SELECT statements are quite common, **GAUSS** is capable of executing any arbitrary SQL statement.

To insert a new stock symbol, you could do the following:

```
qid = dbExecQuery("INSERT INTO stocks (id,
    symbol, price) VALUES (100, 'TEST', 30)");
```

In the event of multiple inserts, this can be improved through usage of parameter binding:

```
string stocks = { "MSFT", "FB", "GOOG" };
prices = { 32.78, 45.95, 897.96 };

qid = dbCreateQuery("INSERT INTO stocks (id,
    symbol, price) VALUES (:id, :sym, :price)");

for i(1, rows(stocks), 1);
    dbQueryBindValue(qid, ":id", i);
    dbQueryBindValue(qid, ":sym", stocks[i]);
    dbQueryBindValue(qid, ":price", prices[i]);
    dbQueryExecPrepared(qid);
endfor;
```

The following is an identical example, using the same data, but using different style placeholders:

```
qid = dbCreateQuery("INSERT INTO stocks (id,
    symbol, price) VALUES (?, ?, ?)");

for i(1, rows(stocks), 1);
    dbQueryBindValue(qid, 1, i);
    dbQueryBindValue(qid, 2, stocks[i]);
    dbQueryBindValue(qid, 3, prices[i]);
    dbQueryExecPrepared(qid);
endfor;
```

This operation is even possible using a short-hand method:

```
for i(1, rows(stocks), 1);
    string args = itos(i) $| stocks[i] $| itos(prices[i]);
    qid = dbExecQuery("INSERT INTO stocks (id,
        symbol, price) VALUES (?, ?, ?)", args);
endfor;
```

23.2.4 Transactions

If the underlying database engine supports transactions, **dbHasFeature**(*id, DB_TRANSACTIONS*) will return 1. You can use **dbTransaction**(*id*) to start a transaction, proceeded by SQL commands you wish to execute. At any time within the context of an open transaction, you can **dbCommit**(*id*) to finalize the executed commands or **dbRollback**(*id*) to reverse all commands executed since the start of the transaction. When using transactions you must start the transaction before you create or execute a query.

Transactions can be used to ensure that a complicated operation is atomic, meaning every part of the operation is completed or none at all.

For example, in the context of 1000 separate inserts, **dbRollback** can be called to cancel all of them, even if called on the 999th insert:

```
dbTransaction(id);

for i (1, 1000, 1);
    dbExecQuery(id, "INSERT INTO temp (num) VALUES (?)", i);

    if i == 999;
        dbRollback(id);
        break;
    endif;
endfor;
```

24 Foreign Language Interface

The Foreign Language Interface (FLI) allows users to create functions written in C, FORTRAN, or other languages, and call them from a **GAUSS** program. The functions are placed in dynamic libraries (DLLs, .so's, .dylib's, or 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.

24.1 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 func-
   tion.

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

3. Return an integer.

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

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

2. int **func**(double *arg1 [, double *arg2,...]);

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.

24.2 Creating Dynamic Libraries

The following describes how to build a dynamic library called `hyp.dll` (on Windows) or `libhyp.so` (on Linux) from the source file `hyp.c`.

As mentioned in the previous section, your FLI functions may take only pointers to doubles as arguments. Therefore, you should define your FLI functions to be merely wrapper functions that cast their arguments as necessary and then call the functions that actually do the work. This is demonstrated in the source file `hyp.c`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// This code is not meant to be efficient. It is meant
// to demonstrate the use of the FLI.
// This does all the work, not exported
static int hypo(double *x, double *y, double *h, int r,
int c)
{
   double *wx;
   double *wy;
   double *dp;
   double *sp1;
   double *sp2;
   int i, elems;

   elems = r*c;

    // malloc work arrays
   if ((wx =(double *)malloc(elems*sizeof(double))) ==
NULL)
      return 30;      // out of memory

   if ((wy =(double *)malloc(elems*sizeof(double))) ==
NULL)
   {
```

```
    free(wx);
    return 30;        // out of memory
}

dp = wx;
sp1 = x;

// square x into work area wx
for (i=0; i<elems; i++)
{
    *dp = *sp1 * *sp1;
    ++sp1;
    ++dp;
}

dp = wy;
sp2 = y;

// square y into work area wy
for (i=0; i<elems; i++)
{
    *dp = *sp2 * *sp2;
    ++sp2;
    ++dp;
}

dp = h;
sp1 = wx;
sp2 = wy;

// compute hypotenuse into h which was allocated by GAUSS
for (i=0; i<elems; i++)
{
    *dp = sqrt(*sp1 + *sp2);
    ++sp1;
    ++sp2;
```

```
        ++dp;
    }

    // free whatever you malloc
    free(wx);
    free(wy);

    return 0;
}

/* exported wrapper, all double * arguments, calls the real
** function with whatever data types it expects
*/
int hypotenuse(double *x, double *y, double *h, double *r,
double *c)
{
    return hypo( x, y, h, (int)*r, (int)*c);
}
```

The following Makefiles contain the compile and link commands you would use to build
the dynamic library on various platforms. For explanations of the various flags used, see
the documentation for your compiler and linker.

Windows

```
hyp.dll: hyp.obj
        link /dll /out:hyp.dll hyp.obj
hyp.obj: hyp.c
        cl -c -MD -GX hyp.c
```

Linux

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

```
CCOPTIONS = -g -O2 -fpic -lm -lc -shared
CC = gcc
```

```
libhyp.so: hyp.cpp
$(CC) $(CCOPTIONS) -o $@ hyp.c
```

For details on linking your dynamic library, see `dlibrary` in the GAUSS LANGUAGE REFERENCE.

25 Data Transformations

GAUSS allows expressions that directly reference variables (columns) of a data set. This is done within the context of a data loop:

```
dataloop infile outfile;
 drop wages interest_rate;
 csed = ln(sqrt(csed));
 select exports > 3.35 and married $== "y";
 make chfac = hcfac + wcfac;
 keep unemployment chfac exports;
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.

25.1 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	Allow 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	Control 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.

25.2 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.

25.3 Debugging Data Loops

The translator that processes data loops can be turned on and off. When the translator is on, there are three distinct phases in running a program:

Translation	Translation of main program file to temporary file.
Compilation	Compilation of temporary file.
Execution	Execution of compiled code.

25.3.1 Translation Phase

 In the translation phase, the main program file is translated into a temporary file. Each data loop is translated into a procedure and a call to this procedure is placed in the temporary file at the same location as the original data loop. The data loop itself is com-

mented out in the temporary file. All the data loop procedures are placed at the end of the temporary file.

Depending upon the status of line number tracking, error messages encountered in this phase will be printed with the file name and line numbers corresponding to the main file.

25.3.2 Compilation Phase

In the compilation phase, the temporary file is compiled. Depending upon the status of line number tracking, error messages encountered in this phase will be printed with the file name and line numbers corresponding to both the main file and the temporary file.

25.3.3 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.

25.4 Reserved Variables

The following local variables are created by the translator and used in the produced code:

x_cv	x_iptr	x_ncol	x_plag
x_drop	x_keep	x_nlag	x_ptrim
x_fpin	x_lval	x_nrow	x_shft
x_fpout	x_lvar	x_ntrim	x_tname
x_i	x_n	x_out	x_vname
x_in	x_name	x_outtyp	x_x

These variables are reserved, and should not be used within a dataloop... endata section.

26 The GAUSS Profiler

GAUSS includes a profiler, which enables you to determine exactly how much time your programs are spending on each line and in each called procedure, thereby providing you with the information you need to increase the efficiency of your programs. The **GAUSS** Profiler (`gaussprof`)and `tcollect` are both run from a command prompt window, not at a **GAUSS** prompt.

26.1 Using the GAUSS Profiler

There are two steps to using the **GAUSS** Profiler: collection and analysis.

26.1.1 Collection

To collect profiling information, you must run your **GAUSS** program in `tcollect`, an executable shipped with **GAUSS** that is identical to `tgauss` except that it generates a file containing profiling information each time it is run:

```
tcollect -b myfile.e
```

The output displayed by `tcollect` includes the name of the output file containing the profiling information. tcollect output files have a gaussprof prefix and a `.gco` extension.

Note that running `tcollect` on long programs may generate a very large `.gco` output file. Thus you may want to delete the `.gco` files on your machine regularly.

26.1.2 Analysis

To analyze the information stored in the `tcollect` output file, you must run the `gaussprof` executable, which is also shipped with **GAUSS**, on that file. `gaussprof` produces an organized report, displaying the time usage by procedure and by line.

Assuming that running `myfile.e` in `tcollect` produced an output file called `gaussprof_001.gco`, you could analyze the results in that file as follows:

```
gaussprof gaussprof_001.gco
```

The syntax for gaussprof is:

```
gaussprof  [flags] profile_data_file ...
```

where *[flags]* may be any of the following:

-p	profile procedure calls
-l	profile line numbers
-h	suppress headers

-sp order	procedure call sort order where *order* contains one or more of the following:	
	e	exclusive time
	t	total time
	c	number of times called
	p	procedure name
	a	ascending order
	d	descending order (default)
	Columns are sorted all ascending or all descending.	
-sl order	line number sort order where order contains one or more of the following:	
	t	time spent on line
	c	number of times line was executed
	f	file name
	l	line number
	a	ascending order
	d	descending order (default)
	Columns are sorted all ascending or all descending.	

The default, with no flags, is: *-pl -sp dep -sl dtf.*

27 Time and Date

GAUSS offers a comprehensive set of time and date functions. These functions afford the user the ability to return the current time and date, to carry out most related calculations and format the results for output. **GAUSS** also allows the user to perform timed iterations.

In the year 1 AD the calendar in general use was the Julian calendar. The Gregorian calendar that we use today was not invented until the late 1500's. This new calendar changed the method of calculating leap years on century marks. With the Julian system simply every fourth year was a leap year. The Gregorian system made every fourth year a leap year with the exception of century marks which are only leap years if divisible by 400. The British adoption of this calendar, which the **GAUSS** date functions are based on, did not happen until the year 1752. In that year eleven days were removed; September 2, 1752 was followed by September 14, 1752.

`dtvnormal` and `utctodtv` are accurate back to 1 AD. The rest of the **GAUSS** date functions assume a normal Gregorian system regardless of year. Thus, they will not account for the days taken out in September of 1752, nor will they account for all century marks being leap years before the adoption of the Gregorian system in 1752.

The time is given by your operating system, daylight savings time is not automatically accounted for by **GAUSS** in calculations.

27.1 Time and Date Formats

The Time and Date formats in **GAUSS** fall into one of two major categories, matrix/vector and string. The matrix/vector formats can be used for either calculations or if desired for output. The string formats are, however, mostly for use as ouput. Some manipulation of strings is possible with the use of the **stof** function.

A 4x1 vector is returned by both the **date** and **time** functions.

```
d = date;
d;
1997.00    // Year
5.00000    // Month
29.0000    // Day
56.4700    // Hundredths of a second since midnight

t = time;
t;
10.00    // Hours since midnight
17.00    // Minutes
33.00    // Seconds
13.81    // Hundredths of a second
```

These vectors can be written to a string of the desired form by passing them through the corresponding function.

```
d = { 1997, 5, 29, 56.47 };
datestr(d);
5/29/97
datestrymd(d);
19970529
```

```
t = { 10, 17, 33, 13.81 };
timestr(t);
10:17:33
```

A list and brief description of these, and other related functions is provided in the table in Section 25.2 .

Another major matrix/vector format is the dtv, or date and time vector. The dtv vector is a 1x8 vector used with the **dtvnormal** and **utctodtv** functions. The format for the dtv vector is:

Year	*Month*	*Day*	*Hour*	*Min*	*Sec*	*DoW*	*DiY*
1955	4	21	4	16	0	4	110

Where:

Year	Year, four digit integer.
Month	1-12, Month in year.
Day	1-31, Day of month.
Hour	0-23, Hours since midnight.
Min	0-59, Minutes.
Sec	0-59, Seconds.
DoW	0-6, Day of week, 0=Sunday.
DiY	0-365, Days since Jan 1 of current year.

dtvnormal normalizes a date. The last two elements are ignored for input, as shown in the following example. They are set to the correct values on output. The input can be 1x8 or Nx8.

```
dtv = { 1954 3 17 4 16 0 0 0 };
dtv = dtvnormal(dtv);
```

$$1954 \quad 3 \quad 17 \quad 4 \quad 16 \quad 0 \quad 3 \quad 75$$

```
dtv[3] = dtv[3] + 400;
print dtv;
```

$$1954 \quad 3 \quad 417 \quad 4 \quad 16 \quad 0 \quad 3 \quad 75$$

```
dtv = dtvnormal(dtv);
print dtv;
```

$$1955 \quad 4 \quad 21 \quad 4 \quad 16 \quad 0 \quad 4 \quad 110$$

27.2 Time and Date Functions

Following is a partial listing of the time and date functions available in **GAUSS**.

datestr	Formats a Date vector to a string (mo/dy/yr).
datestrymd	Formats a Date vector to an eight character string of the type yyyymmdd.
dayinyr	Returns day number in the year of a given date.
_daypryr	Returns the number of days in the years given as input.
dtvnormal	Normalizes a 1x8 dtv vector.
etdays	Computes the difference in days between two dates.
ethsec	Computes the difference between two times in hundredths of a second.
etstr	Formats a time difference measured in hundreths of a second to a string.
_isleap	Returns a vector of ones and zeros, 1 if leap year 0 if not.

timestr	Formats a Time vector to a string hr:mn:sc.
timeutc	Universal time coordinate, number of seconds since January 1, 1970 Greenwich Mean Time.
utctodtv	Converts a scalar, number of seconds since, or before, Jan 1 1970 Greenwich mean time, to a dtv vector.

Below is an example of two ways to calculate a time difference.

```
d1 = { 1996, 12, 19, 82 };
d2 = { 1997, 4, 28, 4248879.3 };
dif = ethsec(d1,d2);
ds = etstr(dif);
```

$$dif = 1.1274488e + 09$$

$$ds = \quad 130 \; days \quad 11 \; hours \quad 48 \; minutes \quad 7.97 \; seconds$$

If only the number of days is needed use **etdays**.

```
d1 = { 1996, 12, 19, 82 };
d2 = { 1997, 4, 28, 4248879.3 };
dif = etdays(d1,d2);
```

$$dif = 130.00000$$

The last element of d1 is optional when used as an input for **etdays**. _isleap_ returns a matrix of ones and zeros, ones when the corresponding year is a leap year.

```
x = seqa(1970,1,20);
y = _isleap(x);
delif(x,abs(y-1));
1972.0000       // Vector containing all leap years
```

Time and Date

```
1976.0000          // between 1970 - 1989
1980.0000
1984.0000
1988.0000
```

To calculate the days of a number of consecutive years:

```
x = seqa(1983,1,3);
y = _daypryr(x);
sumc(y);
1096.0000
```

To add a portion of the following year:

```
g = { 1986, 2, 23, 0 };
dy = dayinyr(g);
sumc(y)+dy;
1150.0000
```

For more information on any of these functions see their respective pages in the command reference.

27.2.1 Timed Iterations

Iterations of a program can be timed with the use of the **hsec** function in the following manner.

```
et = hsec;          // Start timer

/* Segment of code to be timed */

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

In the case of a program running from one day into the next you would need to replace the **hsec** function with the **date** function. The **ethsec** function should be used to

compute the time difference; a straight subtraction as in the previous example will not give the desired result.

```
dstart = date;      // Start timer

/* Segment of code to be timed */

dend = date;        // Stop timer

// Convert time difference to seconds
dif = ethsec(dstart,dend)/100;
```

28 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; it is not run from a **GAUSS** prompt but rather from a command prompt window.

The syntax is:

> atog `cmdfile`

where `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.

28.1 Command Summary

The following commands are supported in ATOG:

> **append**   Append data to an existing file.
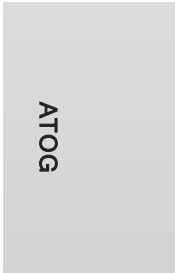
complex	Treat data as complex variables.
input	The name of the ASCII input file.
invar	Input file variables (column names).
msym	Specify missing value character.
nocheck	Don't check data type or record length.
output	The name of the **GAUSS** data set to be created.
outtyp	Output data type.
outvar	List of variables to be included in output file.
preservecase	Preserve case of variable names in output file.

The principle 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;
```

In 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 are:

ATOG

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 lowercase, and the names of numeric variables will be saved in uppercase. The **$** in the **invar** statement specifies that the variables that follow are character type. The **#** specifies numeric. If **$** and **#** are not used in an **invar** statement, the default is numeric.

Comments in command files must be enclosed between '**@**' characters.

28.2 Commands

A detailed explanation of each command 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 that 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 inter-leaved, 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 /research/data/myfile.asc;
```

specifies a file to be located in the /research/data subdirectory.

invar

Soft Delimited ASCII Files

Soft delimited files may have spaces, commas, or cr/lf as delimiters between elements. Two or more consecutive delimiters with no data between them are treated as one delim-iter. 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,
```

while

```
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:

ATOG

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

If the **invar** statement specified three 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:

$$\begin{array}{ccc} 456.4 & 345.2 & 533.2 \\ . & 624.3 & 639.5 \\ 602.3 & 333.4 & . \end{array}$$

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

$$\begin{array}{ccc} 456.4 & 345.2 & 533.2 \\ . & 624.3 & 639.5 \end{array}$$

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 as follows:

> invar record = *reclen (format) variables (format) variables*;

where,

reclen	the total record length in bytes, including the final carriage return/line feed if applicable. Records must be fixed length.
format	(*start,length.prec*) where:
	start starting position of the field in the record, 1 is the first position. The default is 1.
	length the length of the field in bytes. The default is 8.
	prec optional; a decimal point will be inserted automatically *prec* places in from the RIGHT edge of the field.

If several variables are listed after a format definition, each succeeding field will be assumed to start immediately after the preceding field. If an asterisk is used to specify the starting position, the current logical default will be assumed. An asterisk in the

ATOG

length position will select the current default for both *length* and *prec*. This is illegal: (3,8.*).

The type change characters **$** and **#** are used to toggle between character and numeric data type.

Any data in the record that is not defined in a format is ignored.

The examples below assume a 32-byte record with a carriage return/line feed occupying the last 2 bytes of each record. The data below can be interpreted in different ways using different **invar** statements:

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

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 /gauss/dat/test;
```

creates the file `test.dat` on the /gauss/dat directory.

outtyp

Selects the numerical accuracy of the output file. Use of this command should be dictated by the accuracy of the input data and storage space limitations. The format is:

```
    outtyp   fmt;
```

where *fmt* is:

> D or 8   double precision
> F or 4   single precision (default)
> I or 2   integer

The ranges of the different formats are:

Bytes	Type	Significant Digits	Range
2	integer	4	$-32768 <= X <= 32767$
4	single	6-7	$8.43E - 37 <= |X| <= 3.37E + 38$
8	double	15-16	$4.193E - 307 <= |X| <= 1.67E + 308$

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. In this example:

```
    invar $name #age pay $sex #var[1:10] x[005];
    outvar sex age x001 x003 var[1:8];
```

the **outvar** statement selects the following variables:

column	name	data type
1	sex	character
2	AGE	numeric
3	X001	numeric
4	X003	numeric
5	VAR01	numeric
6	VAR02	numeric
7	VAR03	numeric
8	VAR04	numeric
9	VAR05	numeric
10	VAR06	numeric
11	VAR07	numeric
12	VAR08	numeric

outvar is not used with packed ASCII files.

preservecase

Optional; preserves the case of variable names. The default is **nopreservcase**, which will force variable names for numeric variables to upper case and character variables to lower case.

28.3 Examples

Example 1 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 /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.

Example 2 The second example is a packed ASCII file `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 /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.

Example 3 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 /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.

28.4 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

ATOG

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.

29 Maximizing Performance

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

29.1 Library System

Some temporary files are created during the autoloading process. If you have a *tmp_ path* configuration variable or a *tmp* environment string that defines a path on a 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.

29.2 Loops

1. **Use matrix built-in matrix operators rather than 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:

   ```
   //Count starts at 0
   c = 0;

   //Loop counter
   i = 1;
   do while i <= rows(x);

   //If absolute value of x[i] > 1, increment counter
   if abs(x[i]) > 1;
   c = c+1;
   endif;

   //Increment loop counter
   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.

2. **When you must use loops, use `for` loops rather than `do` loops.**

Both of the following loops are equivalent:

```
//'do' loop
i = 1;
do while i < 10;
i = i + 1;
endo;

//'for' loop;
for i(1, 10, 1);

endfor;
```

but the for loop is around 10 times faster and requires less typing!

3. **Preallocate results vectors rather than using concatenation in a loop**

Many algorithms create a vector or matrix of result for each iteration of the main loop. For this purpose you can either use concatenation:

```
//Create an empty matrix
results = {};
```

```
for i(1, 10, 1);
x = rndn(100, 1);

//Append the mean of this iteration's
//'x' to the final results vector
results = results | meanc(x);
endfor;
```

or you can preallocate a vector to the full size of the final results vector and use indexing to place the results from each iteration.

```
iters = 10;

//Create a vector of zeros to hold all results
results = zeros(iters, 1);

for i(1, iters, 1);
x = rndn(100, 1);

//Place the mean from this iteration
//into the results vector
results[i] = meanc(x);
endfor;
```

The code snippets above are equivalent, but for large matrices the second will run many times faster. This is because in the first code section, a new, just slightly larger results vector much be created on each iteration of the loop. The second code section, however, only needs to update the value of one element of the results vector in each iteration.

29.3 Memory Usage

Computers today can have large amounts of RAM. This doesn't mean that large data sets should be read entirely into memory. Many **GAUSS** procedures and applications are written to allow for data sets to be read in sections rather than all at once. Even if you

have enough RAM to store the data set completely, you should consider taking advantage of this feature. The speed-ups using this feature can be significant. For example, **ols** is called using a data set stored in a matrix versus stored on the disk in a **GAUSS** data set. The computer is a 2.8 Megahertz computer with Windows XP.

```
y = rndn(250000,1);
x = rndn(250000,100);
xlbl = 0$+"X"+ftocv(seqa(1,1,100),1,0);
lbl = "Y" | xlbl;

call saved(y~x,"test",lbl);
__output = 0;

t0 = date;
call ols("",y,x);
t1 = date;

t2 = date;
call ols("test", "Y",xlbl);
t3 = date;

print ethsec(t2,t3)/100 " seconds";
print;
print ethsec(t0,t1)/100 " seconds";
    25.750000 seconds
    9.6720000 seconds
```

This represents more than a 50% speedup by leaving the data on the disk.

maxvec, **maxbytes**

maxvec is a **GAUSS** procedure that returns the value of the global variable __ *maxvec* that determines the amount of data to be read in at a time from a **GAUSS** data set. This value can be modified for a particular run by setting __*maxvec* in your command file to some other value. The value returned by a call to **maxvec** can be permanently modified by editing system.dec and changing the value of __*maxvec*. The value returned when running **GAUSS Light** is always 8192.

maxbytes is a **GAUSS** procedure that returns the value of a scalar global $\_\_maxbytes$ that sets the amount of available RAM. This value can be modified for a particular run by setting $\_\_maxbytes$ in your command file to some other value. The value returned by a call to **maxbytes** can be permanently modified by editing system.dec and changing the value of $\_\_maxbytes$.

If you wish to force **GAUSS** procedures and applications to read a **GAUSS** data set in its entirety, set $\_\_maxvec$ and $\_\_maxbytes$ to very large values.

29.3.1 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.

29.3.2 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.

30 Singularity Tolerance

The tolerance used to determine whether or not a matrix is singular can be changed. The default value is 1.0e-14 for both the LU and the Cholesky decompositions. The tolerance for each decomposition can be changed separately. The following operators are affected by a change in the tolerance:

Crout LU Decomposition

crout(x)

croutp(x)

inv(x)

det(x)

y/x when neither x nor y is scalar and x is square.

Cholesky Decomposition

chol(x)

invpd(x)

solpd(y, x)

y/x when neither x nor y is scalar and x is not square.

30.1 Reading and Setting the Tolerance

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

30.2 Determining Singularity

There is no perfect tolerance for determining singularity. The default is 1.0e-14. You can adjust this as necessary.

A numerically better method of determining singularity is to use **cond** to determine the condition number of the matrix. If the equation

```
1 / cond(x) + 1 eq 1
```

is true, then the matrix is usually considered singular to machine precision. (See LAPACK 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.)

31 Publication Quality Graphics

GAUSS Publication Quality Graphics (PQG) is a set of routines built on the graphics functions in **GraphiC** by Scientific Endeavors Corporation. These routines are deprecated but included for backward compatibility.

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.

31.1 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`	Draw 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.

Publication Quality
Graphics

31.2 Using Publication Quality Graphics

31.2.1 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 the header, data setup, graphics format setup, and graphics call.

Header

In order to use the graphics procedures, the **pgraph** library must be activated. 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 their 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 that may be configured 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**, Section 31.6 .)

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 data points only (observed data); the data points 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 loc-
ations */
_plegstr= "Sin wave 1.\0"\          /* 4 lines legend text */
 "Sin wave .8\0"\
 "Sin wave .6\0"\
 "Sin wave .4";
ylabel("Amplitude");                /* Y axis label */
xlabel("X Axis");                   /* X axis label */
title("Example xy Graph");          /* main title */
xy(x,y);                            /* call to main routine */
```

Example 3 In this example, two graphics panels are drawn. The first is a full-sized sur-
face 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 vec-
tor */
z = cos(5*sin(x) - y);              /* z is a 71x71 matrix */
begwind;                            /* initialize graphics win-
dows */
makewind(9,6.855,0,0,0);            /* first window full size
*/
makewind(9/2,6.855/2,1,1,0);        /* second window inset to
```

```
              first */
              setwind(1);                        /* activate first window
              */
              graphset;                          /* reset global variables
              */
              _pzclr = { 1, 2, 3, 4 };           /* set Z level colors */
              title("cos(5*sin(x) - y)");        /* set main title */
              xlabel("X Axis");                  /* set X axis label */
              ylabel("Y Axis");                  /* set Y axis label */
              scale3d(miss(0,0),miss(0,0),-5|5); /* scale Z axis */
              surface(x,y,z);                    /* call surface routine */
              nextwind;                          /* activate second window.
              */
              graphset;                          /* reset global variables
              */
              _pzclr = { 1, 2, 3, 4 };           /* set Z level colors */
              _pbox = 15;                        /* white border */
              contour(x,y,z);                    /* call contour routine */
              endwind;                           /* Display windows */
```

While the structure has changed somewhat, the four basic elements of the graphics program are all here. The additional routines **begwind**, **endwind**, **makewind**, **nextwind**, and **setwind** are all used to control the 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.

31.2.2 Graphics Coordinate System

PQG uses a 4190x3120 pixel resolution 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. (For more information, see **Inch Units in Graphic Panels**, Section 31.3.6 .)

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.

31.3 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. Graphc panels may be any size and position on the window and may be tiled or overlapping, transparent or nontransparent.

31.3.1 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=noontransparent).

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);
```

31.3.2 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 that is 4 inches wide by 2.5 inches tall, 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);
```

31.3.3 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.

31.3.4 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.

31.3.5 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 rt. corner */
logx(x,y);
nextwind;                  /* Graph 3, lower left corner */
logy(x,y);
nextwind;                  /* Graph 4, lower rt. corner */
loglog(x,y);
nextwind;                  /* Graph 5, center, overlayed */
bar(x,y);
endwind;                   /* End graphic panel processing,
                              display graph */
```

31.3.6 Inch Units in 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 "graphic panel 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.

31.3.7 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.

31.4 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:

exttt\ 000	String termination character (null byte).
[	Enter superscript mode, leave subscript mode.
]	Enter subscript mode, leave superscript mode.
@	Interpret next character as literal.
exttt\ 20 n	Select font number n. (see SELECTING FONTS, following).

The escape code $\L$ (or $\l$) 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 `[..]` to create the expression $M(t) = E(e^{tz})$:

```
_pmsgstr = "M(t) = E(e[tx])";
```

Use `@` to generate `[` and `]` in an X axis label:

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

31.4.1 Selecting Fonts

Four fonts are supplied with the **Publication Quality Graphics** system. They are Simplex, Complex, Simgrma, and Microb. (For a list of the characters available in each font, see **PQG FONTS**, CHAPTER 32 .)

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 following example demonstrates how to select a font for use in a string:

```
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.

31.4.2 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 **PQG FONTS**, CHAPTER 32 .) The Simgrma characters are specified by either:

1. The character number, preceeded by a "\".

2. The regular text character with the same number.

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

To produce the title $f(x) = sin^2(\pi x)$, the following title string should be used:

```
title("\201f(x) = sin[2](\204p\201x)");
```

The "p" (character 112) corresponds to "π" in Simgrma.

To number the major X axis tick marks with multiples of $\pi/4$, the following could be passed to **asclabel**:

```
lab = "\2010 \204p\201/4 \204p\201/2 3\204p\201/4 \204p";
asclabel(lab,0);
xtics(0,pi,pi/4,1);
```

xtics is used to make sure that major tick marks are placed in the appropriate places.

This example will number the X axis tick marks with the labels μ^{-2}, μ^{-1}, 1, μ, and μ^2:

```
lab = "\204m\201[-2] \204m\201[-1] 1 \204m m\201[2]";
asclabel(lab,0);
```

This example illustrates the use of several of the special Simgrma symbols:

```
_pmsgstr = "\2041\2011/2\204p ,\201e[-\204m
[\2012]\201/2]d\204m";
```

This produces:

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

31.5 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

Publication Quality Graphics

5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	Grey	15	White

31.6 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 **makewind** in the GAUSS LANGUAGE REFERENCE.) This is used by the 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 9.0x6.855 inches (unrotated) or 6.855x9.0 inches (rotated). If this is 0, the maximum size will be used. Default is 0.
	Note: Used internally. (For the same functionality, see **makewind** in the GAUSS LANGUAGE REFERENCE). This is used by the graphic panel routines. The user must not set this when using the graphic panel procedures.
_parr	Mx11 matrix, draws one arrow per row of the input matrix (for total of M arrows). If scalar zero, no arrows will be drawn.

row

[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**, CHAPTER 31.5 .
[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,1	line type:

0]

1		dashed
2		dotted
3		short dashes
4		closely spaced dots
5		dots and dashes
6		solid

[M,1 1] 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 };
```

_ par row 3 Mx12 matrix, draws one 3-D arrow per row of the input matrix (for a total of M arrows). 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,1 0]   color of arrow, see **COLORS**, CHAPTER 31.5 .

[M,1 1]   line type:

1	dashed
2	dotted
3	short dashes
4	closely spaced dots
5	dots and dashes
6	solid

[M,1 2]   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 };
```

_   scalar, 2x1, or 3x1 vector for independent control for each axis.

pax es	The first element controls the X axis, the second controls the Y axis, and the third (if set) controls 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:

```
/* turn X axis on, Y axis off */
_paxes = { 1, 0 };
/* turn all axes off */
_paxes = 0;
/* turn all axes on */
_paxes = 1;
```

_ _pax ht_	scalar, size of axes labels in inches. If 0, a default size will be computed. Default is 0.

_ _pba rty p_	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**, CHAPTER 31.5 .

_ _pba_	scalar, width of bars in bar graphs and histograms. The valid range is 0-1. If 0, the bars will be a single pixel wide. If 1, the bars will

rwid	touch each other. 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 0, the box plot is drawn as two vertical lines representing the quartile ranges with a filled circle representing the 50th percentile.
[2]	box color, **COLORS**, CHAPTER 31.5 . If 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 = 75th - 25th

min = 25th - 1.5 intqrange

max = 75th + 1.5 intqrange

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.

_
*pbo
xli
m*

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]*.

_
*pco
lor*

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**, CHAPTER 31.5 (*_pcsel* is not documented elsewhere).

_
*pcr
op*

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 a scalar, it 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**, Section 31.4 .

_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**, CHAPTER 31.5 .

[M,9]   line thickness. This value may be 0 or greater. A
value of 0 is normal line width.

To create one error bar using solid lines, use

```
_perrbar = { 1 0 2 2 1 3 6 2 0 };
```

_
pfr   2x1 vector, controls frame around axes area. On 3-D plots this is a
ame   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.

_   2x1 vector to control grid.
pgr
id

[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.

_
plc
trl

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 number of 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 };
```

scalar or 1x4 vector, legend control variable.

_
ple
gct
l

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 pixel
[2]		legend text font size, where 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:	

if item type is line (1<=_*pline[M,1]*<=3):

[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.

if item type is circle (_*pline[M,1]* = 4 or _*pline[M,1]* = 5):

[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.

if item type is radius (_*pline[M,1]* = 6 or _*pline[M,1]* = 7):

[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**, CHAPTER 31.5 .	
[M,9]	controls line thickness. This value may be zero or greater. A value of zero is normal line width.	

_
pli   Mx9 matrix. Allows extra lines to be added to an *xyz* or
ne3   **surface** graph in 3-D plot coordinates.
d

[M,1]	x starting point.
[M,2]	y starting point.
[M,3]	z starting point.
[M,4]	x ending point.
[M,5]	y ending point.
[M,6]	z ending point.
[M,7]	color.
[M,8]	line type:

1	dashed
2	dotted
3	short dashes
4	closely spaced dots
5	dots and dashes
6	solid

[M,9]	line thickness, 0 = normal width.

[M,1 0]	hidden line flag, 1 = obscured by surface, 0 = not obscured.
_plo tsh f	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 graphic panel routines. The user must not set this when using the graphic panel procedures.

_plo tsi z	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 graphic panel routines. The user must not set this when using the graphic panel procedures.

_plt ype	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**, CHAPTER 31.5 .

[1]	axes.
[2]	axes numbers.
[3]	X axis label.
[4]	Y axis label.
[5]	Z axis label.
[6]	title.
[7]	box.
[8]	date.
[9]	background.

If this is scalar, it will be expanded to a 9x1 vector.

_pmsgctl   Lx7 matrix of control information for printing the strings contained in _pmsgstr.

[L,1]	horizontal location of lower left corner of string.
[L,2]	vertical location of lower left corner of string.
[L,3]	character height in inches.
[L,4]	angle in degrees to print string. This may be -180 to 180 relative to the positive X axis.

[L,5]	location coordinate system.	
	1	location of string in plot coordinates
	2	location of string in inches
[L,6]	color.	
[L,7]	font thickness, may be 0 or greater. If 0 use normal line width.	

_
*pms
gst
r*
   string, contains a set of messages to be printed on the plot. Each message is separated from the next by a null byte (\000). The number of messages must correspond to the number of rows in the *_pmsgctl* control matrix. This can be created as follows:

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

_
*pno
tif
y*
   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.

_
*pnu
m*
   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 axis
2	axes numbers displayed, horizontally oriented on axis

For example:

```
_pnum = { 0, 2 }; /* no X axis numbers, *//*
horizontal on Y axis */
```

_
pnu
mht

scalar, size of axes numbers in inches. If 0, a size of .13 will be used. Default is 0.

_
pro
tat
e

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

_
psc
ree
n

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

_
psi
len
t

scalar. If 0, a beep will sound when the graph is finished drawing to the window. Default is 1 (no beep).

_
pst
ype

scalar or Kx1 vector, controls symbol used at data points. To use a single symbol type for all points, set this to one of the following scalar values:

1	circle
2	square
3	triangle
4	plus
5	diamond
6	inverted triangle
7	star
8	solid circle

9	solid square
10	solid triangle
11	solid plus
12	solid diamond
13	solid inverted triangle
14	solid star

If this is a vector, each line will have a different symbol. Symbols will repeat if there are more lines than symbol types. Default is 0 (no symbols are shown).

_
psu
rf

2x1 vector, controls 3-D surface characteristics.

[1]	if 1, show hidden lines. Default is 0.
[2]	color for base, see **COLORS**, CHAPTER 31.5 . 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. Default is 7.

_
psy
m

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**, CHAPTER 31.5 .

[M,6]	type of coordinates:

	1	plot coordinates
	2	inch coordinates
[M,7]	line thickness. A value of zero is normal line width.	

_ *psy* *m3d*	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**, CHAPTER 31.5 .
[M,7]	line thickness. A value of 0 is normal line width.

Use _*psym* for plotting extra symbols in inch coordinates.

_ *psy* *msi* *z*	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.
_ *pte* *k*	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.
_ *pti* *cou* *t*	scalar. If 1, tick marks point outward on graphs. Default is 0.
_ *pti*	scalar, the height of the title characters in inches. If this is 0, a default height of approx. 0.13 inch will be used.

tlht	
_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, no magnification is performed. Default is 0.

    [1]   magnification value. 1 is normal size.

    [2]   horizontal center of zoomed plot (0-100).

    [3]   vertical center of zoomed plot (0-100).

To see the upper left quarter of the screen magnified 2 times use:

```
_pzoom = { 2 25 75 };
```

_pzp
max
    scalar, the maximum number of places to the right of the decimal point for the Z axis numbers. Default is 3.

_pzs
ci
    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.

32 PQG Fonts

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**, Section 31.4.1 .)

32.1 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		

32.2 Simgrma

33	ϵ	61	$\neq$	89	ψ	117	υ
34	$($	62	$\geqq$	90	$\gtrsim$	118	$)$
35	$\equiv$	63	$\simeq$	91	$[$	119	ω
36	$\approx$	64	$\cup$	92	∂	120	ξ
37	$\uparrow$	65	$\frac{1}{2}$	93	$]$	121	ψ
38	$\surd$	66	$\frac{1}{3}$	94	$\cap$	122	ζ
39	$'$	67	H	95	$\downarrow$	123	$\{$
40	$\subset$	68	Δ	96	$\sim$	124	$\int$
41	$\supset$	69	$\frac{1}{8}$	97	α	125	$\}$
42	$\times$	70	Φ	98	β	126	$\propto$
43	$\pm$	71	Γ	99	η		
44	$\int$	72	X	100	δ		
45	$\mp$	73	$\frac{2}{3}$	101	ε		
46	$\cdot$	74	$\perp$	102	φ		
47	$\div$	75	$\frac{3}{8}$	103	γ		
48	∇	76	$\wedge$	104	χ		
49	$\sqrt{}$	77	$\frac{5}{8}$	105	ι		
50	ϕ	78	$\frac{7}{8}$	106	t		
51	$<$	79	$\frac{1}{4}$	107	κ		
52	$>$	80	Π	108	λ		
53	$/$	81	Θ	109	μ		
54	$\exists$	82	P	110	ν		
55	$\|$	83	Σ	111	o		
56	∞	84	$\lesssim$	112	π		
57	$\odot$	85	Υ	113	ϑ		
58	$\rightarrow$	86	$\leftrightarrow$	114	ρ		
59	$\leftarrow$	87	Ω	115	σ		
60	$\leqq$	88	Ξ	116	τ		

32.3 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		

32.4 Complex

PQG Fonts

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		

Index