

GAUSSTM
Programming Language

BASIC GAUSS WORKSHOP
Version 5.0.1
September 13, 2008

Aptech Systems, Inc.
Maple Valley, WA

The information in this workbook is subject to change without notice and does not represent a commitment on the part of Aptech Systems. The manual and the accompanying software are provided under the terms of a license agreement or non-disclosure agreement. The software may be used and copied only according to the terms of the agreement. No part of this manual may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language by any means without the prior written permission of:

Aptech Systems, Inc.
23804 SE Kent-Kangley Road
Maple Valley, WA 98038

Copyright ©1994-2002 by Aptech Systems, Inc.
All Rights Reserved.

September 13, 2008

GAUSS is a trademark of Aptech Systems, Inc.

Contents

1	Introduction	1
1.1	External GAUSS Resources	2
1.2	Programming Tips	2
2	The GAUSS Environment	5
2.1	GAUSS Data Types	5
2.2	The Working (Current) Directory	6
2.3	The GAUSS Symbol Table and Program Execution	6
2.3.1	Symbol Search Order	6
2.3.2	Building Libraries	8
2.4	Configuration	9
2.4.1	The gauss.cfg File	9
2.4.2	sysstate	10
2.5	Exercises	11
3	Help	13
3.1	The GAUSS Manuals	13
3.2	The Help Menu	13
3.3	Context Sensitive Help	16
3.4	Exercises	17

4	The GAUSS Windows Environment	19
4.1	GAUSS Windows	20
4.1.1	Tiling	21
4.2	Status Bar	21
4.3	Running Commands Interactively	22
4.3.1	Exercises	23
4.4	Running Commands from Files	24
4.4.1	Exercises	24
4.5	File Menu	25
4.6	Edit Menu	25
4.6.1	Bookmarks	26
4.6.2	Macros	26
4.7	View Menu	26
4.8	Configure Menu	26
4.8.1	Preferences	27
4.8.2	Editor Properties	30
4.9	Run Menu	31
4.10	Debug Menu	32
4.11	Tools Menu	34
4.12	Window Menu	35
4.13	Help Menu	35
4.14	Exercises	36
5	Data I/O	39
5.1	GAUSS Data Sets	39

5.1.1	Writing GAUSS Data Sets	39
5.1.2	Reading GAUSS Data Sets	41
5.1.3	Using GAUSS Datasets	45
5.1.4	Matrix and String Files	45
5.2	ASCII Files	46
5.2.1	Writing ASCII Files	46
5.2.2	Reading ASCII Files	48
5.2.3	Working with ASCII Files	49
5.3	Spreadsheets	51
5.3.1	Writing Spreadsheets	51
5.3.2	Reading Spreadsheets	52
5.4	Transforming Data Sets - Dataloop Translation	52
5.5	Exercises	53
6	Symbols, Procedures, Global Variables, and Libraries	55
6.1	Procedures	56
6.1.1	Writing a Procedure in the Command Window	57
6.1.2	SRC Files	58
6.1.3	Global Variables in External Procedures	59
6.2	Libraries	59
6.2.1	Building Library Files	60
6.2.2	The Library Tool	61
6.3	User Examples - Procedures and Libraries	62
6.4	Exercises	63

7	Fundamentals	65
7.1	Strings and String Arrays	65
7.1.1	Special Characters in Strings	66
7.2	Numeric and Character Matrices	67
7.2.1	Submatrices	68
7.2.2	Special Matrices	69
7.3	Matrix Operators	70
7.3.1	Conformability	71
7.3.2	Numeric Operators	71
7.3.3	Exercises	74
7.3.4	Other String and Matrix Operators	74
7.4	N-Dimensional Arrays	75
7.4.1	Array Definition	76
7.4.2	Array to Matrix Conversion	77
7.4.3	Array Manipulation	78
7.4.4	Other Array Operators	82
7.5	Structures	82
7.5.1	Arrays of Structures	82
7.6	Type Conversion	82
7.6.1	Using the \$+ operator	83
7.6.2	Exercises	84
7.7	Missing Values	85
7.7.1	Exercises	86
7.8	Scalar and Element-by-Element Relational Operators	86

7.9	Scalar and Element-by-Element Logical Operators	88
7.9.1	Exercises	89
7.10	Formatting Output and Printing Output	89
7.10.1	Printing Mixed Matrices of Characters and Numbers	92
7.10.2	Exercises	94
7.11	Control Statements	95
7.11.1	Loops	95
7.11.2	Conditional Branching	96
7.12	Procedures and Keywords	97
8	Graphics	99
8.1	Using the VWR Graphics Viewer	100
8.2	Graphics Windows	100
8.2.1	Menus	101
8.3	Using PQG Graphics	102
8.3.1	Header	102
8.3.2	Data Setup	102
8.3.3	Graphics Format Setup	102
8.3.4	Graphics Coordinate System	102
8.3.5	Graphic Panels	103
8.3.6	Using Graphic Panel Functions	104
8.3.7	Saving Graphic Panel Configurations	107
8.4	Graphics Text Elements	107
8.4.1	Selecting Fonts	108
8.4.2	Greek and Mathematical Symbols	108
9	TGAUSS - The Command Line Interface	111
9.1	Interactive Commands	112
	Index	117

Chapter 1

Introduction

GAUSS is a powerful matrix programming language that is ideal for mathematical and statistical applications. The ability to test code on the fly, quickly debug it and, in **GAUSS** for Windows, to quickly cut and paste across windows, significantly speeds application development and provides for modularity and reusability.

Computations are usually much faster when matrix operations are used instead of **do** or **for** loops. For example, consider a generalized least squares estimator,

$$\hat{\beta}_{gls} = (X'\Omega^{-1}X)^{-1}X'\Omega^{-1}Y$$

In the case of heteroscedasticity, Ω is diagonal and the estimator is produced by weighted least squares:

$$\hat{\beta}_{gls} = (X^*X^*)^{-1}X^{*'}Y^*$$

where $x_{ij}^* = x_{ij}/\sqrt{\omega_i}$ and $y_i^* = y_i/\sqrt{\omega_i}$.

The less efficient procedure performs this calculation using a do loop. It assumes that the independent variables are stored in a $T \times k$ matrix **x**, the dependent variables in a $T \times 1$ column vector **y**, and the weights in a $T \times 1$ column vector **w**, where the i^{th} element of **w** is equal to $1/\sqrt{\omega_i}$.

```
proc wls_a(y,x,w);
  local i, j;
  i = 1;
  do until i > rows(w);
    y[i] = y[i] * w[i];
    j = 1;
```

```

do until j > cols(x);
  x[i,j] = x[i,j] * w[i];
  j = j + 1;
endo;
i = i + 1;
endo;
retp(olsqr(y,x));
endp;

```

The more efficient procedure uses matrix multiplication within **GAUSS**.

```

proc wls_v(y,x,w);
  retp(olsqr(y.*w,x.*w));
endp;

```

The calculations are now nested completely in the return from the procedure. Nesting calculations contributes to greater speed.

1.1 External GAUSS Resources

The Aptech Systems, Inc. website, www.aptech.com, contains a list of external links to **GAUSS** resources. Click the **Links** button at the top of the home page.

The first item, the GAUSSIANS list, is an excellent way to learn **GAUSS** tips and solutions from other users. To subscribe to GAUSSIANS, send

```
subscribe gaussians
```

in the body of an email message (not on the subject line) to majordomo@eco.utexas.edu.

Another useful link is The American University archive of **GAUSS** code, containing numerous free procedures. The American University archive of **GAUSS** code is at <http://gurukul.ucc.american.edu/econ/gaussres/GAUSSIDX.HTM>. This link also contains directions to other collections of **GAUSS** code.

The list of **GAUSS** resources also contains external **GAUSS** tutorials and manuals.

1.2 Programming Tips

Your programs will be easier for others to follow and revise if you follow some simple tips:

1. INTRODUCTION

- *Clean code means that do loops are indented*
- *Clean code means that comments are liberally used, allowing readers to follow the program's logic. **GAUSS** has three comment styles:*
 1. *Enclose a block of comments between two @ symbols.*
 2. *Enclose a block of comments between /* and */.*
 3. *Comments on a single line are preceded with //, the C++ comment style.*
- *Clean code means using spaces liberally, and not necessarily including as many operations as possible in a single line of code.*

1. *INTRODUCTION*

Chapter 2

The GAUSS Environment

This chapter briefly explains the **GAUSS** environment and how to configure it. The purpose is to make the chapter on the **GAUSS** Windows Environment easier to follow. Short discussions of **GAUSS** data types, structures, the working directory, the **GAUSS** symbol table, and external symbols and libraries are provided. More complete discussions of these topics are in Chapters 4 and 6.

2.1 GAUSS Data Types

GAUSS has six data types, matrices, strings, string arrays, structures, scalar members of a structure, and arrays.

The **show** and **type** commands are often used to see the type of a symbol. **show** is used to display symbol table information. **show symbolname** returns whether **symbolname** is a matrix, string, string array, structure, array, scalar structure member, procedure, function, or keyword.

The **type** command, e.g. **type(x)**, returns the following:

Type	Type Number
matrix	6
string	13
string array	15
array	21
structure	17
scalar structure element	6

Matrices may contain numeric and character data; **GAUSS** does not distinguish internally between the two. Both are stored as double precision (8 byte) data. Numeric matrices and numeric matrix elements are printed to the screen using a **print** command (e.g. **print x;**). Character matrix elements are limited in length to 8 characters.

Character matrix elements are printed to the screen using a **print \$variablename** command. Strings are printed using a **print variablename** command.

2.2 The Working (Current) Directory

The working directory, also known as the current directory, is where program development occurs. Unless explicit paths are provided in code or in the `gauss.cfg` file (see below), it is where files of all sorts (program files, data files, etc) are saved and from where they are read. For example, **GAUSS** looks in the working directory for the file specified in `>> edit filename;`. If it does not exist, a new file named `filename` will be created and, when saved, will exist in the current directory.

The location of the current directory is found by typing `cdir(0);` from the **GAUSS** prompt.

The current directory is changed using a `chdir` or `changedir` command, as in:

```
chdir \myproject;
```

2.3 The GAUSS Symbol Table and Program Execution

Symbols are the names of strings, string arrays, matrices, structures, arrays, procedures, keywords, and functions. Symbols may be global in scope or local to a procedure.

The **GAUSS** symbol table holds symbol names in memory. All symbols on the right side of an expression, except **GAUSS** intrinsic procedure names, must exist in the symbol table prior to being operated upon.

2.3.1 Symbol Search Order

GAUSS processes a line of code by searching for its symbol names and symbol definitions. The search occurs in a well-defined order. **GAUSS** first determines whether a given symbol name is an intrinsic symbol name, second whether it exists in the symbol table, or third whether it exists in the active library files (`.lcg` files). All

2. THE GAUSS ENVIRONMENT

library files are in the **lib_path**, a single directory whose location is specified in the **gauss.cfg** file (a text file in the **GAUSS** installation directory).

Library files contain global symbol names and references to files defining the symbols. Here is a subset of the **CML** library file, **cml.lcg**, written two ways. The first set contains explicit paths to the files containing the source code for the referenced symbols. The second set does not contain paths to the source code files.

```
// ----- First Set -----
c:\gauss50\src\cml.dec
    _cml_ver                : matrix
    _cml_Algorithm          : matrix
    ...
c:\gauss50\src\cml.src
    cml                     : proc
    CMLset                  : proc

// ----- Second Set -----
cml.dec
    _cml_ver                : matrix
    _cml_Algorithm          : matrix
    ...
cml.src
    cml                     : proc
    CMLset                  : proc
```

Global variables are defined in **.ext** and **.dec** files. Procedures, functions, and keywords are defined in **.src** files. These topics are discussed at length in Chapter 6.

When a symbol name is found in a library file, **GAUSS** compiles the entire file containing the source code for the symbol's definition, inserting all its symbols into the symbol table. **GAUSS** finds a source code file using either the explicit path or, if no path is provided, by searching for the referenced file name in the **src_path**, specified in either the **gauss.cfg** file or in a subsequent call to **sysstate**, option 22.

The search through library files for symbol names occurs in the order in which the libraries are activated. Two libraries are always active, the **User** and **GAUSS** libraries (unless they are deactivated using the **Configure/Preferences/Compile Options** tab page). Other libraries are made active using the **library** command. The following results in four active libraries:

```
library cml, pgraph;
```

In this case, **GAUSS** searches first through the **User** library for a symbol name, next through the **cml** and **pgraph** libraries, and finally through the **GAUSS** library.

2.3.2 Building Libraries

Libraries are constructed from **lib** commands or by using the **Lib Tool**, available from the **Tools** menu.

The **lib** Command

Here are two sets of **lib** commands, each constructing the Constrained Maximum Likelihood module library file, `cml.lcg`. The first set of **lib** statements inserts explicit paths to the source code symbol definition files in the `cml.lcg` file. The second set of **lib** statements uses the `-nopath` option, causing **GAUSS** to look for the symbol definition files in the `src_path`,

```
1. lib cml cml.dec -a; // Creates a path in the cml.lcg file
   lib cml cml.src -a; // Creates a path in the cml.lcg file

2. lib cml cml.dec -n; // Doesn't create a path in the cml.lcg file (default)
   lib cml cml.src -n; // Doesn't create a path in the cml.lcg file (default)

   lib cml cml.dec;    // Doesn't create a path in the cml.lcg file (default)
   lib cml cml.src;    // Doesn't create a path in the cml.lcg file (default)
```

The Library Tool

Open the Library Tool by clicking the **Lib Tool** menu item on the **Tools** menu. Create a new library with the **New Library** button. Remove a library by selecting the **Delete Library** button. Add files to a library with the **Add** button. Remove files from a library with the **Remove** button.

To add absolute path names to the library index, use the **Add Paths** button. To only use file names and the `src_path` variable (defined in the `gauss.cfg` file) for searching libraries, use the **Strip Paths** button.

Use **Rebuild** to recompile all the files used in the library, and rebuild the library index file. Use the **Revert to Original** button to revert to the configuration the library was in when the Library Tool was opened.

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

2. THE GAUSS ENVIRONMENT

2.4 Configuration

Certain features of the **GAUSS** environment are configured either before starting **GAUSS** by editing the `gauss.cfg` file or at run-time by calling the `sysstate` function. `sysstate` is also used to determine the state of these features at run-time.

2.4.1 The `gauss.cfg` File

The `gauss.cfg` file is a text file in the **GAUSS** installation directory. It contains numerous options that are set when **GAUSS** is started. The top part of the `gauss.cfg` file sets default paths for various types of files. Here is a portion of the top part of the `gauss.cfg` file. Note the comment lines indicating whether single or multiple paths are allowed. The `GAUSSDIR` variable refers to the **GAUSS** installation directory.

1. This section of the `gauss.cfg` file defines the `src_path` and `lib_path`.

```
# multiple paths for program files
src_path = $(GAUSSDIR)\src;$(GAUSSDIR)\examples

# one path for library files
lib_path = $(GAUSSDIR)\lib
```
2. This section of the `gauss.cfg` file defines the path to the error log file.

```
# one path for the error log file
err_path = $(GAUSSDIR)\wksp
```
3. This section of the `gauss.cfg` file defines the path to the command log file. The command log may be opened in an `Edit` window. It is treated like any other file opened in an `Edit` window. Code lines may be deleted or inserted and blocks of code may be selected and executed. Blocks of code can also be cut or copied and pasted to another file or to the `Command` window.

```
# one path and filename for the command log
log_file = $(GAUSSDIR)\wksp\command.log
```
4. This section of the `gauss.cfg` file defines the path to DLL files. They are made available to a **GAUSS** program with the `dlibrary` command.

```
# one path for DLIBRARY command
dlib_path = $(GAUSSDIR)\dlib
```
5. The following sections define paths for various **GAUSS** save and load commands. They are initially set so that all saves and loads occur to and from the working directory unless an explicit path is provided in code.

```

# one path for SAVE command
#save_path =

# one path for LOADM command
#loadm_path =

# one path for LOADP, LOADF, LOADK commands
#loadp_path =

# one path for LOADS command
#loads_path =

# one path for workspace files
workspace path = $(GAUSSDIR)\wksp

```

Numerous other default settings are in the `gauss.cfg` file, including turning on or off the dataloop translator and setting compiler options, printer options, and various tolerances used in **GAUSS** calculations. Many of these may also be set at runtime, using the `sysstate` command.

2.4.2 sysstate

`sysstate` is called interactively or from a **GAUSS** program. It modifies various runtime options. The first argument in the call to `sysstate` determines the feature to be modified.

- 1 **GAUSS** version
- 2 EXE file location
- 4 **save** path
- 5 **load**, **loadm** path
- 6 **loadf**, **loadp** path
- 7 **loads** path
- 8 toggles complex numbers, default on.
- 9 sets trailing character for imaginary part, default i.
- 10 printer width, default = 80
- 11 auxiliary output width, default = 80
- 12 precision, 64 or 80
- 13 LU tolerance, default = 1e-14
- 14 Cholesky tolerance, default = 1e-14
- 15 screen on or off, default on

2. THE GAUSS ENVIRONMENT

- 16 automatic print mode
- 17 automatic lprint mode
- 18 auxiliary output parameters
- 19 print format
- 21 imaginary tolerance, default 2.23e-16. Imaginary parts to numbers are omitted unless that part exceeds this tolerance.
- 22 source path, the **src_path** variable
- 24 DLL directory
- 25 toggles how missing values are treated in comparisons
- 30 base year toggle

The first line of the following **GAUSS** code prints the location of DLL files. The second line returns the location of the DLL files, to a string variable. Lines 4 through 7 verify that a string variable was returned. The last line is another way of verifying that a string was returned.

```
print sysstate(24,0);
y = sysstate(24,0);
if (type(y) == 13);
    print "y is a string";
else;
    print "y is not a string";
endif;
show y;
```

GAUSS online help for **sysstate** contains a complete discussion of the options and setting them.

2.5 Exercises

- 2.1 Find the library path in the `gauss.cfg` file (in your **GAUSS** installation directory). Look at the files in this directory. Open one of them and look at the symbols it contains. Pick a procedure in your chosen `.lcg` file and find its definition (in a `.src` file).
- 2.2 Do the same for global variables, finding them in an `.lcg` file and their explicit definition in a `.dec` file.
- 2.3 Use **sysstate** to view the value of **src_path**.

2. *THE GAUSS ENVIRONMENT*

Chapter 3

Help

3.1 The GAUSS Manuals

Two pdf files are shipped with **GAUSS**, the *GAUSS User's Guide* and a *GAUSS Language Reference*. The pdf files may also be downloaded via ftp from ftp.aptech.com. The manuals may be read with Adobe Acroread, available for free from the Adobe website, www.adobe.com. The manuals may also be accessed via the **GAUSS** help system.

Be sure to enter ftp.aptech.com from a command window, not your browser's address box.

3.2 The Help Menu

The **GAUSS** Help Menu has the following items:

- User's Guide
- Keyboard
Lists **GAUSS** keystroke shortcuts.

Up arrow	Up one line
Down arrow	Down one line
Left arrow	Left one character

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

Edit Keys

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

Text Selection Keys

SHIFT+Up arrow	Select one line of text up
SHIFT+Down arrow	Select one line of text down
SHIFT+Left arrow	Select one character to the left
SHIFT+Right arrow	Select one character to the right
SHIFT+CTRL+Left arrow	Select one word to the left
SHIFT+CTRL+Right arrow	Select one word to the right
SHIFT+HOME	Select to beginning of the line
SHIFT+END	Select to end of the line
SHIFT+PAGE UP	Select up one screen
SHIFT+PAGE DOWN	Select down one screen
SHIFT+CTRL+HOME	Select to the beginning of the document
SHIFT+CTRL+END	Select to the end of the document

Command Keys

CTRL+A	Redo
CTRL+C	Copy selection to the Windows clipboard
CTRL+D	Open the Debug window
CTRL+E	Open the Matrix Editor
CTRL+F	Find/Replace text
CTRL+G	Go to the specified line number
CTRL+I	Insert the GAUSS prompt
CTRL+L	Insert last
CTRL+N	Make the next window active

3. *HELP*

CTRL+O	Open the Output window and change its state
CTRL+P	Print the current window, or selected text
CTRL+Q	Exit GAUSS
CTRL+R	Run selected text
CTRL+S	Save the window to a file
CTRL+W	Open the Command window
CTRL+V	Paste the contents of the Windows clipboard
CTRL+X	Cut the selection to the Windows clipboard
CTRL+Z	Undo

Function Keys

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

- **Reference**

Opens the online **GAUSS** Language Reference guide. The Guide contains the syntax for each **GAUSS** command. Pages from the **GAUSS** Language Reference guide appear when F1 help is invoked.

- **Tip of the Day**

Click to see a **GAUSS** tip.

- **About **GAUSS****

The Kernel Rev. and GUI Rev. numbers are given here. You should supply the Kernel Rev. number to Aptech Systems, Inc. should you need to contact us for technical support.

3.3 Context Sensitive Help

GAUSS for Windows has four types of context sensitive help.

1. The Help toolbar button

The Help toolbar button may be placed anywhere on the **GAUSS** screen to obtain help. For example, click the Help toolbar button and drop it on the toolbar.

2. F1 help

F1 help displays the **GAUSS** Command Reference for particular commands. Simply place the cursor on any command (in either a **Command** or an **Edit** window) and press F1.

3. Ctrl+F1 help

Ctrl-F1 help invokes the **GAUSS** Source Code Browser. It shows the source code for external procedures in active libraries when the cursor is placed in a command and Ctrl+F1 is pressed.

Check Ctrl-F1 help by typing **ols** in a **Command** or **Edit** window, followed by Ctrl+F1. You will see source code for the **ols** procedure. Now type **xy** (creates a plot) and press Ctrl+F1. You will get an error message, **Symbol: xy not found**. The reason is that the **pgraph** library is not active. Make it active and press Ctrl+F1 again to see the **xy** source code.

*Two libraries are always active (unless turned off using **Configure/Preferences/Compile Options**), the **gauss** library and the user library. Check this by typing **library** from a **GAUSS** prompt. Other libraries are activated with the **library** command. For example,*

```
library lib1, lib2;
```

activates libraries lib1 and lib2, making four libraries active. Chapter 6 contains an expanded discussion of libraries.

Users may write their own context sensitive help by inserting ****> procedurename** in the comment section of their procedure. Pressing Ctrl+F1 when the cursor is in **procedurename** causes **GAUSS** to search the active library files (.lcg files) for **procedurename**. When found, **GAUSS** opens the corresponding .src file in a **Edit** window. **GAUSS** then looks for a new line, followed by two ****** characters, followed by the **>** sign, followed by **procedurename**. This line appears at the top of the user's screen. Lines below the top line, in the comment section, explain the procedure.

4. Source Code Browser

A different kind of context sensitive help is provided by the Source Browser, accessed from the **Tools** menu. The Source Browser (similar to Grep) lets you

3. *HELP*

find lines from files in a specified directory that match a given pattern. It's useful for finding functions when you know what you want accomplished but not the procedure name.

3.4 Exercises

- 3.1 Choose some arbitrary intrinsic and external commands. Intrinsic commands may be chosen by clicking the **Help** menu, and navigating through either the **Commands by Category** or **Alphabetical List of Commands** lists. External commands may be found by looking in **src** files, located in the **GAUSS installation\src** subdirectory. Use **F1** and **Ctrl+F1** help on the commands you chose. Note the **>** sign next to the procedure name when using **Ctrl+F1** help.
- 3.2 Create standard normal and uniform random numbers using the **rndKMn** and **rndKMU** commands.

Look at www.aptech.com/papers for a technical discussion of the Kiss-Monster random number generator.

Check the syntax of these commands by pressing **F1** after typing **rndKMn** or **rndKMU** in the **Command Window** or in an **Edit Window**. Use **Ctrl+C** and **Ctrl+V** to copy and paste the format part of the **Help** window to the command window. Enter appropriate values for **r** and **c** and **-1** for the **state** argument. Press return. Print the matrix you created (e.g. **print y;**).

3. *HELP*

Chapter 4

The GAUSS Windows Environment

This chapter describes the **GAUSS** Windows environment. Simple programming examples are presented. You are encouraged to duplicate these programs. Essential concepts to remember when working through the examples are:

1. *All **GAUSS** command lines end in a semicolon*
2. *Clear the screen by typing **cls**; from the command prompt*
3. *Delete the symbol table (start over with a clean workspace) by typing **new**; from the command prompt*

Start **GAUSS** for Windows. You will see the **GAUSS** menu bar, the **GAUSS** toolbar, the working directory toolbar, a **Command** window and, near or at the bottom of the screen, the **GAUSS** status bar. Your cursor will be in the **Command** window, next to a **GAUSS** prompt, >>, in the lower left corner of the screen.

Click the Help toolbar button and drop it on the one of the toolbars. A description of the toolbar components will be loaded into a **Help** window.

Float your mouse over different parts of the **GAUSS** toolbar. You will see pop-up descriptions of each item.

1. ***GAUSS** runs the file **startup** when it starts. Often users put a **chdir** command in the startup file to change the **GAUSS** working directory.*

4. THE GAUSS WINDOWS ENVIRONMENT

2. **GAUSS** starts up with the same configuration it had when last shutdown. Startup options are set in the **GAUSS** installation directory's **startup** file or using the **Configure/Preferences** and **Configure/Editor Properties** menu items. **GAUSS** starts with the preferences that were in place when it was last shutdown.
3. Each window, including each **Edit** window, has its own set of options. This means that you will need to configure each window independently.

4.1 GAUSS Windows

GAUSS for Windows runs in **Command**, **Edit**, and **Debug** modes. There are windows for each of these modes. Additional windows include an **Output** window, a **Matrix Editor** window, and an **HTML Help** window. Code may be run from the **Command** window or an **Edit** window. Output may appear in the **Command** window, an **Output** window, or sent to a file.

Command Window Enter interactive commands and view output in the **Command** window. This window is selected from the Windows menu, by clicking inside it or by using the keyboard shortcut, **Ctrl+W**. Output is written to the **Command** window when an **Output** window is not open.

Edit Window Edit windows are created and opened a number of ways.

1. Clicking the **New** button on the toolbar opens a new **Edit** window.
2. Typing **edit filename** from the **GAUSS** prompt opens a new **Edit** window if **filename** does not already exist in the current directory. For example, typing **edit myfile.src** opens **myfile.src** for editing. It will reside in the working directory when saved. Typing **edit c:\mydir\myfile.src** opens **myfile.src** for editing. It will reside in the **c:\mydir** directory when saved.
3. Typing **edit filename** from the **GAUSS** prompt opens the the editor for an existing file if **filename** already exists.

Ctrl+N cycles through all open windows.

Output Window Output is written to the **Output** window when the **Output** window is open and to the **Command** window when the **Output** window is not open. **GAUSS** commands cannot be executed from the **Output** window.

Debug Window The **Debug** window is displayed during a debugging session. The **Debug Window** contains a full debugger with breakpoints and watch variables, context sensitive help, and a source code browser.

4. THE GAUSS WINDOWS ENVIRONMENT

4.1.1 Tiling

Tiling the **Command** and/either the **Edit** and/or **Output** windows horizontally or vertically is very useful when developing programs. Horizontal tiling is chosen when the output occupies line lengths are long and vertical tiling is chosen when the output line lengths are short. The active window is one with focus. Toggle focus between all open windows using **Ctrl+N** or clicking in the window you want active. All open windows are listed at the bottom of the **Windows** Menu.

GAUSS will automatically tile the input (a **Command** or **Edit** window) and output (a **Command** or **Output** window) windows when **Dual Vertical** or **Dual Horizontal** is selected.

4.2 Status Bar

The Status bar is located along the bottom of the **GAUSS** window. It has six panels.

1. **GAUSS** Status

The first panel of the Status bar shows the current **GAUSS** status. The most common message is **Running filename**.

2. **Cursor** Location

The second panel shows the line number and column number where the cursor is located. This information is useful when moving around a file in the **Edit** window. When a block of text is selected, the values indicate the first position of the selected text.

3. **Data**loop

The third panel shows whether **data**loop translation is active.

4. **OVR**

The fourth panel shows whether overstrike (**OVR**) is active. Press the **Insert** key to toggle between **OVR** active and **OVR** not active.

5. **Caps** Lock

The fifth panel shows whether **Caps** Lock is on.

6. **Num** Lock

The sixth panel shows whether **Num** Lock is on.

4.3 Running Commands Interactively

Single commands or blocks of commands may be run interactively from the **Command Window**. Select the **Command** window by clicking clicking inside it or by typing the keyboard shortcut **Ctrl+W**.

*Use keyboard shortcuts to significantly speed development time. For example, to delete from the cursor position to the end of the screen, type **Ctrl+Shift+Delete**. Move to the bottom of a window window by typing **Ctrl+End**.*

Executing Blocks of Commands Interactively

The ability to test code snippets interactively, in the **Command** window, can lead to tremendous productivity gains. Select the **Configure/Preferences/Cmd Window** tab to configure your interactive environment. Two frames, **Action on Enter** and **Output** will be discussed here.

Action on Enter has the following items:

1. **Execute current whole line.**

When the **Return** key is pressed, the entire line of code containing the cursor is executed, irrespective of the cursor position within the line. If the **Semi-colon enters multi-line** box is checked, lines with explicit semi-colons between the last **GAUSS** prompt and the current line are also executed. This is often called the **Active Block**.

2. **Execute if at end of line.**

Pressing **Return** will execute the line only if the cursor is at the end of the line. **Return** will split the line otherwise.

3. **Semi-colon enters multi-line.**

Check this box to have the ability to type multiple lines of code without having each line executed as you press **Return**. This is useful when writing loops or memory-resident procedures in the **Command** window.

Blocks of code may be run by selecting the block and:

- Selecting **Run Selected Text** from the Run Menu.
- Pressing **Ctrl+R**.
- Clicking the **Run Selected Text** button on the Toolbar.

4. THE GAUSS WINDOWS ENVIRONMENT

The **Output** frame lets you choose where output will appear when you execute code from the **Command** window. It is most relevant when executing code in the middle of a screen (by inserting a **GAUSS** prompt with Ctrl-I).

- **Insert**
Inserts output immediately after the execution point. This can result in overwriting existing text on your screen.
- **Append**
Inserts output at the end of the screen.
- **Relocate on next line**
Inserts output on the line following the execution point.

4.3.1 Exercises

- 4.1 Type some simple **GAUSS** commands in the **GAUSS** Command window. For example,

```
let x = 1 2 3 4
Print x
```

Note that explicit semicolons at the end of each line are not necessary in interactive mode.

- 4.2 Open an **Output** window with Ctrl+O and execute some commands from the **Command** window. Close the **Output** window and execute some commands.
- 4.3 Attempt to type the following into the **Command** window with the **Semi-colon enters multi-line** box unchecked.

```
i=1;
do while i<=10;
Print "i " i;
i=i+1;
endo
```

Check the box and try it again.

- 4.4 Use your mouse to select the above code and press Ctrl+R or the **Run Selected Text** button on the toolbar.

4.4 Running Commands from Files

The **GAUSS** for Windows environment distinguishes between two files, the **Active** file and the **Main** file. The **Active** file is often called the **Current** file.

The Active file is the one displayed in the **Edit** window, the one with focus. An **Active** file may also be the **Main** file.

A file must be saved to disk before it becomes the Active file.

Run the **Active** file or a selected block of code in the **Active** file by:

1. Selecting **Run Active File** on the Run menu. **GAUSS** automatically changes this to **Run Selected Text** if a block is selected.
2. Pressing Ctrl+R.
3. Clicking the **Run Active File** button on the Toolbar. **GAUSS** automatically changes this to **Run Selected Text** if a block is selected.

The Main File appears in the **GAUSS** list box on the toolbar. Make an **Active** file the **Main** file by either running it or by choosing the **Run/Set Main File** menu item.

Run the **Main** file by:

1. Selecting **Run Main File** on the Run Menu.
2. Pressing F5.
3. Clicking the **Run Main File** button on the Toolbar.

4.4.1 Exercises

- 4.5 Click the **New** button on the toolbar to open a new **Edit** window. Copy the code in exercise 4.3 to the newly opened **Edit** window (Use **Ctrl+C** followed by **Ctrl+V**).

*Find the code from exercise 4.3 in one of two ways. The first way is to simply scroll up in the **Command Window**. However, this code vanishes after you exit **GAUSS**. The second way is to copy the code from the **Command Log**. The **Command Log** contains a history of all **GAUSS** commands that have been executed, either from a file or from the command line. Get to it by clicking the **Open Existing File** button on the toolbar (or use the **File/Open** menu item). Click the **Wksp** folder and double click the **Command Log** file. This will put the **Command Log** into a **GAUSS Edit** window. Copy the code and paste it into your blank **Edit** window.*

4. THE GAUSS WINDOWS ENVIRONMENT

Notice that the **Run** and **Stop** toolbar buttons are disabled. The corresponding **Run** and **Stop** menu options under the **Run** menu are also disabled.

Save the file you've just created in a directory of your choice, calling it `foo.pgm`. Move focus to the **Command Window** and back to the **Edit** window. Notice that the toolbar buttons and menu options are now enabled.

Run `foo.pgm` by clicking the **Run Active File** button on the toolbar.

- 4.6 Select the code in the **Edit** window. Float your mouse over the same button as in the previous question. Notice that it now says **Run Selected Text**. Click the button again to run your code.

4.5 File Menu

Some useful items in the **File** menu are **Change Working Directory**, **Clear Working Directory List**, and **Recent Files**.

Change Working Directory may also be implemented using a **chdir** or **changedir** command or by selecting a directory from the working directory toolbar.

The ten most recent files opened are displayed at the end of the **File** menu. If the file you want to open is on this list, click it and **GAUSS** opens it in an **Edit** window.

4.6 Edit Menu

Edit window items and associated keyboard shortcuts are:

Undo	Ctrl+Z
Redo	Ctrl+A
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Select All	
Clear All	
Find	Ctrl+F
Find Again	F3
Replace	Ctrl+Alt+F3
Insert Time/Date	
Go to Line	Ctrl+G
Go to Next Bookmark	F2
Toggle Bookmark	Ctrl+F2
Edit Bookmarks	
Record Macro	Ctrl+Shift+R
Clear Macros	

4.6.1 Bookmarks

Bookmarks enable quick and easy cursor movement to particular lines or sections of code.

To add a bookmark, place the cursor in the line you want to bookmark and press Ctrl+F2 or click the Toggle Bookmark item on the **Edit** menu.

Cycle forward through all bookmarks by pressing F2. Cycle backwards through all bookmarks with Shift+F2.

The Edit Bookmarks window lets you add, remove, name, or jump to a particular bookmark. This window is opened when you select Edit Bookmarks from the **Edit** window.

Margin display must be turned on to see bookmarks. Use the Misc tab in the Configure/Editor Properties menu to turn margins on and off.

4.6.2 Macros

GAUSS will save up to 10 separate keystroke macros.

Create a macro by pressing Ctrl+Shift+R or by clicking the Record Macro item on the **Edit** menu. Press the keystrokes you want recorded (only keystrokes in the active window are recorded, not mouse movements). Stop recording by clicking the Stop button on the Macro dialog. Select a name for your macro.

Macros are not saved when you close **GAUSS**.

4.7 View Menu

The **View** menu lets you toggle the display of the **Main** Toolbar, the Status Bar, the Working Directory Toolbar, and the Debug Toolbar. These toolbars may be dragged and dropped anywhere on the screen.

4.8 Configure Menu

Use the **Configure** menu to set how you want **GAUSS** to look and operate.

4. THE GAUSS WINDOWS ENVIRONMENT

4.8.1 Preferences

Clicking **Configure/Preferences** brings up a tab dialog with six tab pages.

1. Run Options

Configurable Run Options center on whether dataloop translation and line number tracking are turned on. Dataloop translation may also be activated from the `gauss.cfg` file or from the Run menu.

- (a) **Dataloop Translator**
Toggles on/off the translation of a file using dataloop. The translator is not necessary for **GAUSS** program files not using dataloop.
- (b) **Translator line number tracking**
Toggles on/off execution time line number tracking of the original file before translation.
- (c) **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.

2. Compile Options

Configurable compile options include library settings and trace options. The default library setting (autoload, autodelete, and the **GAUSS** and User libraries always active) are sufficient for nearly all development.

Library Frame

- (a) **Autoload**
Toggles on/off the autoloader. The autoloader resolves references to symbols that are not defined in the program that references them. The search path used by the autoloader is first the current directory, then the paths in the `src_path` variable, in the order they appear. If the autoloader is off, no forward references are allowed. Every symbol 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**.
- (b) **Autodelete**
Toggles on/off autodelete. Autodelete controls the handling of references to unknown symbols when autoload is on. If autodelete and autoload are on, the following search path is used to locate symbols not previously defined:
 - user library
 - user-specified libraries
 - gauss library
 - current directory, the the `src_path`

4. THE GAUSS WINDOWS ENVIRONMENT

If autodelete is off and autoload is on, the the following search path is used to locate symbols not previously defined:

- user library
- user-specified libraries
- gauss library

Use an **external** statement for anything referenced above its definition if autodelete is off, i.e.

```
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 symbol table.

- (c) **GAUSS Library**
Toggles on/off the **GAUSS** library
- (d) **User Library**
Toggles on/off the user library
- (e) **Declare Warnings**
Toggles on/off declare warning messages during compiling.

Compiler Trace Frame

Setting compiler tracing whether by file, line, or symbol, can be quite useful in debugging. Compiler tracing may also be set at run-time with the **trace** command.

- (a) **Off**
Turns off the compiler trace function.
- (b) **File**
Traces program file openings and closings.
- (c) **Line**
Traces compilation by line.
- (d) **Symbol**
Creates a report of procedures and the local and global symbols they reference.

4. THE GAUSS WINDOWS ENVIRONMENT

3. Cmd Window

Action on Enter has the following items:

- (a) **Execute current whole line.**
When the **Return** key is pressed, the entire line of code containing the cursor is executed, irrespective of the cursor position within the line. If the **Semi-colon enters multi-line** box is checked, lines with explicit semi-colons between the last **GAUSS** prompt and the current line are also executed. This is often called the **Active Block**.
- (b) **Execute if at end of line.**
Pressing **Return** will execute the line only if the cursor is at the end of the line. **Return** will split the line otherwise.
- (c) **Semi-colon enters multi-line.**
Check this box to have the ability to type multiple lines of code without having each line executed as you press **Return**. This is useful when writing loops or memory-resident procedures in the **Command** window.

Blocks of code may be run by selecting the block and:

- Selecting **Run Selected Text** from the Run Menu.
- Pressing **Ctrl+R**.
- Clicking the **Run Selected Text** button on the Toolbar.

The **Output** frame lets you choose where output will appear when you execute code from the **Command** window. It is most relevant when executing code in the middle of a screen (by inserting a **GAUSS** prompt with **Ctrl-I**).

- **Insert**
Inserts output immediately after the execution point. This can result in overwriting existing text on your screen.
- **Append**
Inserts output at the end of the screen.
- **Relocate on next line**
Inserts output on the line following the execution point.

The output buffer size is set in the **Performance** frame. The default print buffer in **GAUSS 5.0** is larger than in previous versions of **GAUSS**. This speeds execution since screen I/O takes considerable time. Reduce the buffer size or explicitly flush the print buffer if you want output to occur more frequently (e.g. when printing optimization steps). However, be aware that a smaller print buffer can significantly increase execution time.

Explicitly flush the print buffer by inserting

```
print /flush;;
```

4. THE GAUSS WINDOWS ENVIRONMENT

after print statements that you want immediately sent to the screen. The double semicolon ensures that flushing the buffer does not alter the format of the output.

Running a program in a DOS compatibility window is another way to ensure that output occurs more frequently. All output in a DOS compatibility window is unbuffered. Unbuffered output is useful when viewing the intermediate results of an optimization program.

4. DOS Compatibility

The font used in the DOS Compatibility Window may be changed here.

5. Misc.

This tab lets you choose how often autosave is invoked.

4.8.2 Editor Properties

Set numerous editor properties by clicking the Editor Properties menu item. You can customize the formatting of your code and text by changing font colors, fonts, and add line indentations and line numbering to your programs.

1. Color/Font

Color Specifies the way syntax coloring works in the editor.

Font Specifies what font the edit window will use.

2. Language/Tabs

Auto Indentation Style Specifies how the autoindenter will indent your code.

Tabs Specifies how many spaces a tab has.

Language Specifies what syntax the GAUSS editor recognizes for coloring.

Fixup Text Case While Typing Language Keywords Specifies whether the editor will automatically change the case of GAUSS keywords when they use the wrong case.

3. Misc

Smooth Scrolling Enables or disables smooth scrolling when the window is scrolled up/down by one line or left/right by one character.

Show Left Margin Enables or disables the editor's margin. The margin is used for showing breakpoints, bookmarks, or line numbers.

Line Tooltips on Scroll Shows the first line number on screen as a tooltip as you scroll up and down the file.

4. THE GAUSS WINDOWS ENVIRONMENT

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

Make sure the Confine Caret to Text box is checked. It ensures that your cursor will be at the beginning of a line in the Command window, rather than where it's placed.

4.9 Run Menu

Run Menu items are:

- **Insert GAUSS Prompt - Ctrl+I**
This is useful when editing in the middle of a screen, to insert a **GAUSS** prompt and execute code following the prompt.
- **Insert Last Command - Ctrl+L**
GAUSS keeps commands entered in a buffer. Ctrl+L lets you select from a history of the most recent commands.
- **Run Selected Text - Ctrl+R**
Runs a selected block of code.
- **Run Active File - F6**
Runs the Active File, the one with focus.
- **Test Compile Active File - Ctrl+F6**
This is useful to see whether all symbol references are resolved. It does not create a file on disk.

4. THE GAUSS WINDOWS ENVIRONMENT

- **Run Main File - F5**
Runs the Main file.
- **Test Compile Main File - Ctrl+F5**
This is useful to see whether all symbol references are resolved. It does not create a file on disk.
- **Edit Main File - F7**
You can also edit the Main File by typing `edit filename;` from a **GAUSS** prompt or by clicking the **Edit Main File** toolbar button.
- **Stop Program**
GAUSS must complete executing the current procedure before it stops executing. Often, in the case of optimization programs, this requires considerable time. The only shorter solution is to close **GAUSS** using the Windows Task Manager.
- **Build GCG File from Main**
Compiles the Main file and saves the compiled file to disk, with a `.gcg` extension. The resulting compiled file may be freely distributed. Users without **GAUSS** may run it using the **GAUSS** Run-Time module.
- **Set Main File**
Turns the Active File into the Main File and inserts the filename into the Main File list box.
- **Clear Main File List**
Clears the Main File list box.
- **Translate Dataloop Commands**
This must be checked if **dataloop** is being used.

4.10 Debug Menu

The **GAUSS** debugger is started from the **Debug** menu, by clicking **Debug Main File** or **Debug Active File**. Debugging of the main file can also be started by clicking the **Debug Main File** icon in the **GAUSS** toolbar. This requires that the Main file be set and appearing in the drop down **Main File** list box on the toolbar.

When the debugger starts, several items in the **Debug** Menu are activated and a **Debug** toolbar appears on your screen. The **Debug** toolbar has buttons for some of the activated **Debug** menu items. The activated **Debug** menu items include:

4. THE GAUSS WINDOWS ENVIRONMENT

- **Set/Clear Breakpoints - F9**
Enables or disables a line breakpoint.
- **Edit Breakpoints**
Allows for advanced breakpoint management, including displaying breakpoints and adding filename, procedure, line, and cycle breakpoints.

Often a main program has loops that execute code in different files a number of times. Debugging is considerably simplified with the ability to set breakpoints for a given cycle of the loop, and in a given file on a given line number or in a given procedure.

Breakpoints remain valid when a file is changed and lines are inserted or deleted, but are lost when a file is closed, whether it is saved or not.
- **Clear All Breakpoints**
Removes all line and procedure breakpoints from all open files.
- **Go - F5**
Runs to the next active breakpoint or, if no breakpoints are active, to the end of the file.
- **Stop**
Stops debugging.
- **Step Into - F8**
Runs the next executable line of code in the application and steps into procedures.
- **Step Over - F10**
Runs the next executable line of code in the application but does not step into procedures.
- **Step Out - Ctrl+F10**
Runs the remainder of the current procedure and stops at the next line in the calling procedure. Step out returns if a breakpoint is encountered.
- **Set Watch**
Set a watch to see how a variable's value changes while debugging. Watches can be set on matrices, strings, string arrays, structures, and arrays. Individual matrix elements can be changed during the debugging session and loaded into the symbol table by clicking **Matrix/Save**.

Setting a watch opens a **Matrix Editor** window. The **Matrix/Auto-reload** and **View/Stay on Top** menu items are automatically selected.

4. THE GAUSS WINDOWS ENVIRONMENT

View/Stay on Top and Matrix/Auto-reload are not the default selections when a matrix is loaded using Ctrl+E, the Matrix Editor

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

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

*The **trace** command is another powerful debugging tool. You can trace line numbers and procedure calls.*

4.11 Tools Menu

The **Tools** menu contains four items.

- **Matrix Editor - Ctrl+E**

Open a matrix, string, string array, or structure in the **Matrix Editor** by double clicking it and typing Ctrl+E.

The **Matrix Editor** window menu has four items:

1. **Matrix**
The **Matrix** menu lets you load, reload, auto-reload, and save matrices after they have been edited. You might change some of the entries in a matrix using the **Matrix Editor**. Clicking the **Save** menu item will update the matrix's definition in the **GAUSS** Symbol Table.
2. **Format**
The **Format** menu lets you display a matrix, string, or string array in Decimal, Scientific, Hexadecimal, and Character representations. You can also choose the precision displayed.
3. **Edit**
The **Edit/Preferences** menu item lets you adjust the size of the **Matrix Editor** cells. You also have the ability to choose formats, precision, and how pressing a key navigates through the matrix.
4. **View**
The **View** menu lets you control the **Matrix Editor** window and whether you want real or imaginary parts of a matrix displayed. **Minimal View** minimizes the amount of screen space occupied by the **Matrix Editor**. This is useful when setting watch variables.

*Choose **Stay on Top** if you've set a watch while debugging. This ensures that you don't have to change window focus with each step through a program.*

4. THE GAUSS WINDOWS ENVIRONMENT

- Source Browser

The **Source Browser**, invoked with **Ctrl+B** or by clicking the **Source Browser** toolbar button, lets you quickly find, view, and if necessary, modify source code. The **Source Browser** also lets you search for external symbols in active libraries.

The **Source Browser** is especially useful when you don't know the name of a procedure but know what you want accomplished. For example, open the **Source Browser** and type **seemingly unrelated**. You will see a list of all lines in the **src_path** files matching this expression.

- Lib Tool

The **Library Tool** lets you quickly manage your libraries, giving you the functionality of the **lib** command. You can add and remove libraries, add and remove files within a library, and add and remove paths to filenames within a library file.

- DOS Compatibility Window

Click this menu item to turn on the **DOS Compatibility Window**. Unbuffered output is sent to this window when it is open. Unbuffered output (also configurable by setting the buffer size to zero in **Configure/Preferences/ Cmd Window** is useful when viewing intermediate results from an optimization program.

4.12 Window Menu

The Window Menu items are used to arrange windows to fit your preferences. Tiling the **Command** and/either the **Edit** and/or **Output** windows horizontally or vertically is very useful when developing programs. Horizontal tiling is chosen when the output occupies line lengths are long and vertical tiling is chosen when the output line lengths are short. The active window is one with focus. Toggle focus between all open windows using **Ctrl+N** or clicking in the window you want active. All open windows are listed at the bottom of the **Windows** Menu.

GAUSS will automatically tile the input (a **Command** or **Edit** window) and output (a **Command** or **Output** window) windows when **Dual Vertical** or **Dual Horizontal** is selected.

Switch focus to the next open window by typing **Ctrl+N**.

4.13 Help Menu

Section 3.2 discussed the Help Menu.

4.14 Exercises

- 4.8 **GAUSS** has several examples in the `examples` subdirectory. Open `ols.e` in an Edit window (either set the current directory to the `examples` subdirectory using `chdir` and type `edit ols.e` or type `edit pathname\ols.e`. Run the file, sending the results to an Output window. Set focus on the Output window. Select the coefficient estimates and standard error columns and copy them to another file in another open Edit window (activate the ability to select columns from Configure/Editor Properties/Misc).
- 4.9 Step into `ols.e` using the debugger. Notice the cursor location changing in the status bar as you step into the code.
- 4.10 Stop the debugger. Start it again, debugging the `ols.e` file. Step over the code, rather than into the code. Notice the differences.
- 4.11 Stop the debugger. Start it again, debugging the `ols.e` file. Run to the `ols` procedure.
- 4.12 Stop the debugger. Start it again, debugging the `ols.e` file. Set a line breakpoint (with F9) in the `ols` procedure (in the `ols.src` file) and run the program to the breakpoint.
- 4.13 Stop the debugger. Start it again, debugging the `ols.e` file. Search the `ols.src` file for `k = diag(cov)`. Set a breakpoint on this line and another breakpoint on another line (be careful that the breakpoint is not in an `if`, `else`, `endif` set of commands). Run to the breakpoints.

*Search for `k = diag(cov)` two ways, by using the Source Browser, followed by double clicks on each match, and by using the **GAUSS** editor Find command.*

- 4.14 Create another file, say `foo.prg` (either type `edit foo.prg` from the **GAUSS** prompt or click the **New** button on the toolbar). Type the following into `foo.prg`:

```
new;
cls;
i = 1;
do while i <= 6;
    print "The value of i is " i;
    i = i + 1;
endo;
```

*The **new**; and **cls**; commands are often put at the top of program files. They clear the symbol table and screen respectively.*

Save the file and debug it. Set a watch on `i` and run to line three, the fourth time the loop is executed.

*Often the watch window occupies too much real estate. Resize it and move it using the mouse to fit your preferences. You can also choose **View/Minimal View** from the Matrix Editor/Watch window and resize the result.*

4. *THE GAUSS WINDOWS ENVIRONMENT*

- 4.15 The same as 4.14 except, step through the code after the watch is set, instead of running to the breakpoint. Notice how the watch window changes.

4. *THE GAUSS WINDOWS ENVIRONMENT*

Chapter 5

Data I/O

GAUSS reads and writes ASCII datasets, **GAUSS** datasets, **GAUSS** matrix (.fmt) and string (.fst) files, and numerous spreadsheet formats.

5.1 GAUSS Data Sets

GAUSS datasets are written in a special format to a file with a .dat extension and contain data and variable labels.

Many **GAUSS** procedures are optimized for **GAUSS** datasets. For example, calculations are performed in chunks when a **GAUSS** dataset is passed to the **ols** procedure. A certain number of rows are read from the dataset, calculations are performed (e.g. sums of squares are calculated), and more rows are read. Reading the data in chunks often results in faster execution. All the calculations are performed at once when the data are passed in as a matrix.

5.1.1 Writing GAUSS Data Sets

GAUSS datasets may be created from matrices in memory using the **saved** command, from programs using the **create** and **writer** commands, and from ASCII data files using the ATOG ASCII-to-**GAUSS** data conversion utility.

Saving Data Stored in Memory

Data already in a matrix in memory are saved to a **GAUSS** dataset using the **saved** function.

```
x = { 1 1 1, 1 2 4, 1 3 9, 1 4 16, 1 5 25 };
lbl = { X1, X2, X3 };          @ These are the variable names @
call saved(x,"test",lbl);
```

Saving Data Generated Within A Program

Often programmers wish to write to **GAUSS** datasets sequentially, generating rows of the dataset in code. This is accomplished using the **create** and **writer** commands.

The **create** command creates and opens a **GAUSS** dataset. It has two formats:

```
create fh = fname WITH vnames,col,typ;
create fh = fname USING comfile;
```

Input:	fname	literal or ^string, the name of the file to be opened. A path can be included. If no extension is supplied, .DAT is assumed.
	vnames	literal or ^string or character matrix. vnames works in conjunction with col to determine the variable names to be given the columns in the dataset.
	col	scalar, the number of columns in the dataset. col works in conjunction with vnames.
	typ	scalar, the precision used to store the data. typ can be 2, 4 or 8, indicating the number of bytes per element.
	comfile	literal or ^string, the name of the command file containing the information needed to create the dataset. If no extension is supplied, .GCF is assumed.

The following program generates a matrix of random numbers ten times. Each time the matrix is saved to a dataset.

```
vnames = { Y, X1, X2, X3 };          @ The variable names @
create fout = regsim with ^lbl,0,8;  @ Create the file handle fout @
load state1 = state;                @ load a random number state @
i = 1;
do until i > 10;
    {x, state1} = rndKMn(100,4,state1); @ Create the x matrix @
    call writer(fout,x);              @ Write the matrix x to fout @
    i = i + 1;                        @ Add to the loop index @
enddo;
fout = close(fout);                 @ Close the file handle @
```

5. DATA I/O

ATOG - Writing Directly from ASCII Files

Often ASCII data files are quite large and cannot be contained in available memory. There are two ways to read such data, using the **fgetsat** functions (discussed below) and using **ATOG**. **ATOG** is a separate executable that is run from a command prompt, outside of **GAUSS**.

*Remember to run **ATOG** outside of **GAUSS**. An error will occur if you try to run it within **GAUSS***

This example uses **ATOG** to generate a **GAUSS** dataset.

1. Create an ASCII file containing a 8×3 matrix of random numbers. Call this file **test.asc**.
2. Create another ASCII file (click the **Open new file** button on the **GAUSS** toolbar). Put the following statements into this new file.

```
INPUT test.asc;           @ Name of the input file @
OUTPUT test;              @ Name of the output dataset @
INVAR X1 X2 X3;          @ Names of the input variables @
OUTVAR X1 X2 X3;         @ Names of the output variables @
```

Save this file as **test.cmd**.

3. Create the **GAUSS** dataset by executing the **GAUSS** utility, **ATOG**, with the name of the command file as an argument. This may be done either from inside **GAUSS** using the **dos** command, or from an OS prompt. From the OS prompt, enter:

```
atog test
```

4. Notice that your working directory now has the **GAUSS** dataset **test.dat**.
5. Verify that **test.dat** contains the correct data (use **loadd**).

5.1.2 Reading **GAUSS** Data Sets

loadd loads a **GAUSS** dataset in its entirety. The following command loads the **GAUSS** dataset **temp.dat**:

```
x = loadd("temp");
```

getname retrieves **GAUSS** dataset variables names and puts them into a **GAUSS** character matrix.

```

dataset = "c:\\gauss40\\examples\\freqdata";
z = loadadd(dataset);
PRINT meanc(z);

```

```

      .
1.9990000
      .
1.5007800

```

```

lbl = getname(dataset);
$lbl;

```

```

AGE
PAY
sex
WT

```

readr is used to retrieve a subset of the observations in a dataset. Before **readr** is invoked, the dataset must be opened using the **GAUSS open** command. **open** assigns a “file handle”, i.e., a name or label, to the dataset. The file handle is used by **readr** as well as by other commands.

The following is a **GAUSS** program for computing means and a covariance matrix from selected columns of a dataset.

```

dataset = "c:\\gauss40\\examples\\freqdata";
sel = { 1, 2 };
mmy = 0;
smy = 0;
nobs = 0;
nb = 10;

open fin = ^dataset for read;

do until eof(fin);
  y0 = readr(fin,nb);
  y = packr(y0[. ,sel]); /* remove obs with missing data */
  mmy = mmy + y'*y;
  smy = smy + sumc(y);
  nobs = nobs + rows(y);
endo;

mn = smy/nobs; /* means */
vc = mmy/nobs - mn*mn'; /* covariance matrix */

format /rdn 10,4;
PRINT "number of observations " nobs;

```

5. DATA I/O

```
PRINT;
PRINT "number of missing observations" rowsf(fin)-nobs;
PRINT;
PRINT "means" mn';
PRINT;
PRINT "covariance matrix";
PRINT vc;
fin = close(fin);
```

The `~` before `dataset` in the `open` statement indicates that the symbol `dataset` contains a string with the name of the file rather than being the name of the file itself. If the program were being executed in the directory `examples`, or if that directory is listed in the `PATH` environment string, the following statement would work as well:

```
open fin = freqdata for read;
```

Since `freq` is not preceded by `~`, **GAUSS** assumes that it is the name of the file rather than being a string containing the filename.

The computed covariance matrix uses *listwise* deletion of missing data; any row with missing data is deleted by the `packr` function. The `rowsf` function takes a file handle as an argument and returns the number of rows of the dataset associated with that file handle.

The `nb` parameter, set to 10 in the program, determines the number of rows to be read in at a time by `readr`. This parameter controls a trade-off between time and RAM in the reading from the dataset. Setting `nb = 1` causes `readr` to read one observation at a time, thereby reducing the amount of memory required to read the data in but increasing the overall time to compute the covariance matrix. On the other hand, setting `nb` to a larger number would speed up the overall computations but would increase the memory demands, perhaps beyond the capacity of the computer.

GAUSS includes a special function that chooses the largest value for `nb` subject to the available memory.

```
open fin = c:\gauss40\examples\freqdata for read;
nr = getnr(6, colsf(fin));
PRINT nr;

4000.0000
fin = close(fin);
```

The first argument in the call to `getnr` is a fudge-factor roughly interpreted as the number of copies of the matrix that need to be stored. Larger numbers are safer. The second argument is the number of columns in the dataset. The result is the number of rows that can be safely read in, given the currently available RAM.

Setting the pointer with seekr

seekr is used to re-read a **GAUSS** dataset that has already been read, or to select rows for reading. When a dataset row, or set of rows, has been read by **readr**, a “pointer” is set to the beginning of the following row. The next time **readr** is called it will read from that pointer. **seekr** is a function for moving the pointer.

If you wish to re-read a dataset from the beginning,

```
open fin = c:\gauss40\examples\freqdata for read;
call seekr(fin,1);
fin = close(fin);
```

will set the pointer back to the beginning. A subsequent call to **readr** will read rows from the first row.

It is also possible to use **seekr** to read selected rows from a dataset. For example:

```
dataset = "c:\gauss40\examples\freqdata";

sampleSize = 10;
x = {};
rndseed 34567;

open fin = ^dataset for read;

i = 1;
load state1 = state;          @ loads a random number generator state @
do until i > sampleSize;
  {ir,state1} = rndKMu(1,1,state1);
  ir = ceil(rowsf(fin)*ir);
  call seekr(fin,ir);
  x0 = readr(fin,1);
  x = x|x0;
  i = i + 1;
endo;

format /rd 10,4;
PRINT x;
```

```
6.0000  3.0000  0.0000  1.6200
10.0000 1.0000  0.0000  1.3400
5.0000  1.0000  0.0000  1.4200
7.0000  3.0000  0.0000  1.2600
8.0000  1.0000  0.0000  1.7300
5.0000  3.0000  0.0000  1.4600
8.0000  2.0000  0.0000  1.2200
3.0000  3.0000  0.0000  1.2900
7.0000  3.0000  0.0000  1.6000
.      3.0000  0.0000  1.2100
```

5. DATA I/O

5.1.3 Using GAUSS Datasets

Two procedures, **datalist** and **makevars**, further enhance **GAUSS** dataset abilities.

datalist gives a nicely formatted display of a **GAUSS** dataset, with the variable names on top of their respective columns. Suppose `stocks1.dat` is a **GAUSS** dataset. A simple use of **datalist** is:

```
datalist stocks1;
```

makevars lets you use the **GAUSS** dataset variable names in computations, i.e. the names are assigned to their respective columns in a dataset. For example, the following puts all variable names in `stocks1.dat` into the workspace. Their values are the appropriate columns in the `temp.dat` matrix. It uses one of the names, `AMZN` in a calculation.

```
makevars(loadd("stocks1"), 0, getname("stocks1"));  
ssamzn = amzn'amzn;
```

5.1.4 Matrix and String Files

Matrix and string files (those with `.fmt` and `.fst` extensions) are stored in the same format as **GAUSS** datasets. However, they do not have the header variable name information.

Writing Matrix and String Files

The **save** command writes **GAUSS** matrices and strings to `.fmt` and `.fst` files. The matrix or string name will be the name of the `.fmt` or `.fst` file.

```
x = { 1 2 3, 4 5 6 };  
save x;                               @ writes x to x.fmt @  
  
s = "this is a string";  
save s;                               @ writes s to s.fst @
```

The **save** documentation discusses saving matrix files in a directory different from the working directory (the default save location may also be changed in the `gauss.cfg` file.)

Reading Matrix and String Files

The **load** command loads **GAUSS** matrices and strings (.fmt and .fst) files. The matrix or string name is the name of the .fmt or .fst file.

```
x = { 1 2 3, 4 5 6 };
save x;                               @ writes x to x.fmt @
load x;                                @ loads x.fmt @

s = "this is a string";
save s;                                 @ writes s to s.fst @
load s;                                 @ loads s.fst @
```

5.2 ASCII Files

5.2.1 Writing ASCII Files

ASCII datasets are created two ways, by printing a matrix to an auxiliary output file or by writing strings to a file handle using **fputs** or **fputst**.

Printing To An Output File

Output from all **print**, **printfm**, **printfmt**, and errorlog output may be written in ASCII format to an auxiliary output file. Create this file using the **output** command. Here are the three ways of calling **output**.

```
output file = filename on;
output file = filename reset;
output file = filename off;
```

The first call turns on output logging and appends output to the bottom of previously saved output. The second call overwrites **filename** with subsequent output. The third call turns off output logging.

- *The screen format governs the format of matrices printed to the output file. The **format** command changes this format.*
- *The default output width in the output file is 80 characters. This may be changed, to a maximum of 256 characters, using the **outwidth** command.*
- *Often **screen off** followed by **screen on** are used to stop output from appearing on the screen.*

5. DATA I/O

- *All typed commands will appear in the output file. Be sure to remove any extra command lines from the output file before using it*

Here is an example that uses output logging:

```
x = { 1 1 1, 1 2 4, 1 3 9, 1 4 16, 1 5 25 };
screen off;
output file = test.asc reset; // could also use reset on
PRINT x;
output off;
screen on;
```

Look at `test.asc` in the **GAUSS** Editor to verify that the data were written.

Using `fputs` and `fputst`

The **fputs** and **fputst** commands write strings to a file handle. The difference is that **fputst** inserts a newline (a carriage return/line feed if the file was opened in text rather than binary mode) at the end of each string written to the file.

File handles are opened using **fopen** (**fopen** has a number of options, including reading and writing sections of a file and reading and writing text or binary files.)

*Be sure to close file handles after they have been used, using either the **close** command or **closeall**.*

```
f1 = fopen(filename, "r");
f1 = fopen(filename, "w");
f1 = fopen(filename, "a");
```

The first line opens an existing file, `filename`, for reading. The second line opens `filename` for writing. If `filename` already exists, its contents are erased. The third line opens `filename` for appending.

Here's an example illustrating each command.

```
string s0 = { "ddd111", "1234asdf", "fasert324346", "uippjkui" };
f1 = fopen("test1.txt", "w");
fputst(f1, s0);
f1 = close(f1);
f2 = fopen("test2.txt", "w");
fputs(f2, s0);
f2 = close(f2);
```

Open `test1.txt` and `test2.txt` using the **GAUSS** editor. Note that newlines were added in `test1.txt` which was written using **fputst**, i.e., each string in `s0` is written to a separate line. Newlines were not added in `test2.txt`, i.e. the strings were concatenated into a single string. **fputs** is useful for creating files that will be treated as binary files.

Using `fseek` to set the file handle pointer

The file handle pointer is placed after the last line written. Position the pointer using `fseek`. When reading and writing text files, `fseek` should first return the pointer to the beginning of the file, followed by an `fseek` to the desired location.

5.2.2 Reading ASCII Files

Text (ASCII) files are read with the `load` command or with `fgets`, `fgetst`, `fgetsa`, and `fgetsat`.

Reading ASCII Files into Matrices

The `load` command is used to load text files into memory.

```
load x[5,3] = test.asc;
```

The dimensions of the matrix are specified in the `load` command. A matrix may also be loaded using empty brackets. This stores the data in a column vector, awaiting a `reshape` call.

Reading a text file into a string or string array

`fgets`, `fgetst`, `fgetsa`, and `fgetsat` read ASCII files into strings and string arrays. They are particularly useful for reading and writing data files that mix strings of characters with numbers, or for reading fixed length data without delimiters such as files generated by Fortran.

`fgets` and `fgetst` read single lines from files (or the number of characters specified less one or the end of the file, whichever comes first) and turn them into strings. `fgetsa` and `fgetsat` read files into string arrays. `fgets` and `fgetsa` add carriage return/line feeds at the end of each line read. `fgetst` and `fgetsat` do not add carriage return/line feeds at the end of the lines read.

1. The functions with an `a` read in a maximum number of lines.
2. Those without an `a` read in a maximum number of bytes.
3. The functions ending in a `t` delete newline characters before constructing string arrays.

5. DATA I/O

4. The functions that do not end in a **t** are useful for reading binary files where you don't want characters deleted.
5. The functions with the **t** are useful for reading text files into string arrays where you usually want newlines removed.

For example

```
f3 = fopen("test1.txt","r");
s3 = fgetsat(f3,100);

PRINT s3;

ddd111
1234asdf
fasert324346
uippjkui
```

Experiment reading this file using the different functions.

Using **fseek** to set the file handle pointer

*The file handle pointer is placed after the last line written. Position the pointer using **fseek**. When reading and writing text files, **fseek** should first return the pointer to the beginning of the file, followed by an **fseek** to the desired location.*

5.2.3 Working with ASCII Files

GAUSS datasets created either using **ATOG** or **writer** cannot handle variables with strings greater than 8 characters. The **GAUSS load** command also is not able to handle some strings with nonstandard characters, e.g. date variables in the format mm/dd/yy. An additional consideration is whether the file is too big to read in all at once.

The functions described in this section are capable of handling any type of contents of a data file. Using the **fgetsat** function you can read the data into memory a portion at a time. Unless you are certain that the entire file will fit into memory, it is best to read the data in a loop and process them one portion at a time.

Suppose a dataset has nonstandard dates and long character strings:

```
2/10/97  94.52  86450  Microsoft  Computer Software
2/12/97  82.15  102210 Cisco      Computer Hardware
.
.
.
```

This dataset cannot be loaded directly into **GAUSS** because the usual loading function won't handle the forward slashes in the date field, nor will they handle the long strings describing the type of company at the end of each row.

The **fgetsat** function reads all the data into a string array. The following functions turn the strings in the string array into the desired data:

parse, token	for parsing delimited strings
stof	translates a number in a string into a number in a matrix
strindx, strindx	finds the location of one string in another string
strlen	find length of string
strsect	extract substring from a string

With these tools it should be possible to read any dataset into memory.

For example, suppose we wish to read the above dataset above into **GAUSS** matrices. The following code reads the data in portions and creates five vectors containing the data.

```

fname = "data.asc";

if fileexists(fname) $== "";
    errorlog "ERROR: data.asc can't be found";
end;
endif;

dateVar = {}; /* initialize matrices where */
price = {}; /* where data will be stored */
volume = {}; /* for analysis */

string company = ""; /* store these as string arrays */
string companyDesc = ""; /* because we don't know how long */
/* they'll be */

string s0;

g0 = fopen(fname,"r");

do until eof(g0);

    s0 = fgetsat(g0,100); /* reads 100 rows of fname at */
                        /* a time into a string array */

    i = 1;
    do until i > rows(s0);

        s1 = s0[i]; /* get i-th string */

        { dstr, s1 } = token(s1); /* this extracts the date */

```

5. DATA I/O

```
/* converting the mm/dd/yy into the standard date */
/* format yyymmdd won't be easy because mm and dd */
/* may have one or two digits in their fields, so */
/* we will have to find out where the slashes are. */

m1 = strindx(dstr, "/", 1);
m2 = strindx(dstr, "/", m1+1);

_month = stof(strsect(dstr, 1, m1-1));
_day   = stof(strsect(dstr, m1+1, m2-m1+1));
_year  = stof(strsect(dstr, m2+1, strlen(dstr)-m2+1));

dateVar = dateVar | (_day + 1e2*_month + 1e4*(_year+1900))*1e6;

{ pricestr, s1 } = token(s1); /* get price datum */
price = price | stof(pricestr);

{ volumestr, s1 } = token(s1); /* get volume datum */
volume = volume | stof(volumestr);

{ compstr, s1 } = token(s1);
company = company $| compstr;

companyDesc = companyDesc $| s1;

i = i + 1;

    endo;

endo;

g0 = close(g0); /* close the file because we */
              /* won't need it anymore      */

/*
** code goes here to work with the data that has been
** read into the vectors
*/
```

5.3 Spreadsheets

5.3.1 Writing Spreadsheets

The **export** and **exportf** functions are used to write spreadsheet files. Supported formats include Lotus, Excel, Quattro, Symphony, dBase, and Paradox. The filename

extension specified by the user determines which spreadsheet format **GAUSS** writes. This example writes an Excel file.

```
fname = "test.xls";
names = { "A", "B", "C", "D" };
x = rndKMn(20,4,-1);
call export(x,fname,names);
```

5.3.2 Reading Spreadsheets

The **import** and **importf** functions are used to read spreadsheet files. Supported formats include Lotus, Excel, Quattro, Symphony, dBase, and Paradox. The filename extension specified by the user determines which spreadsheet format **GAUSS** reads.

5.4 Transforming Data Sets - Dataloop Translation

Variable transformation is an important part of data analysis. A dataloop is used to transform variables in a dataset. The “translator” must be activated to allow the translation of data loop commands.

A dataloop begins with **dataloop indata outdata**; and ends with **endata**;. The two **dataloop** arguments are the name of the input **GAUSS** dataset, **indata**, and the name of the output dataset, **outdata**. The statements that perform the transformations are entered between the **dataloop** and **endata** statements.

```
mv = {.};
dataloop freqdata newfreq;           @ freqdata is the 'indata' set and
                                     newfreq is the 'outdata' set @

extern mv;
code TEEN with
  0 for AGE >= 21,
  1 for AGE < 21,
  mv for AGE $== mv;

recode SEX with
  1 for sex $== "M",
  0 for sex $== "F",
  mv for sex $== mv;

keep TEEN SEX PAY AGE;
endata;
```

code creates a new dichotomous “dummy” variable, TEEN, based on AGE. **recode** recodes sex from a character variable to a numeric variable. Use **extern** to reference global variables defined outside the data loop. The **keep** statement defines the list of variables that are kept in the transformed dataset.

5. DATA I/O

5.5 Exercises

- 5.1 Open the FREQDATA **GAUSS** dataset (it ships with **GAUSS**. The default installation location for `freqdata.dat` is the examples subdirectory of the **GAUSS** installation directory) and read and list the first ten observations to the screen.
- 5.2 Read in the 100th observation and print it to the screen (it will look like this: 3 3 +DEN 1.03). The third column contains character data, leading to the +DEN entry.
- 5.3 Store the variable labels in a character vector. Print the character vector to the screen. You should see AGE PAY sex WT.

Chapter 6

Symbols, Procedures, Global Variables, and Libraries

Chapter 2 briefly discussed the **GAUSS** environment. This chapter expands on that discussion, emphasizing user-defined procedures and libraries and how **GAUSS** uses them. The symbol table, containing references to variables, procedures, keywords, and functions is first explained. The chapter then discusses procedures, emphasizing how **GAUSS** finds procedure names in `.src` files and libraries (Intricacies of the code in the examples will be discussed in the next chapter. For now simply type the commands and run them.)

Procedures written and run in the **Command** window reside in memory, in the symbol table. Procedures written to a file reside in the main file (the command file) or in a `.src` file. The names of procedures written to a `.src` file must be found and saved in the symbol table before they can be run.

Library files tell **GAUSS** where to find these procedure definitions. **GAUSS** looks for a procedure definition by first searching the **GAUSS** list of intrinsic procedures, next determining whether the procedure name is in the symbol table, and finally whether it is referenced in a library `.lcg` file. Library files contain procedure names and names of the `.src` files which has the code defining them.

Procedures use global and local variables. Global variables must exist in the symbol table prior to being used. The **GAUSS** compiler learns of their existence in a `.ext` file and their initial values are set in a `.dec` file.

The Library Tool and **lib** command catalog procedure names (from `.src` files) and global variable names (from `.dec` files) in `.lcg` files. Section 6.2.1 shows how to build and update library files using the Library Tool and **lib** command.

6.1 Procedures

Users may create their own procedures and catalog them as they wish, in either the **User** library which is always available to calling programs, or in their own libraries. A user-defined library is made available to the **GAUSS** run-time environment using the **library** command.

A procedure definition consists of five parts:

1. Procedure declaration: **proc** statement
2. Local variable declaration: **local** statement. These are variables that exist only when the procedure is executing. They cannot conflict with other variables of the same name in your main program or in other procedures.
3. Body of procedure
4. Return from procedure: **retp** statement
5. End of procedure definition: **endp** statement

Here is an example of a **GAUSS** procedure with one return, **sqrtinv**.

```
proc (1) = sqrtinv(x);           @ procedure declaration
                                could also be: proc (1) = sqrtinv(x); @
    local y;                    @ local variable declaration @
    y = sqrt(x);                @ body of procedure @
    retp(y+inv(x));             @ return from procedure @
endp;                           @ end of procedure @
```

Saving **sqrtinv** in an **src** file and cataloging the **.src** in a library is way to ensure that **sqrtinv** is available for reuse.

- Make a new folder underneath the `c:\(gaussdir)\src` directory (where `(gaussdir)` is the installation directory for **GAUSS**). Call it **usersrc**.

*Many programmers use DOS to make new folders. It's easy to open a DOS window in **GAUSS**. Simply type **dos;** from the **GAUSS** prompt.*

- Open an Edit window and put the above **sqrtinv** code into it. Either type it directly or copy it from your previous typing or from the Command Log. Save the file as `c:\(gaussdir)\src\usersrc\myprocs.src`.
- Edit the `gauss.cfg` file so that `c:\(gaussdir)\src\usersrc` is in the **src_path**.

*Other procedures may be added to this file. **GAUSS** will compile all the procedures in the file even if only one is called.*

6. SYMBOLS, PROCEDURES, GLOBAL VARIABLES, AND LIBRARIES

The `sqrtinv` procedure is not yet ready for use. It must be catalogued in a library. This is discussed below.

Here is an example of a procedure that returns two variables, the passed in matrix and the result:

```
proc (2) = sqrtinvA(x);           @ procedure declaration @
  local y;                       @ local variable declaration @
  y = sqrt(x);                   @ body of procedure @
  retp(x, y+inv(x));             @ return from procedure @
endp;                             @ end of procedure @
```

It is also possible for a procedure to have no returns:

```
proc (0) = outprt(x,labels);
  PRINT "program output";
  format /rd /m1 10,8;
  PRINT $labels';
  format 10,4;
  PRINT x;
endp;
```

A procedure is called the same way as an intrinsic function.

```
{zed, state1} = rndKMn(3,3,-1); @ this statement defines the
                                input argument to the procedure @
zsi = sqrtinv(zed);             @ this statement calls the procedure @
```

The procedure with two returns would be called as:

```
{zed, state1} = rndKMn(3,3,-1); @ this statement defines the
                                input argument to the procedure @
{ ret1, zsi } = sqrtinvA(zed);  @ this statement calls the procedure @
```

6.1.1 Writing a Procedure in the Command Window

Procedures may be written in the `Command` window. They are put into the symbol table when run, available for use.

Let's write a small procedure for calculating the product of two matrices. Ensure that the `Semi-colon enters multi-line box` is checked on the `Configure/Preferences/Cmd Window` tab page.

6. SYMBOLS, PROCEDURES, GLOBAL VARIABLES, AND LIBRARIES

```
proc (1)= mply(in1,in2);
  local result;
  result=in1*in2;
  retp(result);    @ This could also be retp(in1*in2); @
endp;
```

Type **show** before running the procedure and you will see that it's not in the symbol table. Add it to the symbol table by running it. One way to run the code is to select it and press Ctrol-R. Another way is to select **Run Selected Text** from the **Run** menu. Type **show** again at the command prompt to verify that **mply** is a procedure in the **GAUSS** symbol table, available to any programs that call it.

The two **mply** arguments must be defined to use the procedure. Define the x matrix as a 5 by 5 matrix of random values (with either **rndKMn** or **rndKMmu** and y as a 5 by 1 column vector of random values. Multiply x and y using **mply**. Run the following from the Command window:

```
{x, state1} = rndKMn(5,5,-1);
{y, state1} = rndKMn(5,1,state1);
PRINT mply(x,y);
```

6.1.2 SRC Files

User-defined procedures may be written from the **GAUSS** command line and placed into the symbol table, as in the above example. However, these procedures vanish when **GAUSS** is closed. It is more common to put procedures into files so that they are available to future sessions of **GAUSS**

The most common way of saving procedures to disk is to put them into **.src** files. The **.src** files are catalogued into libraries. Here is an example of an **.src** file, **norma.src**, containing two procedures.

```
/*
** norma.src
**
** This is a file containing the definitions of two
** procedures returning the norm of a matrix x.
** The two norms calculated are the 1-norm and the
** inf-norm.
**/

proc onenorm(x);
  retp(maxc(sumc(abs(x))));
endp;
```

6. SYMBOLS, PROCEDURES, GLOBAL VARIABLES, AND LIBRARIES

```
proc infnorm(x);
    retp(maxc(sumc(abs(x'))));
endp;
```

6.1.3 Global Variables in External Procedures

Often procedures reference global variables, variables that exist in the global symbol table. Each procedure has its own local symbol table which vanishes after the procedure returns. The global variables in the main symbol table can be seen using the **show** command.

Global variables that do not already exist in the symbol table must be declared and initialized prior to being used. The compiler learns that global variables exist from **external** statements, usually in `.ext` files. These files tell the compiler that a variable exists and that it is external to the procedure. For example, suppose a collection of procedures in a file called `myprocs.src` use a global variable, `_myprocs_x`. The `myprocs.ext` file contains:

```
external matrix _myprocs_x;    @ convention is to precede globals with "_" @
```

Each entry in the `.ext` file must be initialized in a corresponding `.dec` file. A `.dec` file is where values are assigned to global variables at compile time, using **declare** statements, `.`. The default initialization value for matrices is zero. The default initialization value for strings is a null string. For example, the initialization statement for the `_myprocs_x` variable looks like this:

```
declare matrix _myprocs_x;    @ This takes the default value of zero @
```

Suppose there are two matrices and that you want to explicitly assign values to them at compile time. Your `.dec` file will contain the following lines:

```
declare matrix _myprocs_x = 3;
declare matrix _myprocs_y = { 1.2, 3, -1 };
```

6.2 Libraries

GAUSS libraries let users place their procedures into logical categories.

The **library** command makes libraries active. The **gauss** and **user** libraries are always active, unless turned off from the **Configure/Preferences/Compile Options** tab page.

Library statements are not cumulative, i.e. subsequent library statements replace completely previously activated libraries (except for **gauss** and **user**). For example, this statement opens the **dstat** library, an applications module which contains functions for the description of data.

6. SYMBOLS, PROCEDURES, GLOBAL VARIABLES, AND LIBRARIES

```
library dstat;
```

This statement opens the Optimization and Maximum Likelihood libraries, replacing the Descriptive Statistics library.

```
library dstat, optmum, maxlik;
```

The **library** command, without an argument, returns a list of the active libraries.

A library is a dictionary of source files and the source files contain symbol definitions. Typically a user will write a procedure in the **Edit** window, save it in a **.src** file, and catalogue the **.src** file in a library. The latter step is known as building the library. One **.src** file may hold many procedures. In addition, the user may create many **.src** files, each corresponding to a different purpose.

GAUSS libraries contain references to global symbols, i.e. procedures, keywords, functions, matrices, and strings. Global variables used in procedures require explicit declaration, in **.ext** and **.dec** files. They allow for re-usability. Library files are in the **lib** subdirectory. Here, as an example, is a section from the time-series cross section library file, **tscs.lcg**:

```
/*
**
**-----**-----**-----**-----**
**-----**-----**-----**-----**/

tscs.dec
    _tsmodel           : matrix
    _tsstnd            : matrix
    _tsmeth            : matrix
    _tsise             : matrix
    _tsmnsfn           : string
    _ts_mn             : string

tscs.src
    tscs               : proc
    _tsgrpmeans        : proc
    _tsprtp            : proc
    tscsset            : proc
    _tsfile            : proc

timeser.dec
    _ts_ver            : matrix
```

6.2.1 Building Library Files

GAUSS searches for a symbol name by first determining whether it is an intrinsic function. The symbol table is next searched. Finally, the list of active libraries is

6. SYMBOLS, PROCEDURES, GLOBAL VARIABLES, AND LIBRARIES

searched, in the order specified in the **library** command. Each library file contains a list of procedures and the name of the `.src` file that contains each procedure. If the `-nopath` option is chosen when the library is built **GAUSS** finds this `.src` file by looking in the `src_path`, specified in the `gauss.cfg` file. Otherwise **GAUSS** will use the explicit path to the `.src` file that exists in the `.lcg` file.

Library `.lcg` files may be revised in two ways, by using the **lib** command or by using the **Lib Tool**, discussed earlier.

The **lib** command adds symbols in `.src` and `dec` files to libraries. Suppose you want to add `norm.src` and, to initialize global variables, the `norm.dec` file to the `math.lcg` library. Both are accomplished via:

```
lib math norm.src -nopath;
lib math norm.dec -nopath;
```

The `math.lcg` library file is created if it does not already exist.

Now you may catalog the **sqrtinv** procedure in the **User** library. Type the following in the **Command** window:

```
lib user myprocs.src -nopath; @ Or simply -n rather than -nopath @
```

Now the procedure is available for use. Look in the `User.lcg` file (in the `c:\(gaussdir)\lib` directory). Note how the procedure is referenced. Now update the library without the `-nopath` option, i.e.

```
lib user myprocs.src;
```

We recommend using the `-nopath` option. **GAUSS** will then search the `src_path` for the appropriate file. This forces the user to keep their `.src` files in known locations.

6.2.2 The Library Tool

The **Library Tool** lets you manage your libraries.

Open the **Library Tool** by clicking the **Lib Tool** menu item on the **Tools** menu. Create a new library with the **New Library** button. Remove a library by selecting the **Delete Library** button. Add files to a library with the **Add** button. Remove files from a library with the **Remove** button.

To add absolute path names to the library index, use the **Add Paths** button. To only use file names for searching libraries, use the **Strip Paths** button.

Use **Rebuild** to recompile all the files used in the library, and rebuild the library index file. Use the **Revert to Original** button to revert to the configuration the library was in when the **Library Tool** was opened.

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

6.3 User Examples - Procedures and Libraries

Open another Edit window and type the following code into it. Save the file as `main.pgm`.

```
{zed, state1} = rndKMn(3,3,-1);    @ this statement defines an argument
                                   to the procedure @
zsi = sqrtinv(zed);                @ this statement calls the procedure @
```

The first function, `rndKMn`, is intrinsic. **GAUSS** finds it immediately and compiles the code to call it. The second function, `sqrtinv`, is a user-defined function. It currently resides in memory, in the symbol table (unless it's been deleted, using either **delete** or **new**).

Here's another example using procedures and libraries.

- Open the `myprocs.src` file you defined earlier. Add the following code to it:

```
proc (3) = regress (x,y);
  local xxi,b,ymx,b,sse,sd,t;
  xxi=invpd(x'x);
  b=xxi*(x'y);
  ymx=y-(x*b);
  sse=ymx'ymx/(rows(x)-cols(x));
  sd=sqrt(diag(sse*xxi));
  t=b./sd;
  retp(b,sd,t);
endp;
```

This procedure has two input arguments, an `x` matrix and a `y` matrix. It calculates regression coefficients, using the `y` matrix as the dependent variable and the `x` matrix as the independent variables. There are three output variables, the calculated coefficients, the coefficient standard errors, and their t-statistics.

- Save the file (in its original location, the `c:\(gaussdir)\src\usersrc` directory subdirectory).
- Check that this is in the `src_path` in the `gauss.cfg` file.
- Save the `myprocs.src` file in the User library by typing at the **GAUSS** prompt:

```
lib user myprocs.src -nopath
```

Use the `regress` procedure by calling it from a main file. Create a new file and type the following into it. Call this file `regress.prg`.

6. SYMBOLS, PROCEDURES, GLOBAL VARIABLES, AND LIBRARIES

```
// Create a 100*4 matrix of standard-normal random numbers
{x, state1} = rndKMn(100,4,-1)*100;
// define a file for output. No path is defined so it will be in your
// working directory
output file = tempdata.asc reset; @ reset - the file is overwritten
each time a write occurs @
// print x on the screen and to the file, in ascii form, space delimited
print x;
// turn off the output file
output off;
// Ascii load from the file, into a one-column vector
load x[] = tempdata.asc;
// turn the column vector into a 100*4 matrix
x = reshape(x,100,4);
// select the first column
y = x[.,1];
// select columns two, three, and four
x = x[.,2:cols(x)];
// call the "regress" function - it has three returns
{ coeff, stderr, tstat } = regress(x,y);
// Print column labels
Print "Coefficients, Standard Errors, T-Statistics";
// horizontally concatenate the returns from "regress" and print the result
print coeff~stderr~tstat;
```

The first line of the file creates a 100*4 matrix of pseudo-random values. The following three lines write the matrix to an output file, in ASCII form. The 5th line loads the ASCII dataset into the matrix x. In this case it's a character matrix with 400 rows and one column. However, the dataset has 100 observations and 4 variables. The second command reshapes this column vector into a 100*4 matrix.

Now you can run `regress.prg`.

*Recall that the default directory for saving files is `c:\(gaussdir)`, the **GAUSS** installation directory which is also the default **GAUSS** working directory. The working directory may be changed with the **chdir** or **ChangeDir** commands. **chdir** has no returns and therefore should be used only interactively. **ChangeDir** returns a null string if it fails and should therefore be used in programs. For example, to make the working directory equal to `c:\temp` you would enter `chdir \temp`; at the **GAUSS** prompt:*

6.4 Exercises

- 6.1 Add the **mply** procedure in section 6.1.1 to `myprocs.src`. The easiest way to do this is to simply open `myprocs.src` in an Edit window and copy and paste from

6. SYMBOLS, PROCEDURES, GLOBAL VARIABLES, AND LIBRARIES

the **Command** window. Update the `myprocs.lcg` library with and without explicit file paths so that it shows the new **mply** function. Notice the differences. Clear your global symbol table and test **mply** to ensure that it works.

- 6.2 Add global variables to **mply**. Update the `myprocs.lcg` file so that it contains the global variable declarations. Clear your global symbol table and test **mply** to ensure that it works.
- 6.3 Write the returns from the **regress** procedure to an Excel file, using the **exportf** function. You will probably need to view **GAUSS** help for **exportf** to do this problem.
- 6.4 Consider again the **regress** procedure. Suppose your data are in an Excel file, rather than an ASCII file. Write a command file to read the data, using **import** or **importf**. You will probably need to view **GAUSS** help to do this problem.

Chapter 7

Fundamentals

This chapter focuses on the different **GAUSS** data types, matrices, strings, string arrays, structures, scalar members of a structure, and arrays.

*The **show** and **type** commands are often used to see the type of a symbol*

7.1 Strings and String Arrays

The **string** keyword is used to create string arrays. Simple assignment of a string of characters in quotation marks creates a 1×1 , “scalar” string array. Elements of string arrays can be of any length.

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

```
This is a string                This is a string
```

A **let** statement is used to create string arrays. If the curly braces are present, the **let** is optional (it is implicit, rather than explicit). Strings are forced to upper case unless placed inside quotation marks.

```
let string s = { ab de, hi "lm" };    string s = { ab de, hi "lm" };
PRINT s;                              PRINT s;
```

```
AB    DE                            AB    DE
HI    lm                            HI    lm
```

***let** statements may include only literals as arguments, such as numbers or letters. They may not contain variable names or other functions.*

7.1.1 Special Characters in Strings

The backslash is used to embed special characters in strings. The most common special characters that need special treatment are the backslash (\) and quote (") characters. These are typically used in path declarations or in strings with quotation marks.

Path Example

Strings are commonly used for specifying dataset names. Two backslashes are required (though not in UNIX) to embed a backslash in a quoted string.

```
dataset = "c:\\gauss40\\examples\\freqdata.dat";
PRINT dataset;

c:\gauss40\examples\freqdata.dat
```

Strings Containing Quotation Marks

Quotation marks must be preceded with a backslash in quoted strings.

```
string s3 = "abcd"efgh";

(0) : error G0097 : String not closed
Undefined symbols:
    EFGH      (0)

string s3 = "abcd\"efgh";
PRINT s3;

abcd"efgh
```

The ASCII character code of a special character is required to print it.

Example of Using Special Characters in Strings

```
string s1 = "\21 \156 \239 \6";
PRINT s1;

§ £ ∩ ♠
```

7.2 Numeric and Character Matrices

Matrices are created the same way as string arrays, using explicit or implicit **let** statements. A single matrix may contain character and numeric data.

Character matrices are often used to store labels. Character matrix elements are limited to 8 characters (they are stored as doubles).

There are two reasons for using character matrices rather than string arrays. First, computing speed is faster with character matrices. Second, character data can be mixed with numeric data, though special instructions are required to print the character elements. It is better to use string arrays if neither of these considerations prevail in your problem.

Three identical character matrices are created with:

```

      xc = { alpha beta, gamma "delta" };
      let xc = { alpha beta, gamma "delta" };
let xc[2,2] = alpha beta gamma "delta" ;

```

```
PRINT $x1;
```

```

ALPHA  GAMMA
BETA   delta

```

Character data is stored in upper case. To force the storage of lower case data enter the data in quotations.

A special print operator, **\$**, is required to print a character matrix. This is because **GAUSS** does not internally distinguish between character and numeric data.

Numeric matrices are created the same way. The following three statements create identical 2×2 numeric matrices:

```

      xn = { 1 2, 3 4 };
      let xn = { 1 2, 3 4 };
let xn[2,2] = 1 2 3 4;

```

The third line in both examples shows that **GAUSS** automatically creates a matrix of the given dimension from a column vector. The **let** is required in this case because curly braces aren't used. The **let** statement is optional when curly braces are used.

7.2.1 Submatrices

String and matrix submatrices are generated by referring to the relevant indices, enclosed within square brackets. A `.` within square brackets refers to either all rows or all columns, depending on its position.

Suppose that matrix `x` is $n \times n$. The following two statements print the entire matrix:

```
PRINT x;
PRINT x[.,.];
```

All rows and columns one and three are printed using:

```
PRINT x[.,1 3];
```

Similarly, all columns and rows 2 and 5 are printed using:

```
PRINT x[2 5, .];
```

Of course individual elements may be referenced. Suppose `x =`

```
1.0000  2.0000  3.0000
4.0000  5.0000  6.0000
7.0000  8.0000  9.0000
10.0000 11.0000 12.0000
```

```
PRINT x[1 3, 2 3];
```

```
2.0000  3.0000
8.0000  9.0000
```

Note that the comma separates rows from columns.

Often the indices are in their own matrix, as in:

```
d = { 3, 1 };
PRINT x[d+1,d];
```

```
12.0000 10.0000
6.0000  4.0000
```

```
PRINT x[.,d];
```

```
3.0000  1.0000
6.0000  4.0000
9.0000  7.0000
12.0000 10.0000
```

7. FUNDAMENTALS

Another example is

```
x = zeros(4,4);
x[1 3,2 4] = ones(2,2);
PRINT x;

0.0000  1.0000  0.0000  1.0000
0.0000  0.0000  0.0000  0.0000
0.0000  1.0000  0.0000  1.0000
0.0000  0.0000  0.0000  0.0000
```

7.2.2 Special Matrices

Several special matrices are often used in statistical and mathematical computations. **GAUSS** has intrinsic functions for these:

- **Matrix of Ones:** `x = ones(2,2);` // a 2 x 2 matrix of ones
- **Matrix of Zeros:** `x = zeros(2,2);` // a 2 x 2 matrix of zeros
- **Identity Matrix:** `x = eye(2);` // a 2 x 2 identity matrix
- **Null or Empty Matrix**

It is sometimes convenient to define a null or empty matrix, which can be concatenated to, or otherwise redefined in a loop.

```
m = {};
i = 1;
do while i < 4;
    m = m|i;
    i = i + 1;
endo;

PRINT m';

1.0000  2.0000  3.0000
```

- **Additive Sequence:** `seqa(start, increment, number of elements)`

```
x = seqa(1,1,4);          x = seqa(0.01,0.2,4);
PRINT x;                 PRINT x;

1.0000                    0.010
2.0000                    0.210
3.0000                    0.410
4.0000                    0.610
```

- **Pseudo-random Matrices**

GAUSS has a new Kiss-Monster random number generator with a huge period (something like 10^{8859}). The papers at www.aptech.com/papers discuss in more detail the Kiss-Monster random number generator.

The **rndKMi** function generates a series of random integers and is the basis for all other Kiss-Monster random number procedures (beta, gamma, normal, negative binomial, poisson, uniform, and von Mises).

The state of the random number generator is an input argument to all the Kiss-Monster functions. State is a 500×1 vector. It defines the numbers produced by **rndKMi**. Setting the state argument to -1 causes **GAUSS** to use the system clock to initialize the random number generator.

The **rndKMu** and **rndKMn** functions use the Kiss-Monster algorithm to create matrices of uniform and standard normal pseudo-random numbers.

```
{xu, state1} = rndKMu(2,2,-1);
PRINT xu;

      0.6011    0.0566
      0.7268    0.6330

{xn, state1} = rndKMn(2,2,state1);
PRINT xn;

     -2.1025    0.3767
     -1.6260    0.1503
```

In the first case, **GAUSS** generates the state of the random number generator from the system clock. The second case uses the state of the random number generator after **rndKMu** is called as the input to **rndKMn**.

Saving the state of the random number generator (for example in **state.fmt**) allows for reproducible simulations. You may be simulating a time series analysis making modifications of some kind between runs. The same state variable should be used for each run, to tell whether any differences are due to your modifications or to the differences in the sequence of pseudo-random numbers.

7.3 Matrix Operators

Many operators require conformable matrices. The dimension of a matrix may be displayed using the **show** command. However, **show** results cannot be used in a program. The **rows** and **cols** commands are used to retrieve matrix dimensions that may be subsequently used in code, e.g.

```
r = rows(x);
c = cols(x);
```

7. FUNDAMENTALS

7.3.1 Conformability

inner-product An operation is *inner-product* conformable if the number of columns of the left-hand-side matrix is equal to the number of rows of the right-hand-side matrix. This type of conformability applies only to matrix multiplication, the ***** operator in **GAUSS**.

strict Strict conformability refers to element-by-element operations when both matrices have the same dimension. It applies to all **GAUSS** operators and functions *except* ***** and **/**.

sweep An operator is sweep conformable if

- either matrix is a $1 \times k$ row vector and the other matrix is $\ell \times k$, or
- either matrix is a $k \times 1$ column vector and the other matrix is $k \times \ell$.

Sweep compatibility is also known as $E \times E$ compatibility.

Suppose **X** is an $n \times k$ matrix. If **Y** is $(1 \times k)$ then the sum **Y+X** represents a vertical sweep; a column element in **Y** is added to each element in the corresponding column of **X**. Similarly, in **Y.*X**, elements in a each column of **X** are multiplied by the corresponding column element in **Y**. If **Y** is $(n \times 1)$ then **GAUSS** sweeps horizontally across the matrix **X**, so that **Y+X** represents the sum of each single row element of **Y** being added to each element in the corresponding row of **X**. Many **GAUSS** operators have sweep abilities.

GAUSS knows whether to sweep vertically or horizontally, depending how the matrices **X** and **Y** are defined.

scalar If either matrix is scalar, i.e., a 1×1 matrix, it is conformable for all operators and **GAUSS** functions. Scalar conformability involves the scalar being swept across all elements in the **X** matrix.

7.3.2 Numeric Operators

Element-by-Element Operators

+ Element-by-Element Addition:

- Element-by-Element Subtraction or negation:

.* Element-by-element multiplication:

The vector-matrix element-by-element operation is exceptionally useful in **GAUSS**, primarily to replace do loops.

```
y = x .* z;
```

If **x** is a column vector, and **z** is a row vector (or vice versa), then the “outer product” or “table” of the two will be computed.

- `./` Element-by-element division:
- `^` and `.^` Element-by-element exponentiation.
 $y = x^z;$
 If x is negative, z must be an integer.
- `%` Element-by-Element Modulo division:
 $y = x \% z;$
 For integers, this returns the integer value that is the remainder of the integer division of x by z . If x or z are noninteger, they will first be rounded to the nearest integer.
- `!` Element-by-Element 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 operator can generate very large numbers and for this reason most functions using the factorial are re-written with the log of the factorial. **GAUSS** contains a function returning the log of the factorial which, in many cases, is more useful than the `!` operator.
- ```
PRINT lnfact(x);
```
- |        |         |         |
|--------|---------|---------|
| 0.0000 | 0.6931  | 1.7917  |
| 3.1780 | 4.7875  | 6.5792  |
| 8.5252 | 10.6046 | 12.8018 |

### Matrix Operators

- `*` Matrix multiplication or multiplication:
- `/` Division or linear equation solution.  
 $x = b / A;$   
 This operator performs standard division if  $A$  and  $b$  are scalars. matrices. If  $b$  and  $A$  are conformable matrices, this operator solves the linear matrix equations.  
 $Ax = b$
- `.*` 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:

## 7. FUNDAMENTALS

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

$$x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$
$$z = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$
$$y = \begin{bmatrix} 4 & 5 & 6 & 8 & 10 & 12 \\ 7 & 8 & 9 & 14 & 16 & 18 \\ 12 & 15 & 18 & 16 & 20 & 24 \\ 21 & 24 & 27 & 28 & 32 & 36 \end{bmatrix}$$

\* ~ Horizontal direct product.

```
z = x * ~ y;
```

$$x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$
$$y = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$
$$z = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 21 & 24 & 28 & 32 \end{bmatrix}$$

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

~ Horizontal concatenation

| Vertical concatenation

' Transpose operator - works with matrices and string arrays.

- **minc(x)** finds the minimum column elements in matrix  $\mathbf{x}$ .
- **maxc(x)** finds the maximum column elements in matrix  $\mathbf{x}$ .
- **reshape(x,r,c)**; reshapes matrix  $\mathbf{x}$  so that it has  $\mathbf{r}$  rows and  $\mathbf{c}$  columns. **reshape** uses the first  $\mathbf{r}$  by  $\mathbf{c}$  elements:
- **rev(x)** reverses the order of the elements of the columns of a matrix.
- **sortc(x,c)**; sorts matrix  $\mathbf{x}$  on column  $\mathbf{c}$ .
- **sortind(x)** returns the sorted indices of  $\mathbf{x}$

- **sortmc**( $x, v$ ); sorts matrix  $\mathbf{x}$  on the multiple columns specified in  $\mathbf{v}$ .
- **trimr**( $x, t, b$ ); removes  $t$  rows from the top of matrix  $x$  and  $b$  rows from the bottom of matrix  $\mathbf{x}$ . **trimr** is faster than indexing.
- **vec**( $\mathbf{x}$ ) stacks the columns of matrix  $\mathbf{x}$ . The effects of **vec** are reversed by transposing the reshaped vector.
- **vecr**( $\mathbf{x}$ ) is a row version of **vec**, i.e., it creates a vector from the *rows* of a matrix rather than the columns as **vec** does. The effects of **vecr**( $\mathbf{x}$ ) may be reversed using **reshape**( $\mathbf{x}$ ) without having to transpose the result.

### 7.3.3 Exercises

- 7.1 Create a pseudo-random matrix. Set a submatrix of that matrix to zero by assigning a smaller matrix of zeros to a portion of the random matrix.
- 7.2 Create a  $10 \times 3$  matrix of pseudo-random numbers. Generate a column vector of the means of the columns of the pseudo-random matrix. Don't use **meanc** for this. (hint: create  $10 \times 1$  column vector of ones and post-multiply the transpose of the pseudo-random matrix by the vector of ones, dividing everything by the number of observations.)
- 7.3 Create a column vector containing a sequence of numbers using **seqa**. Call it  $\mathbf{s}$ . Create a square matrix of ones with the same number of rows and columns as rows in the sequence. Call it  $\mathbf{X}$ . Notice the differences between  $\mathbf{s} * \mathbf{X}$  and  $\mathbf{X} * \mathbf{s}'$ . The first element-by-element multiplication causes **GAUSS** to sweep vertically across  $\mathbf{X}$ . The second causes **GAUSS** to sweep horizontally, down  $\mathbf{X}$ .

### 7.3.4 Other String and Matrix Operators

$\$+$  used to concatenate scalar strings. It is also used for type conversion (discussed below).

```
string s1 = "where the quiet-colored";
string s2 = " end of evening smiles";
PRINT (s1 $+ s2);
```

```
where the quiet-colored end of evening smiles
```

$/$  Transpose operator - works with matrices and string arrays.

$\$|$  Vertical concatenation for string arrays and character matrices

$\$ \sim$  Horizontal concatenation for string arrays and character matrices

## 7.4 N-Dimensional Arrays

### Storage in Memory

Arrays are stored in memory in row major order, the same as matrices. A  $3 \times 2$  matrix is stored as follows:

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

The slowest moving dimension, the row number, is indexed on the right. The fastest moving dimension is indexed on the left.

A  $4 \times 3 \times 2$  array is stored as:

```
[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 the same way, with the entire real part first followed by the entire imaginary part.

### Array Dimensions and Orders

Every array has two key properties, The number of dimensions is returned from the **getdims** command and the size of each dimension, the vector of orders, is returned from the **getorders** command.

The first element in the vector of orders corresponds to the slowest moving dimension. The last element corresponds to the fastest moving dimension.

is

```
4
3
2
```

Dimension numbering follows a well-defined convention. The fastest moving dimension has a dimension number of 1 and the slowest moving dimension has a dimension number of N. A  $4 \times 3 \times 2$  array has 3 dimensions. The first element of a 3 dimensional array with orders  $4 \times 3 \times 2$  equals 4 and refers to the size of dimension 3. The second element equals 3 and refers to the size of dimension 2, etc.

Subarrays are referenced by a locator vector of indices. Because the elements of the vector of indices are always in the same order, from slowest moving to fastest moving,

each unique vector of indices locates a unique subarray. For example, suppose  $x$  is a 5 dimensional array with orders  $6 \times 5 \times 4 \times 3 \times 2$ . An  $6 \times 1$  vector of indices locates the scalar whose position is given by the indices. The locator vectors  $1|4|2|1|1$  and  $1|4|2|1|2$  each refer to a scalar. A  $5 \times 1$  vector of indices, e.g.  $4|4|2|1|2$  references a 1-dimensional array whose starting location is given by the indices. A  $4 \times 1$  vector of indices, e.g.  $6|2|2|2$  references a 2-dimensional array whose starting location is given by the indices, etc.

In general, an  $[N - K] \times 1$  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  $6 \times 5 \times 4 \times 3 \times 2$  array  $y$ , the  $2 \times 1$  vector of indices:

$$\begin{array}{c} 2 \\ 5 \end{array}$$

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

### 7.4.1 Array Definition

Arrays are created with **arrayalloc**, **arrayinit**, **arrayinitcplx**, **areshape**, and **mattoarray**.

**arrayalloc** creates an N-dimensional array with unspecified contents. For example, to create an N-dimensional array  $y$ ,

```
orders = { 2,3,4 }; // vector of orders
cf = 1; // complex flag, equals one for complex
y = arrayalloc(o,cf);
```

**arrayinit** creates an N-dimensional array with a specified fill value. To create a  $3|4|5$  array of ones,

```
orders = { 3,4,5 }; // vector of orders
y = arrayinit(o,1);
```

**arrayinitcplx** creates an N-dimensional array of complex numbers with a specified fill value. To create a  $3|4|5$  array of complex numbers, where the real part equals one and the imaginary part equals -1:

```
orders = { 3,4,5 }; // vector of orders
y = arrayinitcplx(o,1,-1);
```

**areshape** creates an N-dimensional array with a specified fill value. To create a  $3|4|5$  array of ones,

```
y = areshape(3|4|5,1);
```

**mattoarray** changes a matrix to a 1-or-2 dimensional array

```
y = mattoarray(x);
```

## 7.4.2 Array to Matrix Conversion

**arraytomat** creates a matrix (1 or 2 dimensions) from an array. For example, suppose  $a$  is a 5-dimensional array, with orders  $6 \times 5 \times 4 \times 3 \times 2$ . To convert the  $3 \times 2$  matrix at  $a[3,5,1,.,.]$  to a matrix:

```
y = arraytomat(a[3,5,1,.,.]);
```

This command lets all **GAUSS** matrix functions be applied to a matrix. You could, for example, loop through a matrix using an **loopnextindex**, **walkindex**, **nextindex**, and **previousindex** and pull matrices from the array at each step.

**getmatrix** gets a contiguous matrix out of an N-dimensional array. For example,

```
a = seqa(1,1,120);
a = areshape(a,2|3|4|5);
loc = { 1,2 };
y = getmatrix(a,loc);
```

The result will be

```

 21 22 23 24 25
y = 26 27 28 29 30
 31 32 33 34 35
 36 37 38 39 40
```

The `loc` argument is an  $M \times 1$  vector of indices.

**getmatrix4D** gets a contiguous matrix out of an N-dimensional array. It is faster than **getmatrix**. For example,

```
a = seqa(1,1,120);
a = areshape(a,2|3|4|5);
loc = { 1,2 };
y = getmatrix4D(a,2,3);
```

The result will be

```

 101 102 103 104 105
y = 106 107 108 109 110
 111 112 113 114 115
 116 117 118 119 120
```

The two arguments to **getmatrix4D** are scalars, the indices of the first and second elements in the vector of orders of the array.

**getscalar4D** is similar to **getmatrix4D**

**getscalar3D** is similar to **getmatrix4D**

### 7.4.3 Array Manipulation

An extensive set of procedures allow numerous types of array manipulation:

**loopnextindex** walks through the vector of indices of an array, executing a series of statements in a loop as long as it can walk through the index vector. For example, to define an array,  $a$  and set each  $6 \times 7$  subarray of  $a$  equal to a  $6 \times 7$  matrix of standard normal random numbers

```
orders = { 2,3,4,5,6,7 };
a = arrayalloc(orders);
ind = { 1,1,1,1 };
loopni2:
 setarray a, ind, rndn(6,7);
loopnextindex loopni, ind, orders;
```

An optional argument to **loopnextindex** lets you specify which dimension you want looped. You could, using this optional argument, set a portion of the array.

**aconcat** concatenates conformable matrices and arrays in a user-specified dimension. Suppose  $a$  and  $b$  are  $N$  and  $K$  dimensional arrays that are conformable.  $a$  and  $b$  are conformable only if all of their dimensions except the one over which concatenation will occur have the same sizes. If  $a$  or  $b$  is a matrix, then the size of dimension 1 is the number of rows in the matrix, and the size of dimension 2 is the number of columns in the matrix.

```
y = aconcat(a,b,dim);
```

where  $dim$  is a scalar, the dimension in which to concatenate.

For example,

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

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

```
a = reshape(seqa(1,1,20),4,5);
b = zeros(4,5);
y = aconcat(a,b,3);
```

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

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

## 7. FUNDAMENTALS

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

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

$y$  will be a  $2 \times 3 \times 5$  array, such that:

```
[1,1,1] through [1,3,5] =
 0 0 0 0 1
 0 0 0 0 2
 0 0 0 0 3

[2,1,1] through [2,3,5] =
 0 0 0 0 4
 0 0 0 0 5
 0 0 0 0 6
```

**nextindex** returns the index of the next element or subarray in an array.

For example,

```
a = ones(2520,1);
a = areshape(a,3|4|5|6|7);
orders = getorders(a);
ind = { 2,3,5 };
ind = nextindex(ind,orders);
```

```
 2
ind = 4
 1
```

In this example, **nextindex** incremented `ind` to index the next 6x7 subarray in array `a`.

**areshape** reshapes a scalar, matrix, or array into an array of user-specified size. If there are more elements in  $x$  than in  $y$ , the remaining elements are discarded. If there are not enough elements in  $x$  to fill  $y$ , then when `areshape()` runs out of elements, it goes back to the first element of  $x$  and starts getting additional elements from there.

For example:

```
x = 3;
orders = { 2,3,4 };
y = areshape(x,orders);
```

$y$  will be a 2x3x4 array of threes.

**atranspose** transposes an N-dimensional array.

The format of **atranspose** is

```
y = atranspose(x,nd);
```

where  $nd$  is an  $N \times 1$  vector of dimension indices, the new order of dimensions and  $y$  is an N-dimensional array, transposed according to  $nd$ .

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

Example:

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

This example transposes the dimensions of  $x$  that correspond to the first and second elements of the vector of orders.  $x$  is a 2x3x4 array, such that:

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

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

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

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

$y$  will be a 3x2x4 array such that:

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

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

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

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

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

## 7. FUNDAMENTALS

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

**previousindex** is similar to **nextindex**

**setarray** sets a contiguous subarray of an N-dimensional array.

The format is

```
setarray a,loc,src;
```

where **loc** is an  $M \times 1$  vector of indices that locates the subarray of interest, where  $M$  is a value from 1 to  $N$  and **src** is an  $[N - M]$ -dimensional array, matrix, or scalar.

**setarray** resets the specified subarray of  $a$  in place, without making a copy of the entire array. It is faster than **putarray**.

If **loc** is an  $N \times 1$  vector, then **src** must be a scalar. If **loc** is an  $[N-1] \times 1$  vector, then **src** must be a 1-dimensional array or a  $1 \times L$  vector, where  $L$  is the size of the fastest moving dimension of the array. If **loc** is an  $[N-2] \times 1$  vector, then **src** must be a  $K \times L$  matrix, where  $K$  is the size of the second fastest moving dimension, or **src** must be a  $K \times L$  2-dimensional array. Otherwise, if **loc** is an  $M \times 1$  vector, then **src** must be an  $[N-M]$ -dimensional array, whose dimensions are the same size as the corresponding dimensions of array  $a$ .

Example:

```
a = arrayalloc(2|3|4|5|6);
src = arrayinit(4|5|6,5);
loc = { 2,1 };
setarray a,loc,src;
```

**putarray** is similar to **setarray** except that a copy is made of the array.

**walkindex** walks the index of an array forward or backward through a specified dimension.

**walkindex** returns a scalar error code if the index cannot walk further in the specified dimension and direction.

Example:

```
orders = getorders(3|4|5|6|7);
a = arrayinit(orders,1);
ind = { 2,3,3 };
ind = walkindex(ind,orders,-2);
```

```
 2
ind = 2
 3
```

This example decrements the second value of the index vector **ind**.

### 7.4.4 Other Array Operators

**GAUSS** currently supports a limited number of mathematical operations on arrays. New capabilities are continually being added.

All scalar and element-by-element logical and relational operators work with arrays. Add, subtract, and element-by-element multiply and divide work with arrays. **asum** and **amean** are designed for arrays. **real**, **imag**, and **complex** work with arrays.

## 7.5 Structures

Strings, string arrays, and matrices may be elements of a structure. A particular structure type is defined by the elements it contains. For example, the following structure of type `foo` has two members, a matrix and a string:

```
struct foo
{
 matrix x;
 string y;
}
```

A structure type must be instantiated before it can be used. The following instantiates a `foo` structure, assigning it the symbol name `foo1`. The two elements of `foo1` are assigned values and printed.

```
struct foo foo1;
foo1.x = 3;
foo1.y = "hello world";
print foo1.x;
print foo1.y;
```

### 7.5.1 Arrays of Structures

Often it is useful to pass arrays of structures to functions. These are created as follows:

```
struct t { matrix x; string s; string array sa; array a; };
struct t t1;
t1 = reshape(t1,12,1);
```

## 7.6 Type Conversion

Strings, character matrices, and numbers can be converted among each other in two ways. The first way is to use the **\$+** operator. This operator converts between string arrays and character matrices. The second way is to use one of the **ftocv**, **ftos**, or **stof** commands.

## 7. FUNDAMENTALS

### 7.6.1 Using the \$+ operator

The essential idea behind type conversion using the **\$+** operator is to append either a null string or a zero to the string or variable being converted. A character matrix element is transformed into a string by concatenating a null string to its beginning. A string or string array is transformed to a character matrix by concatenating a 0 to its beginning. The null string or 0 must be first, preceding the variable.

**0 \$+ string array** converts a string or string array to a character matrix with same number of rows and columns as the string array, each element truncated at eight characters

**"" \$+ character matrix** converts a character matrix to a string array with the same number of rows and columns as the character matrix

**ftocv** transforms a number into a character matrix element.

**ftos** transforms a scalar into a string.

**ftostrc** transforms a matrix into a string array, using a C format specification

**stof** transforms a string representing a number or a character matrix element representing a number into a number.

For example, convert a character matrix to a string array. Use **show** to confirm that the conversions work properly. First create a character matrix,

```
c1 = { one two, three four };
PRINT $c1;

 ONE TWO
 THREE FOUR

show c1;

C1
 32 bytes at [001388cc] 2,2 MATRIX

64000 bytes program space, 0% used
4130296 bytes workspace, 3970936 bytes free
2 global symbols, 500 maximum, 1 shown
```

Then convert it to a string array,

```
s1 = "" $+ c1;
PRINT s1;
```

```

 ONE TWO
 THREE FOUR

show s1;

S1
 56 bytes at [0013886c] 2,2 STRING ARRAY

64000 bytes program space, 0% used
4130296 bytes workspace, 3970936 bytes free
2 global symbols, 500 maximum, 1 shown

```

Here's an example that shows type conversion using **\$+** and **ftocv**. The second argument of **ftocv** is the minimum field width. **GAUSS** expands the minimum if necessary so that the entire variable is returned. The third argument is the precision (0 places if a scalar or character). See if you can predict the types returned.

```

n = seqa(1,1,10);
show n;
x = ftocv(n,1,0);
show x;
y = "$+ftocv(n,1,0);
show y;
z = 0$+ftocv(n,1,0);

```

Here is an example which creates a 10\*1 character matrix containing the character elements "var1" to "var10".

```

n = seqa(1,1,10); /* creates a 10*1 numeric vector, containing the
 numbers 1,...10 */
var_n = 0$+"var"$+ftocv(n,1,0);

```

The **0\$+** operator at the start of the `var_n` line ensures that the result will be a matrix. The **ftocv(n,1,0)** command converts the numeric 10\*1 sequence vector 1,...,10 into a character matrix. The **"var"\$+ftocv(n,1,0)** command appends the "var" prefix to the character matrix. This is a sweep operation since "var" is a single string which is appended to a 10\*1 character matrix.

## 7.6.2 Exercises

- 7.4 Create a string array of numbers. Convert the string array to a character matrix. Convert the character matrix to a numeric matrix. Convert the numeric matrix to a string array.

## 7. FUNDAMENTALS

- 7.5 Create a string with more than eight characters. Use **reshape** to create a string array where each element of the array is equal to your initial string. Convert the string array to a character matrix. Print the character matrix. Why are your data truncated?
- 7.6 Create a string array containing VAR1,VAR2,... (hint: create a string array with each element equal to VAR. Next create a conformable string array of a numeric sequence. Finally add them together using the **\$+** operator).
- 7.7 Create a  $5 \times 5$  string array where elements in rows 2 and 3 and columns 2 and 3 are set to the letter B and the remaining elements set to the letter A.
- 7.8 Many **GAUSS** procedures require character vector arguments. Here's a fast way to create them. Try the following code (isn't **GAUSS** beautiful?!):

```
string s = first, second, third, fourth, fifth;
s = 0$+s;
```

### 7.7 Missing Values

In the “real world”, data nearly always contain missing values. **GAUSS** integrates missing values into all its applications and functions. A missing value is represented on the screen and in output as a “.”.

```
x = { 1 . 3, 4 5 ., 7 8 9};
PRINT x;

1.0000 . 3.0000
4.0000 5.0000 .
7.0000 8.0000 9.0000
```

The missing value is propagated through any computations in which it is involved. In other words, any operation with a missing value always results in a missing value.

```
PRINT x'x;

66.0000 . .
 . . .
 . . .
```

- **packr(x)** deletes *rows* from matrix **x** which contain missing values. If **x** is a matrix of data, **packr** is equivalent to a list-wise deletion of data.
- **miss(x,v)** converts elements of **x** that are equal to **v** to **GAUSS**'s missing value code.
- **missrv(x,v)** converts missing values in **x** into the values in **v**.

### 7.7.1 Exercises

- 7.9 Create a  $5 \times 5$  matrix of missing values. First create a numeric matrix. Convert the numeric matrix to one with missing value.

## 7.8 Scalar and Element-by-Element Relational Operators

**GAUSS** relational operators are in two categories, scalar and element-by-element. Both return a 1 if a comparison is true, and a 0 if it is false.

Scalar relational operators always return a scalar value, either a 1 or a 0. If strictly conformable matrices are compared, a 1 is returned if *every* comparison is true. Otherwise a 0 is returned. Scalar relational operators are primarily used in **if** statements and **do** loops.

Element-by-element operators return matrices of 1's and 0's corresponding to the outcome of the comparisons.

Scalar and element-by-element relational operators come in three versions. Numeric comparisons are accomplished using abbreviations or the corresponding math notation. The \$ operators do character comparisons. If the relational operator is preceded by a dot, '.', the result is a matrix of 1's and 0's corresponding to an element by element comparison of the matrices.

|                       | Scalar |    |      | Element-by-Element |     |       |
|-----------------------|--------|----|------|--------------------|-----|-------|
| less than             | lt     | <  | \$<  | .lt                | .<  | .\$<  |
| less than or equal    | le     | <= | \$<= | .le                | .<= | .\$<= |
| equal                 | eq     | == | \$== | .eq                | .== | .\$== |
| not equal             | ne     | /= | \$/= | .ne                | ./= | .\$/= |
| greater than or equal | ge     | >= | \$>= | .ge                | .>= | .\$>= |
| greater than          | gt     | >  | \$>  | .gt                | .>  | .\$>  |

### Numeric Scalar Example

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

 1.0000 2.0000 3.0000
 4.0000 5.0000 6.0000
 7.0000 8.0000 9.0000

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

## 7. FUNDAMENTALS

```
PRINT z;

 9.0000 8.0000 7.0000
 6.0000 5.0000 4.0000
 3.0000 2.0000 1.0000
```

```
PRINT (x < z);
```

```
 0.0000
```

```
PRINT (x > z);
```

```
 0.0000
```

Note that a matrix is strictly conformable to a scalar:

```
PRINT (x < 10);
```

```
 1.0000
```

It is important to understand that  $x \neq z$  is not the negation of  $x == z$ . Thus,

```
PRINT (x /= z);
```

```
 0.0000
```

The comparison is false because one of the elements of  $x$  is equal to its corresponding element in  $z$ , which means that the comparison is not true for every element.

```
PRINT not(x == z);
```

```
 1.0000
```

This comparison is true because at least one element of  $x$  is not equal to its corresponding element of  $z$  (actually, all but one).

### Numeric element-by-element example

```
PRINT (x .> 4.5);

 0.0000 0.0000 0.0000
 0.0000 1.0000 1.0000
 1.0000 1.0000 1.0000
```

The result of a comparison is itself a matrix and thus may be used in expressions.

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

```
1.0000 2.0000 3.0000
4.0000 5.0000 6.0000
7.0000 8.0000 9.0000
```

```
PRINT (x .<= 5).*x;
```

```
1.0000 2.0000 3.0000
4.0000 5.0000 0.0000
0.0000 0.0000 0.0000
```

## 7.9 Scalar and Element-by-Element Logical Operators

Logical operators perform logical or Boolean operations on numeric values. For these operators, a 1 is *true* and a 0 is *false*.

|              | Scalar | Element-by-Element |
|--------------|--------|--------------------|
| complement   | not    | .not               |
| conjunction  | and    | .and               |
| disjunction  | or     | .or                |
| exclusive or | xor    | .xor               |
| equivalence  | eqv    | .eqv               |

Scalar operators produce scalar results and are primarily used in **if** statements and **do** loops which require scalars. Element-by-element operators return matrices of 1's and 0's corresponding to the outcome of the comparisons.

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

```
1.0000 2.0000 3.0000
4.0000 5.0000 6.0000
7.0000 8.0000 9.0000
```

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

```
9.0000 8.0000 7.0000
6.0000 5.0000 4.0000
3.0000 2.0000 1.0000
```

## 7. FUNDAMENTALS

```
PRINT (not(x < z) or not(x < z));

1.0000

PRINT (not(x < z) xor not(x < z));

0.0000

y = { 0 1 1 };
PRINT y;

0.0000 1.0000 1.0000

PRINT (.not(x .< 4) .xor y);

0.0000 1.0000 1.0000
1.0000 0.0000 0.0000
1.0000 0.0000 0.0000

PRINT (.not y);

1.0000 0.0000 0.0000
```

### 7.9.1 Exercises

- 7.10 Create a  $3 \times 3$  matrix of Gaussian pseudo-random numbers. Use a relational operator to set elements of the matrix that are less than zero to zero.
- 7.11 Create a second  $3 \times 3$  matrix of Gaussian pseudo-random numbers. Generate a third matrix that equals the element-by-element products of the two matrices, conditional on the products being less than zero. All other elements of the third matrix should equal zero.
- 7.12 Create two  $100 \times 100$  matrices of Uniform pseudo-random numbers. Generate a third matrix with elements equal to the sum of the first two matrices, given that each of the two elements in the sum is greater than .5. Elements of the third matrix are zero if this condition is not met. Use **sumc** to see whether your results roughly match the expected results (Hint: First turn the third matrix into ones and zeros).

## 7.10 Formatting Output and Printing Output

Numeric values printed by **GAUSS** often fill your screen with output. The **GAUSS** default format for printing numeric matrices is a field length of 16 characters and 8

characters after the decimal point.

Use the format statement to change this. This format statement, the **GAUSS** default format,

```
format /mb1 /ros 16,8;
```

says to print 1 carriage return/line feed before each row of a matrix with more than one row (the /mb1 term), to print right justified, either signed decimals or signed scientific notation, depending which is most compact (the /ro term - see below), that there is a space inserted after each number printed (the 's' term in /ros - you may also have commas, tabs, or no trailing character), and that all printed numbers will have a field width of 16 characters with 8 characters after the decimal.

The following pages define all available on-screen (and print) formats. You can also get this information by using the **GAUSS** on-line help system. Put your cursor on the word **format** in the **GAUSS** Command window and press the F1 key.

Experiment now with different format statements. Type the following into the command window:

```
{x, state1} = rndKMn(5,2,-1); @ create a 5*2 matrix of @
 @ pseudo-random standard normal draws @
format /m1 /rds 16,8;
PRINT x;
```

Run the three lines of code a number of times, changing parts of the format specification each time. Here is a description of the **format** command:

**format** *[/typ]* *[/fmted]* *[/mf]* *[/jnt]* **[f,p]**;

**/typ** literal, symbol type flag(s). Indicates the symbol types for which the output format is set.

**/mat, /sa, /str** Formatting parameters are maintained separately for matrices (**/mat**), string arrays (**/sa**), and strings (**/str**). You can specify more than one */typ* flag; the format will be set for all types indicated. If no */typ* flag is listed, **format** assumes **/mat**.

**/fmted** literal, enable formatting flag.

**/on, /off** Enable/disable formatting. When formatting is disabled, the contents of a variable are dumped to the screen in a “raw” format. **/off** is currently supported only for strings. “Raw” format for strings means that the entire string is printed, starting at the current cursor position. When formatting is enabled for strings, they are handled the same as string arrays. This shouldn’t be too surprising, since a string is actually a 1x1 string array.

## 7. FUNDAMENTALS

- /mf** literal, matrix row format flag.
- /m0** no delimiters before or after rows when printing out matrices.
- /m1 or /mb1** print 1 carriage return/line feed pair before each row of a matrix with more than 1 row.
- /m2 or /mb2** print 2 carriage return/line feed pairs before each row of a matrix with more than 1 row.
- /m3 or /mb3** print “Row 1”, “Row 2”... before each row of a matrix with more than one row.
- /ma1** print 1 carriage return/line feed pair after each row of a matrix with more than 1 row.
- /ma2** print 2 carriage return/line feed pairs after each row of a matrix with more than 1 row.
- /a1** print 1 carriage return/line feed pair after each row of a matrix.
- /a2** print 2 carriage return/line feed pairs after each row of a matrix.
- /b1** print 1 carriage return/line feed pair before each row of a matrix.
- /b2** print 2 carriage return/line feed pairs before each row of a matrix.
- /b3** print “Row 1”, “Row 2”... before each row of a matrix.

- /jnt** literal, matrix element format flag – controls justification, notation and trailing character.

### Right-Justified

- /rd** Signed decimal number in the form  $[-]#####.###$ , where  $####$  is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed.
- /re** Signed number in the form  $[-]#.##E\pm###$ , where  $\#$  is one decimal digit,  $##$  is one or more decimal digits depending on the precision, and  $###$  is three decimal digits. If precision is 0, the form will be  $[-]#E\pm###$  with no decimal point printed.
- /ro** This will give a format like **/rd** or **/re** depending on which is most compact for the number being printed. A format like **/re** will be used only if the exponent value is less than -4 or greater than the precision. If a **/re** format is used, a decimal point will always appear. The precision signifies the number of significant digits displayed.
- /rz** This will give a format like **/rd** or **/re** depending on which is most compact for the number being printed. A format like **/re** will be used only if the exponent value is less than -4 or greater than the precision. If a **/re** format is used, trailing zeros will be suppressed and a decimal point will appear only if one or more digits follow it. The precision signifies the number of significant digits displayed.

**Left-Justified**

- /ld** Signed decimal number in the form  $\llbracket - \rrbracket \#\#\#\#. \#\#\#\#$ , where  $\#\#\#\#$  is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the precision. If the precision is 0, no decimal point will be printed. If the number is positive, a space character will replace the leading minus sign.
- /le** Signed number in the form  $\llbracket - \rrbracket \#. \#\#E\pm\#\#\#$ , where  $\#$  is one decimal digit,  $\#\#$  is one or more decimal digits depending on the precision, and  $\#\#\#$  is three decimal digits. If precision is 0, the form will be  $\llbracket - \rrbracket \#E\pm\#\#\#$  with no decimal point printed. If the number is positive, a space character will replace the leading minus sign.
- /lo** This will give a format like **/ld** or **/le** depending on which is most compact for the number being printed. A format like **/le** will be used only if the exponent value is less than -4 or greater than the precision. If a **/le** format is used, a decimal point will always appear. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.
- /lz** This will give a format like **/ld** or **/le** depending on which is most compact for the number being printed. A format like **/le** will be used only if the exponent value is less than -4 or greater than the precision. If a **/le** format is used, trailing zeros will be suppressed and a decimal point will appear only if one or more digits follow it. If the number is positive, a space character will replace the leading minus sign. The precision specifies the number of significant digits displayed.

**Trailing Character**

The following characters can be added to the */jnt* parameters above to control the trailing character if any:

**format /rdn 1,3;**

- s** The number will be followed immediately by a space character. This is the default.
  - c** The number will be followed immediately with a comma.
  - t** The number will be followed immediately with a tab character.
  - n** No trailing character.
- f** scalar expression, controls the field width.
- p** scalar expression, controls the precision.

**7.10.1 Printing Mixed Matrices of Characters and Numbers**

Character data may be easily mixed with numeric data in a mixed matrix. However, special print functions are required to print mixed matrices.

## 7. FUNDAMENTALS

```
a = { alpha 1, beta 2, gamma 3, delta 4 };

PRINT a; PRINT $a;

+DEN 1.0000 ALPHA
+DEN 2.0000 BETA
+DEN 3.0000 GAMMA
+DEN 4.0000 DELTA
```

The first print function prints only the numeric column correctly while the second print function prints only the character column.

Mixed matrices are printed using either **printfmt** or **printfm**. Both functions use a 1\*k mask array, indicating whether a column is numeric ( = 1) or alpha ( = 0). Both functions also use explicit format instructions for numeric and character data. The explicit instructions include the type of format desired (e.g. right justified, left justified), the minimum field length, and the precision.

**printfmt** uses **GAUSS** global variables to define numeric and character print formats. It has two arguments, the matrix to be printed and a mask to indicate which columns are character and which columns are numeric. The **GAUSS** global print variables for numeric and character data are named **\_\_fmtnv**, and **\_\_fmtcv**, respectively. Default values for all global **GAUSS** variables, including **\_\_fmtnv**, and **\_\_fmtcv**, are found in the **gauss.dec** file. Their declaration looks like this:

```
declare matrix __fmtnv = { ".*.*lg" 16 8 };
declare matrix __fmtcv = { ".*.*s" 8 8 };
```

The global values for **\_\_fmtnv** and **\_\_fmtcv** are changed by either editing the **gauss.dec** file and issuing a **new** command or by assigning new values directly.

*The easiest way to use different format specifications is to copy and paste from the **GAUSS** help file, i.e. bring up the help for **printfm** and copy and paste the desired format into your code.*

```
a = { alpha 1, beta 2, gamma 3, delta 4 };
__fmtnv = { ".*.*lf" 8 0 };
__fmtcv = { "-*.*s" 8 8 };
mask = { 0 1 };
call printfmt(a,mask);

ALPHA 1
BETA 2
GAMMA 3
DELTA 4
```

**printfm** is the second way to print mixed matrices. It has three arguments, the matrix to be printed, a mask indicating which columns are character and which are numeric, and an explicit format matrix, **fmt**. The number of **fmt** rows is the same as the number of columns in the matrix to be printed, one row of formatting information for each column.

The explicit format matrix, **fmt** is identical in form to the **GAUSS** global format variables used in **printfmt**. The first column of **fmt** is a string containing a format specification. The second column of **fmt** is the width of the field, and the third column of **fmt** is the precision to be printed, the number of places after the decimal place. All print format matrices have three columns.

```
a = { alpha 1, beta 2, gamma 3, delta 4 };
mask1 = { 0 1 };
fmt = { "-*. *s" 8 8,
 ".*. *lf" 6 2 };
call printfm(a,mask1,fmt);
```

```
ALPHA 1.00
BETA 2.00
GAMMA 3.00
DELTA 4.00
```

In this example, **printfm** is invoked using **call** to prevent the return from **printfm** from being printed to the screen. If **call** is not used, **printfm** returns a 1 if successful and a 0 if it fails.

```
a = { alpha 1, beta 2, gamma 3, delta 4 };
call printfmt(a,mask1);
```

```
ALPHA 1
BETA 2
GAMMA 3
DELTA 4
```

### 7.10.2 Exercises

- 7.13 Create a  $2 \times 3$  string array of numbers and a  $2 \times 3$  numeric matrix of the same numbers. Use the **show** command to see that the matrices are properly defined
- 7.14 Create a mixed matrix of character and numeric data. Give the numeric data lots of values after the decimal point (you might use **rndKMu** or **rndKMn** to do this). Print your matrix to the screen with a numeric format field with of 5 and a precision of 2.

## 7. FUNDAMENTALS

### 7.11 Control Statements

#### 7.11.1 Loops

**GAUSS** has **do while**, **do until** and **for** statements for looping.

```
x = { 1 1 1, 1 2 4, 1 3 9, 1 4 16, 1 5 25 };
m = 0;
i = 1;
do until i > 5;
 m = m + x[i,.]' * x[i,.];
 i = i + 1;
endo;
PRINT m;
```

```
5.0000 15.0000 55.0000
15.0000 55.0000 225.0000
55.0000 225.0000 979.0000
```

An alternative to **do until** is **do while**. For example, in the above loop **do while i <= 4** is equivalent to **do until i > 5**. Note that they are not generally equivalent. Suppose, for example, the loop variable, **i**, is incremented in non-integer amounts.

```
i = 0;
do until i > 5;
 PRINT i;
 i = i + 2.1;
endo;
```

The output of this program is:

```
0.0000
2.1000
4.2000
```

Change the **do until** statement to the following:

```
i = 0;
do while i <= 4;
 PRINT i;
 i = i + 2.1;
endo;
```

The output of this is:

```
0.0000
2.1000
```

One less number is printed in the latter version of the **do** loop because **i** is greater than 4 after the 2nd time through the loop, whereas in the former version, **i** is still less than 5.

The **for** statement is faster than the **do while** and **do until** statements. However, its functionality is more limited. It uses only integers as the loop index whereas anything, an integer, complex number, ratio, string, or matrix, may be used as the loop index in a **do while** or **do until** loop. The **for** loop stops only when the specified limit value is exceeded. Stopping conditions for the **do while** and **do until** loops are more general; anything may be specified. Finally, recursive procedure calls are allowed in **do** loops but not in **for** loops.

```
x = zeros(4,1);
for i(1,4,1);
 x[i] = i^2;
endfor;
PRINT x;
```

```
1.0000
4.0000
9.0000
16.0000
```

Break out of a loop using the **break** command. Go to the top of a loop with the **continue** command.

### 7.11.2 Conditional Branching

**if...elseif...else...endif** statements are used to control the conditional execution of statements. The **elseif** and **else** are optional.

Here's an example using the uniform Kiss-Monster random number generator, **rndKMu**. The third argument to **rndKMu** is the state of the random number generator. A constant state will lead to identical random numbers being generated. This example first defines an initial state based on your computer's system clock. It saves this state to disk, as **state.fmt**, enabling subsequent calls to **rndKMu** to use a constant identical state.

```
{z, state1} = rndKMu(1,1,-1);
save state = state1;
```

## 7. FUNDAMENTALS

```
i = 0;
load state1 = state; @ loads the previously defined state vector @
do until i > 10;
 {z, state1} = rndKMu(1,1,state1);
 z = 10*z;

 if z < 2;
 PRINT "A";;
 elseif z >= 2 and z < 8;
 PRINT "B";;
 else;
 PRINT "C";;
 endif;

 i = i + 1;
endo;
```

BABBAAACBAB

The double semi-colons at the end of the print statement tells **GAUSS** not to add a line-feed when printing.

### 7.12 Procedures and Keywords

Often a single computational task is performed many times. The code for this type of problem is more easily maintained if it is kept in one place. A **GAUSS** procedure provides a means for “encapsulating” a calculation. Any number of matrices, strings, or pointers to other procedures (i.e. arguments) may be passed to, and returned from, a procedure. Chapter 6 discusses procedures in detail.

**keyword** take a string argument as input and does not provide a return. The purpose is to provide a more natural, statistical package-like syntax for invoking functions in **GAUSS**. Inputting required information could involve some tedious editing of a command file. **keyword** procedures simplify this input and provide an interface for people who are not versed in **GAUSS** programming. Keywords are covered in the **GAUSS** Advanced Course.

7. *FUNDAMENTALS*

## Chapter 8

# Graphics

GAUSS Publication Quality Graphics (PQG) routines are built on the functions in GraphiC by Scientific Endeavors Corporation.

All output to a PQG window occurs during a call to one of the main graphics procedures:

|                |                                           |
|----------------|-------------------------------------------|
| <b>bar</b>     | Bar graphs.                               |
| <b>box</b>     | Box plots.                                |
| <b>contour</b> | Contour plots.                            |
| <b>draw</b>    | Draws graphs using only global variables. |
| <b>hist</b>    | Histogram                                 |
| <b>histp</b>   | Percentage histogram.                     |
| <b>histf</b>   | Histogram from a vector of frequencies.   |
| <b>loglog</b>  | Log scaling on both axes.                 |
| <b>logx</b>    | Log scaling on X axis.                    |
| <b>logy</b>    | Log scaling on Y axis.                    |
| <b>polar</b>   | Polar plots.                              |
| <b>surface</b> | 3-D surface with hidden line removal.     |

|            |                      |
|------------|----------------------|
| <b>xy</b>  | Cartesian graph      |
| <b>xyz</b> | 3-D Cartesian graph. |

These routines allow for significant customization using additional graphics procedures and global graphics variables. Graphic panel size and location are customizable. Users can create a single full size graph, insert a smaller graph into a larger one, tile a window with several equally sized graphs, or place several overlapping graphs in the same window. Users can add legends, extra lines, arrows, symbols, messages, and change fonts.

## 8.1 Using the VWR Graphics Viewer

The above procedures automatically write a graphics file with a default name of `graphic.tkf` (changeable with the `_ptek` global variable.) This file is read with the `vwr` graphics viewer. The `README.vwr` file in the **GAUSS** installation directory discusses the capabilities of the `vwr` viewer. This file may be opened from a graphics window's `Help` menu.

The `vwr` viewer is invoked from a command prompt. A simple syntax is:

```
vwr graphic.tkf
```

## 8.2 Graphics Windows

**GAUSS** automatically creates one or more graphics windows when graphics calls are made (e.g., **xy**, **surface**).

The default behavior is to open a new graphics window for every graph created, but this is configurable via either `pqgwin` or `setvwrmode`.

```
pqgwin one; // Creates one window for all graphs
call setvwrmode("one"); // Creates one window for all graphs
pqgwin many; // Creates a new window for each graph
call setvwrmode("many"); // Creates a new window for each graph
```

A graphics window has a menu, but no other special controls. The main menu options are **File**, **Edit**, **View**, and **Convert**.

## 8. GRAPHICS

### 8.2.1 Menus

#### File Menu

The **File** menu has options for printing and saving the graph.

**Print Setup** Opens a setup dialog where you can specify the parameters for printing the graph.

**Print** Prints the graph to the printer or a file, according to the parameters specified in **Print Setup**.

#### Edit Menu

The **Edit** menu lets you copy the graph to a bitmap (Ctrl-C) or metafile (Ctrl-D).

#### View Menu

The **Options** menu item in the **View** menu controls color conversion, printing, and cursor coordinates.

*Occasionally users will report that their graph shows nothing. This nearly always occurs because a graph's background color matches the color of the lines. Playing with the Vector Color Conversion settings (including possibly using a Custom Color Map) usually solves this problem.*

The **View** menu also lets you zoom in and out of the displayed graph. Zoom in by moving the mouse while holding down the right button. When done, press the left button (while holding the right button down). Abort the zoom by pressing **Esc**.

You can zoom into a graph multiple times, but zoom out only once.

#### Convert Menu

The **Convert** menu has a submenu, allowing you to select from the following conversion formats:

- Enhanced Metafile
- Enhanced PostScript
- HPGL Plotter
- Windows Bitmap (DIB)

The **tkf2ps** and **tkf2eps** functions let you convert **.tkf** files to Postscript and Encapsulated Postscript files at run-time. The syntax is:

```
ret = tkf2ps("mytekfile.tkf", "mypsfile.ps");
ret = tkf2eps("mytekfile.tkf", "myepsfile.eps");
```

## 8.3 Using PQG Graphics

There are four parts to a graphics program, the header, the datasetup, the graphics format setup, and the graphics call. These elements should be in any program that uses graphics routines. Graphics format setup often involves setting coordinates and configuring graphics panels.

### 8.3.1 Header

The header includes a **library** command to make the **pgraph** library active (it must be active to use the graphics routines) and, typically, a **graphset** command to reset the graphics global variables to their default state. For example:

```
library mylib, pgraph;
graphset;
```

### 8.3.2 Data Setup

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

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

### 8.3.3 Graphics Format Setup

Most of the graphics procedures contain default values that allow graphs to be generated. These defaults may be overridden through the use of global variables and graphics procedures. Changeable elements include axis numbering, labeling, cropping, scaling line symbol sizes and types, legends, and colors. The *GAUSS User's Guide* contains a complete list of graphics related global variables and their use.

### 8.3.4 Graphics Coordinate System

PQG uses a 4190 x 3120 pixel grid on a 9.0 x 6.855 inch printable area. Three units of measure are supported with most of the graphics global elements:

## 8. GRAPHICS

### Inch Coordinates

Inch coordinates are based on the dimensions of a full-size 9.0 x 6.855 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.

Some global variables allow coordinates to be input in inches. Coordinate values in inches and used in a graphics panel are scaled to window inches and positioned relative to the lower left corner of the graphic panel. A graphic panel inch is a true inch in size only if the graphic panel is scaled to the full window. Otherwise x and y coordinates are scaled relative to the horizontal and vertical graphic panel sizes respectively.

### Plot Coordinates

Plot coordinates refer to the coordinate system of the graph, in units of the user's X, Y, and Z axes.

### Pixel Coordinates

Pixel coordinates refer to the 4096 x 3120 pixel coordinates of a 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.

### 8.3.5 Graphic Panels

Multiple graphic panels are supported. These let the user display multiple graphs on one window or page, given that one window display is chosen using **pggwin** or **setvwrmode** (see section 8.2).

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

#### Tiled Graphic Panels

Tiled graphic panels do not overlap. Use the **window** command to divide a window into any number of tiled graphics panels. **window** takes three parameters: number of rows, number of columns, and graphic panel attribute (1 = transparent, 0 = non-transparent).

This example divides the window into six equally sized graphic panels. There are two rows of three graphic panels; three graphic panels are in the upper half of the window and three in the lower half. The attribute value of 0 is arbitrary since no other panels are beneath the 6 displayed panels.

```
/* the syntax is: window(nrows,ncols, attr) */
window(2,3,0);
```

### Overlapping Graphic Panels

Overlapping graphic panels are laid on top of each other as they are created. An overlapping graphic panel is created with the **makewind** command.

This example creates an overlapping graphic panel that has a 4 inch horizontal length and a 2.5 inch vertical length. The panel is positioned 1 inch from the left edge of the page and 1.5 inches from the bottom of the page. It is nontransparent:

```
/* the syntax is: makewind(hsize,vsize,hpos,vpos,attr) */
window(2,3,0);
makewind(4, 2.5, 1, 1.5, 0);
```

### Nontransparent Graphic Panels

A nontransparent graphic panel is blanked before graphics information is written to it. Information in previously drawn graphic panels lying under it will not be visible.

### Transparent Graphic Panels

Transparent graphic panels are not blanked, i.e. the graphic panel beneath a transparent panel is seen. Transparent graphic panels are used to add text or to superimpose one graphic panel on top of another. They let lines, symbols, arrows, error bars, and other graphics objects extend from one graphic panel to the next. To do this, first create the desired graphic panel configuration. Then create a full-window, transparent graphic panel using a **makewind** or **window** command. Set global variables to position the desired object(s) on the transparent graphic panel. Use the **draw** procedure to draw it. This graphic panel will be a transparent **overlay** on top of the other graphic panels.

## 8.3.6 Using Graphic Panel Functions

The following is a summary of the graphic panel functions:

- begwind**      Graphic panel initialization procedure.
- endwind**     End graphic panel manipulation, display graphs.

## 8. GRAPHICS

|                 |                                                              |
|-----------------|--------------------------------------------------------------|
| <b>window</b>   | Partition the window into tiled graphic panels.              |
| <b>makewind</b> | Create a graphic panel with the specified size and position. |
| <b>setwind</b>  | Set to the specified graphic panel number.                   |
| <b>nextwind</b> | Set to the next available graphic panel number.              |
| <b>getwind</b>  | Get the current graphic panel number.                        |
| <b>savewind</b> | Save the graphic panel configuration to a file.              |
| <b>loadwind</b> | 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 */
 /* two rows and two columns */
xsize = 9/2; /* Create a graphic panel that overlaps */
ysize = 6.855/2;
makewind(xsize, ysize, xsize/2, ysize/2, 0);
x = seqa(1,1,1000); /* Create the X data */
y = (sin(x) + 1) * 10; /* Create the Y data */
setwind(1); /* Graph #1, upper left corner */
 xy(x,y);
nextwind; /* Graph #2, upper right corner */
 logx(x,y);
nextwind; /* Graph #3, lower left corner */
 logy(x,y);
nextwind; /* Graph #4, lower right corner */
 loglog(x,y);
nextwind; /* Graph #5, center and overlaid */
 bar(x,y);
endwind; /* End graphic panel processing. Display them */
```

### Calling Graphics Routines

Input to a graphics routines consists of data and global variables. Three examples follow. The first two are different versions of the same graph. The variables beginning with `_p` are global control variables (A detailed description of these variables is in the *GAUSS User's Guide*).

**Example 1** This is a simple XY plot that uses the entire window. Four sets of data are plotted. The line type and symbols at each data point are automatically selected. The graph includes a legend, a title, and a time/date stamp (time stamp is on by default):

```

library pgraph; /* activate the PGRAPH library */
graphset; /* reset the global graphics variables */
_plctrl = -1; /* ensure that no lines connect the points */
x = rndn(8,1); /* generate the data */
y = rndn(8,1);
_plegctl = 1; /* legend on */
title("Example xy Graph"); /* The main title */
xy(x,y); /* Call the xy procedure */

```

**Example 2** Two graphics panels are drawn in this example. The first is a full-sized surface representation. The second is a half-sized inset containing a contour of the same data. The second panel is located in the lower left corner of the window:

```

library pgraph; /* activate the PGRAPH library */
x = seqa(-10, 0.1, 71)'; /* generate the data. x is a row vector */
y = seqa(-10, 0.1, 71); /* y is a column vector */
z = cos(5*sin(x) - y); /* z has dimension 71 x 71 */
begwind; /* initialize graphics */
makewind(9, 6.855, 0, 0, 0); /* the first panel - full size */
makewind(9/2, 6.855/2, 1, 1, 0); /* the second panel - half size */
setwind; /* activate the first panel */
graphset; /* reset the global graphics variables */
_pzclr = {1, 2, 3, 4}; /* set the z level colors */
title("cos(5*sin(x) - y)"); /* The main title */
ylabel("Y Axis"); /* Y axis label */
xlabel("X Axis"); /* X axis label */
scale3d(miss(0,0), miss(0,0), -5|5); /* scale the z axis */
surface(x,y,z); /* call the surface routine */

nextwind; /* activate the second panel */
graphset; /* reset the global variables */
_pzclr = {1, 2, 3, 4}; /* set the z level colors */
_pbox = 15; /* a white border */
contour(x,y,z); /* call the countour routine */
endwind; /* display the panels */

```

The additional routines **begwind**, **endwind**, **makewind**, **nextwind**, and **setwind** are used to control the graphics panels.

As Example 2 illustrates, the code between graphic panel functions (**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.

## 8. GRAPHICS

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 in the **examples** subdirectory. There are many of them, including include psur\*.e and px\*.e.

### 8.3.7 Saving Graphic Panel Configurations

The functions **savewind** and **loadwind** save graphic panel configurations, i.e. the global variables containing information about the current graphic panel configuration. Load this configuration again with **loadwind**.

## 8.4 Graphics Text Elements

Graphics text elements (e.g. titles, messages, axes labels, axes numbering, and legends) can be modified and enhanced by changing fonts and by adding superscripts, subscripts, and special mathematical symbols.

Escape codes in text strings enable this functionality. They are passed to **title**, **zlabel**, **ylabel**, and **asclabel**, or assigned to **\_pmsgstr** and **\_plegstr**.

The escape codes are:

- \000** String termination character (null byte).
- [** Enter superscript mode, leave subscript mode.
- ]** Enter subscript mode, leave superscript mode.
- @** Interpret the next character as a literal.
- \20n** Select font number n

The escape code **\l** can be embedded into title strings to create a multiple line title, as in:

```
title ("This is the first line\lthis is the second line");
```

A null byte separates strings in **\_plegstr** and **\_pmsgstr**:

```
_pmsgstr = "First string\000second string\000Third string";
_plegstr = "Curve 1\000Curve 2";
```

Use **[.]** to create the expression  $M(t) = E(e^{tx})$ , as in:

```
_pmsgstr = "M(t) = E(e[tx])";
```

### 8.4.1 Selecting Fonts

Four fonts are supplied with the Publication Quality Graphics system: Simplex, Complex, Simgrma, and Microb. The *GAUSS User's Guide* shows the characters available in each font.

The **fonts** command must be called before any of the fonts can be used in text strings. Load fonts by passing a string containing the names of all fonts to be loaded to the **fonts** procedure. To load all the fonts:

```
fonts ("simplex complex microb simgrma");
```

A loaded font is selected by embedding an escape code of the form `\20n` in the string to be written in the new font. The `n` is 1, 2, 3, or 4, depending on the order in which the fonts were loaded. 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
```

Examples for selecting different fonts are:

```
title("\201This is the title using the Simplex font");
xlabel("\202This is the label for X using the Complex font");
ylabel("\203This is the label for Y using the Microb font");
```

Once a font is selected, all succeeding text will use that font until another font is selected. A default font (Simplex) is loaded and selected automatically if no fonts are selected.

### 8.4.2 Greek and Mathematical Symbols

The Simgrma font allows for the display of Greek and Mathematical symbols. The following examples assume that Simgrma was the fourth font loaded. Simgrma characters are specified by either

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

## 8. GRAPHICS

The *GAUSS User's Guide* displays the available Simgrma characters and their numbers.

The following string produces the title  $f(x) = \sin^2(\pi x)$ :

```
title("\201f(x) = sin[2](\204p\201x)");
```

The `p` (character number 112) corresponds to  $\pi$  in Simgrma (i.e. we use the second way to display a Simgrma character, the regular text character with the same number).

To number the major X axis tick marks in 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 numbers the X axis tick marks with the labels  $\mu^{-2}$ ,  $\mu^{-1}$ ,  $1$ ,  $\mu$ , and  $\mu^2$ :

```
lab = "\204m\201[-2] \204m\201[-1] 1 \204m m\201[2]";
asclabel(lab,0);
```

8. *GRAPHICS*

## Chapter 9

# TGAUSS - The Command Line Interface

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

Programs often run faster in TGAUSS because it does not have a GUI. In addition, batch files may be run using TGAUSS.

The format for using TGAUSS is:

```
tgauss flag(s) program1 program2...
```

Different flags are:

- b** Execute file in batch mode and exit. 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.
- e expression** Executes a **GAUSS** expression. This command is not logged when **GAUSS** is in batch mode.
- o** Suppresses the sign-on banner (output only).
- T** Turns the dataloop translator on.
- t** Turns the dataloop translator off.

## 9.1 Interactive Commands

The **quit** command exits TGAUSS. You can also use the **system** command to exit TGAUSS from either the command line or a program.

The **ed** command opens an input file in an external text editor (see **ed** in the **GAUSS** Language Reference.) The format for **ed** is: **ed filename .**

The **browse** command lets you 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 .**

The **config** command gives you access to the configuration menu, letting you change the way **GAUSS** runs and compiles files. The format for **config** is: **config .** The configuration menu contains the following items:

### *Run Options*

**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 Options*

**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 declare warning messages during compiling.

### **Compiler Trace**

**Off** Turns off the compiler trace function.

**File** Traces program file openings and closings.

**Line** Traces compilation by line.

## 9. TGAUSS - THE COMMAND LINE INTERFACE

**Symbol** Creates a report of procedures and the local and global symbols they reference.

TGAUSS has an excellent command line source level debugger. The **debug** command starts the debugger. The format for **debug** is: **debug filename .** Several options control the **debug** process:

### *General Functions*

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

### *Listing Functions*

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

### *Execution Functions*

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

**filename line cycle** The debugger will stop every cycle times it reaches the specified line in the named file.

## 9. TGAUSS - THE COMMAND LINE INTERFACE

**filename line** The debugger will stop when it reaches the specified line in the named file.

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

**line cycle** The debugger will stop every cycle times it reaches the specified line in the current file.

**filename** The debugger will stop at every line in the named file.

**line** The debugger will stop when it reaches the specified line in the current file.

**procedure cycle** The debugger will stop every cycle times it reaches the first line in a called procedure.

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

**j** [[args ]] Executes code to a specified line, procedure, or cycle in the file without stopping at breakpoints. The optional arguments are the same as g, listed above.

**jx number** Executes code to the execution count specified (number) without stopping at breakpoints.

**o** Executes the remainder of the current procedure (or to a breakpoint) and stops at the next line in the calling procedure.

### *View Commands*

**v** [[ vars ]] Searches for (a local variable, then a global variable) and displays the value of a specified variable.

**v\$** [[ vars ]] Searches for (a local variable, then a global variable) and displays the specified character matrix.

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

**r** Specifies the number of rows to be shown.

**c** Specifies the number of columns to be shown.

**index,index** 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.

## 9. TGAUSS - THE COMMAND LINE INTERFACE

### *Breakpoint Commands*

- lb** Shows all the breakpoints currently defined.
- b** **[[args** **]]** Sets a breakpoint in the code. The syntax for each optional argument is:
- filename line cycle** The debugger will stop every cycle times it reaches the specified line in the named file.
  - filename line** The debugger will stop when it reaches the specified line in the named file.
  - filename ,, cycle** The debugger will stop every cycle times it reaches any line in the named file.
  - line cycle** The debugger will stop every cycle times it reaches the specified line in the current file.
  - filename** The debugger will stop at every line in the named file.
  - line** The debugger will stop when it reaches the specified line in the current file.
  - procedure cycle** The debugger will stop every cycle times it reaches the first line in a called procedure.
  - procedure** The debugger will stop every time it reaches the first line in a called procedure.
- d** **[[args** **]]** Removes a previously specified breakpoint. The optional arguments are the same arguments as b, listed above.

9. *TGAUSS - THE COMMAND LINE INTERFACE*

# Index

% , 72  
\* , 71, 72  
\* ~ , 73  
.\* , 2, 71  
\*. , 72  
+ , 71  
- , 71  
/ , 53, 72  
./ , 72  
^ , 72

:: , 97

/= , 86  
== , 86  
> , 86  
>= , 86  
< , 86  
<= , 86

**browse**, 112  
**config**, 112  
**ed**, 112  
**lib**, 8  
**quit**, 112  
**system**, 112

## A \_\_\_\_\_

**and**, 88  
atog, 41

## B \_\_\_\_\_

batch mode, 111  
**break**, 96  
breakpoints, 33  
building libraries, 8

## C \_\_\_\_\_

**call**, 94  
**cdir**, 6  
**changedir**, 6  
character matrix, 41, 63, 82  
**chdir**, 6  
**close**, 42, 47  
**code**, 52  
command log, 9  
compiler trace, 28, 112  
concatenation, matrix, 73, 74  
concatenation, strings, 74  
configuration, 9  
Configure Menu, 26  
conformability, 71  
**continue**, 96  
covariance matrix, 42  
**create**, 40  
Ctrl+F1 help, 16  
current directory, 6

## D \_\_\_\_\_

data types, 5, 65  
dataloop, 111  
dataloop translator, 10  
dataset, 39  
dBase, 51  
Debug Menu, 32  
DLLs, 9  
**do** loop, 1, 42, 52, 95  
dynamic libraries, 9

## E \_\_\_\_\_

**else**, 96  
 empty matrix, 69  
**endata**, 52  
**endif**, 96  
**eof**, 42  
**eq**, 86  
**eqv**, 88  
 error log, 9  
 Excel, 51  
 $E \times E$ , 71  
 exponentiation, 72  
**export**, 53  
**exportf**, 53  
**extern**, 52  
**eye**, 69

## F \_\_\_\_\_

F1 help, 16, 90  
 factorial, 72  
**fgets**, 48  
**fgetsa**, 48  
**fgetsat**, 48, 50  
**fgetst**, 48  
 file handle, 42  
**\_fmtcv**, 93  
**\_fmtnv**, 93  
**fopen**, 50  
**for** loop, 95  
**fputs**, 47  
**fputst**, 47

## G \_\_\_\_\_

gauss.cfg, 6, 7, 8, 9, 45, 56, 61  
**ge**, 86  
 generalized least squares, 1  
**getname**, 41  
**getnr**, 43  
 global variables, 59  
 graphics, 99  
**gt**, 86

## H \_\_\_\_\_

hat operator, 72  
 help, 13  
 horizontal direct product, 73

## I \_\_\_\_\_

identity matrix, 69  
**if**, 96  
**import**, 53  
**importf**, 53

## K \_\_\_\_\_

**keep**, 52  
 Kiss-Monster, 70  
 Kronecker, 72

## L \_\_\_\_\_

**le**, 86  
 least squares, 1  
**lib**, 55  
**lib\_path**, 11  
**library**, 59  
 library file path, 9  
 library files, 6, 9, 11  
 library path, 11  
 Library Tool, 55  
 linear equation solution, 72  
**Infact**, 72  
**load**, 48  
 logical comparison, 86  
 Lotus, 51  
**lt**, 86

## M \_\_\_\_\_

matrix file, 45  
**maxc**, 73  
**meanc**, 41  
**minc**, 73  
**miss**, 85  
 missing values, 85  
 modulo division, 72  
 multiplication, 72

INDEX

N \_\_\_\_\_

**ne**, 86  
**not**, 88  
null matrix, 69  
numeric matrix, 67

O \_\_\_\_\_

**olsqr**, 1  
**ones**, 69  
**open**, 42  
**or**, 88  
outer product, 71  
**output on/off/reset**, 47

P \_\_\_\_\_

**packr**, 42, 85  
Paradox, 51  
path specification, 66  
**printfm**, 94  
**printfmt**, 94  
printing character matrices, 6  
**proc**, 1  
procedure, 1, 97  
pseudo-random, 70

Q \_\_\_\_\_

Quattro, 51  
quotation marks, 66

R \_\_\_\_\_

random, 70  
**readr**, 42  
**recode**, 52  
regression, 1  
relational operator, 86  
**reshape**, 73  
**rev**, 73  
**rndKMn**, 70  
**rndKMu**, 70, 96  
**rows**, 42  
**rowsf**, 42, 43

S \_\_\_\_\_

**save**, 45, 46  
**saved**, 39  
scalar, 71, 88  
**screen on/off**, 47  
search order, 6, 61  
**seekr**, 44  
**seqa**, 69  
sequence, 69  
shortcuts, keyboard, 13  
simulation, 40  
**sortc**, 73  
**sortind**, 73  
sorting matrices, 73  
**sortmc**, 73  
Special Characters, 66  
spreadsheets, 51  
**src\_path**, 9, 11  
state, 96  
status bar, 21  
**stof**, 50  
**strindx**, 50  
string, 65  
string array, 65, 82  
string file, 45  
**strsect**, 50  
structures, 5  
submatrices, 68  
**sumc**, 42  
symbol table, 5, 6, 19, 36, 57, 58  
symbols, 6  
Symphony, 51  
**sysstate**, 9, 10

T \_\_\_\_\_

table, 71  
tensor, 72  
text file, 41, 48  
**token**, 50  
trace, 28, 112  
transpose, 73, 74  
**trimr**, 74  
type conversion, 82

V \_\_\_\_\_

**vec**, 74

**vecr**, 74

View Menu, 26

W \_\_\_\_\_

watches, 33

weighted least squares, 1

working directory, 6

**writer**, 40

X \_\_\_\_\_

**xor**, 88

Z \_\_\_\_\_

**zeros**, 69