Learning Econometrics with GAUSS

by

Ching-Fan Chung

Institute of Economics, Academia Sinica

♠ · ♡ · ◇ · ♣

Contents

Chapter 1

Introduction

GAUSS is a computer programming language. We use it to write programs, i.e., collections of commands. Each command in a GAUSS program directs a computer to do one mathematical or statistical operation. The main difference between the GAUSS language and the other statistics packages is that using GAUSS requires us to know more about the mathematical derivation underlying each statistical procedure because we have to translate it into GAUSS language. The advantage of using a programming language is the maximum flexibility it allows for. There is virtually no limit in what we can do with GAUSS. The main difference between GAUSS and other programming languages, such as Fortran, C, Pascal, etc. is that the basic elements for mathematical operations in GAUSS are matrices. This is a powerful feature that can save us from many programming details and allow us to concentrate more on statistics and econometrics.

Let's use an example to explain this feature. Consider the following equation:

```
1  y = X*b + e;                                    /* 1.1 */
```

If we see equation `/* 1.1*/` in an econometrics class, then it is quite natural to associate it with the following "handwritten" equation:

$$y = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}. \tag{1.1}$$

It is almost like our instinct to think '`y`' in equation `/* 1.1 */` as a column vector, containing observations on the dependent variable, '`X`' a matrix, containing observations on some explanatory variables, '`b`' a column of parameters, and '`e`' of course a column of error terms.[1] In other words, we will interpret equation `/* 1.1 */` just like we interpret equation (1.1). The only difference is simply the style: equation (1.1) is in a more elaborated font where Greek letter $\boldsymbol{\beta}$ is used, while equation `/* 1.1 */` is written in plain English letters. Perhaps we are not so sure about why there is an asterisk '`*`' between '`X`' and '`b`' in `/* 1.1 */`. But we might just guess that it means matrix multiplication. We may also wonder why in `/* 1.1 */` there is a semicolon after '`e`' and why the equation number is written in a strange way like `/* 1.1 */`. These and many other questions about styles will be explained fully in the next two chapters. But no matter how strange we may feel about the expression in `/* 1.1 */`, the important thing is that we can always guess its meaning while those unusual details generally do not bother us very much. In fact, the interpretation of `/* 1.1 */` is so natural that we may wish computers can understand it just like we do. Fortunately, computers do understand it, but only in GAUSS.

GAUSS is a computer language in which expressions are very close to their hand-written counterparts. As a result, GAUSS minimizes our effort to translate a handwritten mathematical expression to a form that computers can understand. To appreciate how much trouble GAUSS has saved us from, we have to know the nature of computers. Other than the capability of doing some basic arithmetic in high speed, computers actually are very dumb. We have taken education for years to become what we are now. Computers never

[1] Whenever elements of a GAUSS program, such as '`y`', '`X`', '`b`', '`e`', appear in the text, they will be in a special font, that differs from the standard text font, and enclosed by single quote ' and '.

really "learn" anything. Just imagine how difficult it would be to explain things like vectors and matrices to a kindergarten kid. By the same token, to explain the meaning of econometric equations in matrix form to computers is not an easy job. For instance, computers need to be informed that a letter like 'y' should be understood as a mathematical item called vector that follows certain rules. Nevertheless, GAUSS saves us all these troubles. When we write equation `/* 1.1 */` in GAUSS, computers will understand it just like the way we want them to.

This book is written for a person who has little knowledge about computer programming to get on with GAUSS as quickly as possible. Every thing in GAUSS is explained from scratch. However, some knowledge about DOS, the basic operation system of the IBM-compatible computers, is assumed. GAUSS has gone through several editions. The edition we discuss here is GAUSS-386i version 3.2.

1.1 Getting Started with GAUSS

The procedure of using GAUSS is as follows: we first type our GAUSS program in a text file and then submit this file to a personal computer for processing. Once the computer fully understands what we want to do in the GAUSS program, the program is executed. This process is called *to have a computer run a GAUSS program in the edit mode*. (There is an alternative way of running a GAUSS program called running it in the command mode, which is seldom used when the GAUSS program contains more than five statements.)

To type the GAUSS program into a text file, we need a wordprocessor (e.g., WordPerfect, Word, AMI, etc.) or an editor (e.g., the DOS editor, the GAUSS editor, etc.). No matter which wordprocessor or editor we use, we have to make sure the file we create is a plain ASCII file without any *formatting codes*. Wordprocessor users are especially cautioned here. Almost all wordprocessors embed formatting codes in files they create. These formatting codes explain why files created by one wordprocessor cannot be read directly by other wordprocessors. However, almost all wordprocessors, under some special directions, can create ASCII files. So we can use any wordprocessor to write GAUSS programs if we know how to get rid of those formatting codes and leave a plain ASCII file. Editors work like scaled-down wordprocessors. But in general the files created by editors are plain ASCII files. Both *the DOS editor* and *the GAUSS editor* are quite good for the purpose of typing GAUSS programs and each can be learned in less than half an hour. The GAUSS editor will be briefly introduced in the next section. (However, for those people who do not want to use the GAUSS editor, that section may be skipped without affecting the continuity of the discussion.)

Once we have finished typing the GAUSS program in an ASCII file with a file name, say, `prg.1`, we then submit it for execution. To do this, first we have to go into *the GAUSS environment* from *the DOS environment*. That is, under DOS, we type

```
1  gaussi
```

after the DOS prompt '>' and press the 'Enter' key. A few seconds later, some information about the status of GAUSS, such as the size of usable memory, will appear on the screen and we are in the GAUSS environment. The GAUSS prompt '>>' will show up on the screen waiting for us to type GAUSS commands. In GAUSS terminology, the GAUSS environment we are in is referred to as the GAUSS command mode. We can directly type GAUSS statements on the screen and execute them (a procedure that is called running GAUSS in the command mode). But this is not most convenient way to use GAUSS. We should instead type a GAUSS program in an ASCII file first and then submit it for execution in the edit mode, as briefly

described at the beginning of this section. That is, when our GAUSS program in the file, says, `prg.1` is ready, we type

```
run prg.1
```

and press the 'Enter' key after the GAUSS prompt '>>' . GAUSS will then start to *compile* (translate it to machine codes that computers really understand) the file 'prg.1.' If there is no error in the program and the compilation process gets through, then the program will be executed immediately after it is compiled. This whole procedure is called *running a GAUSS program in the edit mode*. After the program is executed, we get either the result we want or some error messages on the screen. If the program `prg.1` needs to be revised, we can call up the wordprocessor or the editor from inside the GAUSS by preceding the usual command with three additional letters 'dos'. For examples, if we use the DOS editor to revise the file, the standard command under DOS is

```
edit prg.1
```

But if we are in the GAUSS command mode, then the command changes to

```
dos editor prg.1
```

after the prompt sign '>>'.

1.1.1 Executing DOS Commands in GAUSS

We can access all the DOS commands from the GAUSS command mode. In fact, other than running GAUSS programs, what we do in the command mode are mostly DOS related activities such as copying or deleting files, making or changing directories, etc. To do these, we simply precede all the usual DOS commands with 'dos'. For example, to copy the file `prg.1` to another file `prg.2`, we type

```
dos copy prg.1 prg.2
```

There is a big advantage in executing DOS commands in the command mode: all the commands will stay on the screen so they can be revised or reused. You do not have to retype the entire command again and again when the same command is to be repeated.

1.1.2 Some GAUSS Keystrokes

All the commands to be executed in the command mode need to be preceded by the GAUSS prompt '>>'. If there is no GAUSS prompt on the screen, press the F3 key to create one. There are some other useful keys in the command mode:

- The four arrow keys: to move the cursor around the screen.
- The 'Home' keys: to go to the end of the line and then the end of the screen.

- The 'Backspace' key: to delete a character to the left.
- The 'Del' key: to delete a character.
- The 'Escape' key: to exit form the command mode and go back to DOS.
- F1: to recall the previous screen.
- Ctrl-F2: to execute the file that was last run.
- Ctrl-Enter (pressing the 'Ctrl' and 'Enter' keys simultaneously): to add a blank line.
- Ctrl-Home: to clear the screen.
- Ctrl-N: to add a blank line.
- Alt-D: to delete a line.
- Alt-H: to access the On-Line Help, which provides a fairly detailed description of all GAUSS commands. The use of On-Line Help is quite self-explanatory. After Alt-H is pressed, a help screen will be displayed. Pressing 'H' again will give us the prompt 'Help On:' at the bottom of the screen. It is from this prompt we can access all other On-Line Help information. More about On-Line Help will be discussed in sections 4.5 and 9.5.

1.1.3 A Note on Computer Memory

GAUSS can automatically access all the memory in the computer. 4MB memory is the minimum requirement for GAUSS-386 version 3.2. Occasionally, the "insufficient memory" problem may occur. Other than adding more memory chips to the computer, an easier remedy is use another version of GAUSS: GAUSS-386 VM. The letters "VM" means it can transform the hard disk space to *Virtual Memory* - a kind of simulated memory. The disadvantage of using virtual memory is that computation slows down considerably.

1.2 The GAUSS Editor

This section presents a brief explanation of the GAUSS editor. It is a part of the GAUSS that is used to typed and revise the GAUSS program in an ASCII file. Although we may use a wordprocessor or some other editor like the DOS editor for such tasks, the advantage of using GAUSS editor is that the GAUSS program can be submitted for execution directly from inside the GAUSS editor. The commands described here are not complete but will get almost all your editing jobs done. To use the GAUSS editor to edit an ASCII file, say, prg.0 in the command mode, we type

```
1  edit prg.0
```

and press the 'Enter' key after the GAUSS prompt '>>' . The content of the file 'prg.0', if any, will appear on the screen ready for editing. All the keys for the command mode described in the previous section still work inside the GAUSS editor and there are many more. Let's first consider an important feature of the GAUSS editor – blocking. Multi-line text in a file can be *blocked* for special uses. To block off a section

of text, press Alt-L at both ends. The blocked text will then be highlighted. The blocked text can be copied or moved to other places in the file. To do this, either press the '+' key in the numeric keypad to *copy* the blocked text to scrap (which is a temporary storage place outside the file), or press the '-' key in the numeric keypad to *move* the blocked text to scrap. After the scrap is filled with some data, then move the cursor to another place and press the 'Ins' key to retrieve the blocked text from the scrap. As such the blocked text can be copied or moved to any place in the file. The blocked text can be further manipulated by the following keystrokes:

- Alt-W: to copy the blocked text to another file.
- Ctrl-X: to execute the blocked text.
- Alt-P: to print the blocked text.
- The 'Del' key: to delete the blocked text.

Text can be searched and replaced using the GAUSS editor:

- F5: for searching.
- F6: for searching and replacing.

After we finish editing, there are three ways to exit the GAUSS editor and go back to the command mode:

- Alt-X: a menu of options will show up for selection.
- F1: to save the file and exit.
- F2: to save the file and then execute it.

While in the command mode after editing a file, there are several keys related to the GAUSS editor:

- Shift-F1: to directly go back to the last edited file for additional editing.
- Shift-F2: to execute the last edited file again automatically.
- Ctrl-F1: to edit the last run file automatically, given that a file has just been run.
- Ctrl-F3: to edit the last output file automatically, given that a file has just been run which produces an output file.

These keystrokes may be difficult to remember at first. But just a few exercises can change such feeling completely.

1.3 GAUSS Statements

Equation /* 1.1 */ in page 1 is a typical GAUSS statement which contains an equality sign. It is the GAUSS counterpart of the handwritten equation (1.1). When we write down an equation like (1.1) on scratch paper, the exact values of y, $\mathbf{X}$, $\boldsymbol{\beta}$, and $\boldsymbol{\varepsilon}$ do not really concern us. However, when we type equation /* 1.1 */ in a GAUSS program, we need to be very specific about the values in the matrices 'y', 'X', 'b', and 'e' as to what exactly are contained in each matrix: how many variables, how many observations, the

format of the numbers as integers or as real numbers; etc. Because of this, almost all GAUSS programs start with GAUSS statements that assign data to matrices which can be operated in the latter part of the program. Here, it is important to know that whenever a new matrix is defined, the numbers of its column and row should be firmly remembered. Operating on unconformable matrices with incompatible dimensions is the most common mistake in writing GAUSS programs.

Suppose we have input data into the three matrices 'X', 'b', and 'e,' so that we know exactly the size of each of these matrices. With this information, we should also know whether they are suitable for operations like matrix multiplication '*' and matrix addition '+'. If they are, then the operations on the right-hand side of /* 1.1 */ should produce a result that can be equal to 'y' on the left-hand side.

Now, instead of thinking the result from the right-hand side *is equal to* the left-hand side matrix 'y', as the equality sign '=' implies, we should interpret the equality sign '=' as an *assignment* command: 'y' is assigned with the result of the right-hand side operation. The reason for having this new interpretation is because it is how computers interpret the equality sign. We note this interpretation does not change the fact that the contents of both sides of the equality sign are equal. Many GAUSS statements contain equality signs and their interpretations should always be assignment.

1.3.1 Some Syntax Rules

Semicolons are used to end statements. Extra spaces can be inserted between items, such as 'y', '=', 'X', '*', 'b', '+', and 'e' in equation /* 1.1 */, to make statements more legible. All extra spaces between items are ignored. Empty lines between GAUSS statements are also allowed. More than one statement can be typed in one line, though we usually leave one statement in one line to make a program more readable. Also, at most 80 characters are allowed in each line. If a statement is too long to fit into one line, it can be continued in the next line.

Another way to make GAUSS programs more legible is to write *comments* in the program. To distinguish GAUSS statements, which can be executed by a computer, from comments, which is to be read by human only and ought to be ignored by the computer, we enclose all comments between two '@' or between '/*' and '*/'. For example, to number the equation we use the comment '/* 1.1 */' which can also be written as '@ 1.1 @'. It will be ignored by the computer.

Uppercase and lowercase in GAUSS make no difference. For example, we can freely interchange the uppercase 'X' in equation /* 1.1 */ with the lowercase 'x'. Symbols of variables can contain up to 32 characters (8 charaters prior to version 3.2) from 26 English letters, 0, 1, . . ., 9, and underscore '_'.

1.3.2 Two Types of Errors

If *syntax errors* are detected during the compilation process, GAUSS will immediately stop and report to the computer screen. Syntax errors mean anything that we erroneously type and are not recognizable to GAUSS. These errors are relatively easy for *debugging*, i.e., correcting. GAUSS usually gives us rather clear error messages on the screen.

There is another type of errors, the so-called *logic errors*, that are usually harder to spot. For example, suppose the calculation we intend is

$$z = (x + 10)y.$$

If in the GAUSS program we erroneously type

```
z = (x + 10*y;
```

then GAUSS will spot the error of missing right parenthesis and stop the execution of this GAUSS command. This is a syntax error. However, if what we type is

```
z = x + 10*y;
```

then there is no syntax error and what we have is a perfectly legitimate GAUSS statement. GAUSS will execute it as

$$z = x + 10y,$$

and assign 'z' a value that is not really what we want, which is $z = (x + 10)y$. This is a logic error. It is our responsibility to make certain a GAUSS program is free from logic errors.

It is quite common that earlier logic errors cause some syntax errors later in the program. For example, suppose the value of 'z', if calculated correctly, is expected to be positive. So in the latter part of the program we take square root of 'z'. When GAUSS tries to execute this command, a syntax error will result if 'z' was not computed correctly early on and had a negative value. Debugging such syntax errors may take more time because we have to trace back to the origin of the problem. However, the worst situation is that we make some logic errors that do not contradict to anything else in the program. The program can run without encountering any syntax errors but produce something we do not want. This is perhaps the worst thing that can happen to a GAUSS program. So it is usually quite necessary to test a complicated but syntax error-free GAUSS program with experimental data to guard against logic errors.

Chapter 2

Data Input and Output

Consider the definition of the error term in equation (1.1):

$$\boldsymbol{\varepsilon} = \mathbf{y} - \mathbf{X}\boldsymbol{\beta}. \tag{2.1}$$

Suppose we have data on the dependent variable $\mathbf{y}$ and a few explanatory variables $\mathbf{X}$. The parameter vector $\boldsymbol{\beta}$ is also known to us. If we want to compute the error vector $\boldsymbol{\varepsilon}$, then we use the GAUSS command

```
e = y - X*b;                                    /* 2.1 */
```

Let's assume the data consist of N observations and there are K explanatory variables. So the dimensions of the matrices $\mathbf{y}$, $\mathbf{X}$, and $\boldsymbol{\beta}$ are $n \times 1$, $n \times k$, and $k \times 1$, respectively. Suppose these data are recorded on a piece of paper, then the question is: how can we read these data into the matrices 'y', 'X', and 'b' in a GAUSS program?

2.1 ASCII Files

The most straightforward way to read data into matrices is through the direct assignment statements as follows:

```
y = {1,  3,  4.5,  -4,  5};

X = {1   4.2   6.1,
     1   3.9   2.7,
     1   2.4   0,
     1  -7.35  3.2,
     1   6.8   2.2};

b = {2.1,  0.3,  2.2};
```

The numbers on the right-hand side are our hypothetical data. Note that we have $n = 5$ and $k = 3$ here. (As mentioned earlier, keeping these dimensions in mind is important in writing GAUSS programs.) From the pattern the data is listed, it is easy to infer that commas separate rows, spaces separate columns, and all data are enclosed in braces.

There is no difference between the second assignment statement for 'X' and the following one:

```
X = {1  4.2  6.1, 1  3.9  2.7, 1  2.4  0, 1 -7.35 3.2, 1  6.8  2.2};
```

since extra spaces between items are ignored in GAUSS.

There is an equivalent way to define 'y', 'X', and 'b' using the 'let' command:

```
let y[5,1] = 1 3 4.5 -4 5;
let X[5,3] = 1 4.2 6.1 1 3.9 2.7 1 2.4 0 1 -7.35 3.2 1 6.8 2.2;
let b[3,1] = 2.1 0.3 2.2;
```

In these 'let' commands, the dimensions of matrices are explicitly specified and enclosed by brackets. Since the dimensions of matrices are known, it becomes unnecessary to use commas or braces on the right-hand side to separate data. Data will be assigned to a matrix *row by row*. This mechanism of feeding matrices with data *in rows* is typical in GAUSS. This convention is followed by many types of GAUSS operations as will be seen later.

There are three more conventions, or the so-called *defaults*, associated with the 'let' command when the 'let' command is not completely specified. First, if the dimension is not given, then a column vector will be assumed. For example, the statement

```
let a = 2 3 8 10 4;
```

creates a 5×1 column vector 'a'. Secondly, if only one data entry is provided, then this single entry will fill the entire matrix:

```
let a[3,8] = 0.7;
```

creates a 3×8 matrix of 0.7. Thirdly, if no entry is given, then 0 is assumed:

```
let a[2,5];
```

creates a 2×5 matrix of 0.

2.1.1 ASCII Data Files

Although the previous two methods for data input seem straightforward enough, there is a more flexible alternative. In this method, we first type the three sets of data in three different ASCII files with file names, says, `y.dat`, `x.dat`, and `b.dat`, respectively, while data in these files are listed in a matrix format. For example, in the ASCII file `x.dat`, we have:

```
1   4.2   6.1
1   3.9   2.7
1   2.4   0
1  -7.35  3.2
1   6.8   2.2
```

Note that no commas or braces are included. Given the three ASCII data files, we use the following three GAUSS commands to read data from them:

```
load y[5,1] = y.dat;
load X[5,3] = x.dat;
load b[3,1] = b.dat;
```

Again, the dimensions of matrices need to be explicitly specified in these 'load' commands. This data input method is the most common one because we do not always type data ourselves but obtain some ASCII data files from somewhere else.

In most ASCII data files, data are displayed just like those in the above x.dat example: different variables are listed in columns, which are separated by spaces, and observations are listed along rows. However, as mentioned earlier, GAUSS has the automatic mechanism of feeding matrices in rows. So the data in the ASCII file x.dat can actually be listed as

```
1   4.2   6.1  1  3.9  2.7  1  2.4  0
1  -7.35  3.2  1  6.8  2.2
```

As long as the dimension of 'x' in the 'load' command is correctly specified, data will be loaded into 'x' correctly – one row by another.

2.1.2 ASCII Output Files

After data have been read into the matrices 'y', 'X', and 'b', the assignment operation /* 2.1 */ can then be executed to create the error vector 'e'. The question now is how we can access the resulting values in 'e', either to read them or to store them for later uses. Consider the following GAUSS statements:

```
output file = residual.out on;
format /rd 10,4;
print e;
```

The 'output file' command in the first line *opens* (i.e., creates or retrieves) an ASCII file with the name 'residual.out', which can be any file name with extension that follows the standard rule for file names. The file 'residual.out' can be a new file or an existing file. If 'residual.out' is an existing file with some data already in it, then the subcommand 'on' causes new data, which we are about to produce, to be appended onto the end of this file without affecting those existing data. An alternative subcommand is 'reset' which resets the referred file so that all the existing data will be erased.

The 'format' command in the second line describes how the data should be listed in the output file. Its second subcommand '/rd' means the listed data are to be right-justified and the third subcommand '10,4' means in total ten spaces are reserved for each entry which is rounded to four digits after the decimal point. The '10,4' subcommand may be changed to suit different needs. A common one is '6,0' which means to list the values as integer numbers (without decimal points) over six spaces.

Although there are seven other alternatives, the '/rd' subcommand is used most often. Another common one is '/re', with which the value 0.012345 ($= 1.2345 \times 10^{-2}$) will be listed as 1.2345E-2 (given the other

subcommand is ‘6,4’). If left-justified listing is desired, change ‘r’ in the above subcommands to ‘l’. To find out more about the other possibilities, press Alt-H and then type ‘format’. On-Line Help for the ‘format’ command will appear. (There we can find a third subcommand which is much less used.)

The ‘print’ command in the third line means to list all the elements of ‘e’. The results will be listed both on the computer screen as well as in the output ASCII file residual.out using the format specified by the ‘format’ command. The ‘print’ command can be abbreviated as

```
1  e;
```

That is, we simply type the name of the matrix, followed by a semicolon. This simplified print command will be used throughout this book.

A GAUSS program can contain more than one print command. All the printed matrices will be included in the same output ASCII file and follow the format based on the ‘format’ command that is last executed.

Once the output ASCII file is created with data printed into it, we can use a word processor or an editor to view, revise, or print those data.

The data in the output ASCII file, like any ASCII data file, can also be read back into a matrix in a GAUSS program using the ‘load’ command as described earlier. For example, the entries listed in the ASCII file residual.out can be loaded back to the matrix ‘e’ as follows:

```
1  load e[5,1] = residual.out;
```

Note that when an ASCII data file is loaded, we have to make sure the number of data entries in the ASCII file matches the matrix size specified in the ‘load’ command. Again, no matter how the data entries are listed in the ASCII file, they will be read into the matrix row by row.

2.1.3 Other Commands Related to ASCII Output Files

Suppose we do not want the values in ‘e’ to be listed in any ASCII file and all we want is to read them on the screen, then we just skip the ‘output file’ command.

If an output ASCII file is already opened, it can be ‘closed’ by

```
1  output off;
```

The output file can be reopened again to accept new output entries by

```
1  output on;
```

If we want to list results from several operations at several places in a long program, we can open an ASCII file at the beginning of the program and then close and reopen it as often as we want.

If an empty line is to be included in the output file between two printed matrices, then between the two ‘print’ commands type:

```
1  print;
```

or, simply,

```
?;
```

By including many such commands, we can produce multiple empty lines. This is a useful technique which makes the output ASCII file easier to read.

If other than numbers we also want a string of characters to appear in the output file (usually as the titles of output entries or to give some explanations to the outputs), use the quotation command. For example, if we want a line like "The residuals are" to appear before the values of 'e', type

```
"The residuals are";
e;
```

Everything inside the quotation marks will appear in the output file. The semicolon after the quotation command can be omitted. In such a case, the first element of 'e' will be listed immediately after the word "are" in the same line.

If we only want to list the values of 'e' in the output ASCII file and do not want them to appear on the screen (this is sometimes needed to save time because printing on the screen can be time-consuming), we can add the following command before the print command:

```
screen off;
```

To turn the screen on again, type

```
screen on;
```

2.1.4 An Example

In this example we demonstrate how to use GAUSS to deal with real data that are in the ASCII format. The data are drawn from the monograph *International Evidence on Consumption Patterns* by Henri Theil, Ching-Fan Chung, and James Seale.[1] They consist of per capita consumption on 10 categories (or aggregate commodities) in 1980 for 60 countries. The 10 categories are

1. Food;
2. Beverages and Tobacco;
3. Clothing and Footwear;
4. Gross Rent and Fuel;

[1] *International Evidence on Consumption Patterns*, Henri Theil, Ching-Fan Chung, and James Seale, Greenwich, Connecticut: JAI Press, 1989.

5. House Furnishings and Operations;
6. Medical Care;
7. Transport and Communications;
8. Recreation;
9. Education; and
10. Other.

The data are in three ASCII files whose names are '`volume`', '`share`', and '`totalexp`', respectively. Both '`volume`' and '`share`' files contain 60×10 matrices. The 60 rows correspond to the observations on the 60 countries and 10 columns for 10 categories.

The data in the file '`volume`' are the volumes of per capita consumption (in terms of a set of standardized measurement units). These volumes can be considered as the *quantities* q_{ic}, $i = 1, \ldots, 10$ and $c = 1, \ldots, 60$, of the 10 commodities.

If in addition to these quantities, we also have *prices* p_{ic}, then we can define the *expenditures* on these 10 commodities simply by the products $p_{ic}q_{ic}$, from which we can also define the *total expenditures*:

$$m_c = \sum_{i=1}^{10} p_{ic}q_{ic}, \qquad c = 1, \ldots, 60.$$

The file '`totalexp`' is a 60×1 vector which contains the data on the 60 countries' total expenditure m_c. Note that a country's total expenditure m_c can also be referred to as it's income.

Finally, we note the *budget shares* of the commodities are defined by

$$s_{ic} = \frac{p_{ic}q_{ic}}{m_c}, \qquad i = 1, \ldots, 10, \quad c = 1, \ldots, 60.$$

The file '`share`' contains the 60 observations on 10 budget shares.

Suppose we want to read data from these three different ASCII files and then print them in a single ASCII file called `all.out` with some description. We use the editor to type the following GAUSS commands in an ASCII file, say, `try.1`:

```
load q[60,10] = a:\data\volume;
load s[60,10] = a:\data\share;
load m[60,1] = a:\data\totalexp;

output file = a:\all.out on;

"The Quantities of 10 Commodities from 60 Countries:";?;
format /rd 8,2;
q;?;?;?;

"The Budget Shares of 10 Commodities from 60 Countries:";?;
s;?;?;?;
```

```
13
14    "The Total Expenditure of 10 Commodities from 60 Countries:";?;
15    format /rd 15,0;
16    m;?;?;?;
17
18    output off;
```

This is a simple GAUSS program in which we assume the three files ‘volume’, ‘share’, and ‘totalexp’ are all located at the subdirectory ‘\data” of a diskette which is in drive a. After the program file try.1 is executed, the output file all.out will also go to the same diskette in drive a, but in the root subdirectory. We can use the editor to view the original ASCII data files ‘volume’, ‘share’, and ‘totalexp’, as well as the output file all.out and compare them.

The output in the output file all.out will be arranged in three blocks, separated by three empty lines. Each block has one line of simple description on the top. The entries in the input file ‘volume’ contain one digit after the decimal point, but there will be two digits after the decimal point in the output file, just like those entries in the input file ‘share’. In contrast, the original entries in the input file ‘totalexp’ contain two digits after the decimal point but they will appear as integers in the output file.

It is interesting to see what happen if you misspecify the dimensions of the input matrices as follows

```
1    load q[60,5] =  a:\data\volume;
2    load s[20,10] = a:\data\share;
3    load m[1,60] =  a:\data\totalexp;
```

or

```
1    load q[100,10] = a:\data\volume;
2    load s[600,1] =  a:\data\share;
3    load m[60,10] =  a:\data\totalexp;
```

Examining the error messages or the corresponding output file is informative. From these mistakes we learn how syntax errors or logic errors can be generated. One possible logic error here (which is common and potentially quite dangerous) is that the matrices we create may not contain the data we intend to have.

The three ASCII data files ‘volume’, ‘share’, and ‘totalexp’ will be used throughout this book as our leading example.

2.2 Matrix Files

The ASCII data file we have described so far is one of the three types of data files used in GAUSS. The second type of data files are referred to as *matrix files* while the third one is called *GAUSS data set files*. *Matrix files* are the easiest ones to handle and they will be introduced now. *GAUSS data set files* are more complicated and are designed for larger data sets. They will be discussed much later in appendix B.

If we are not immediately interested in viewing the values in the matrix ‘e’ and all we want is to save them for later uses, then the best way to output the matrices is to use the following command

```
save e;
```

The data in 'e' will be saved as a *matrix file* with the file name 'e.fmt', where the extension '.fmt' is automatically attached to the matrix name. In the GAUSS program we do not need to specify the format or dimension for matrix files. Data will be automatically stored with the maximum precision.

The disadvantage of storing data in matrix files is that they cannot be viewed with a word processor or an editor. To find out what are inside a matrix file, we have to first load the matrix file back to a matrix in a GAUSS program and then print it on the screen or in an ASCII file. However, to load a matrix file is quite easy. For example, to load the e.fmt file back, just type

```
load e;
```

It is possible to change the name of the matrix file when it is saved. For example, the command

```
save res = e;
```

will save data in the matrix file 'res.fmt'. The extension '.fmt' is again automatically attached. So when we specify the file name in the 'save' command, no extension should be included. If the file res.fmt is to be loaded back to a matrix with the name 'a', type

```
load a = res;
```

Since it is not necessary to type the extension '.fmt' or to specify the dimension of the matrix 'a', it is easier than loading an ASCII data file.

Consider the following simple example:

```
load q[60,10] = a:\data\volume;
load s[60,10] = a:\data\share;
load m[60,1] =  a:\data\totalexp;

save a:\data\volume = q,
     a:\data\share = s,
     a:\data\totalexp = m;
```

After these commands being executed, the directory a:\data\share will then contain three more files: volume.fmt, share.fmt, and totalexp.fmt. They are matrix files. They are different from the three original ASCII data files 'volume', 'share', and 'totalexp', which do not have the '.fmt' extension (although the contents are the same).

Chapter 3

Basic Algebraic Operations

3.1 Arithmetic Operators

Arithmetic operators are the easiest part of the GAUSS language because their notation is similar to the corresponding handwritten notation. The basic usage of the three arithmetic operators '+' (sum), '-' (subtraction), and '*' (multiplication) for matrices are defined in the usual way. For example, in the following program

```
let a[3,2] = 1  2  3  4  5  6;
let b[3,2] = 11  12  13  14  15  16;

c = a + b;
d = a - b;
```

The contents of 'c' and 'd' are easy to figure out. Also, we note that the two matrices 'a' and 'b' cannot be multiplied because they are not conformable: matrix multiplication requires the column number of the first matrix to be equal to the row number of the second matrix. The following program is legitimate

```
let a[3,2] = 1  2  3  4  5  6;
let b[2,5] = 11  12  13  14  15  16  17  18  19  20;

c = a*b;
```

The slash '/' is used for *matrix division*. The interpretation of the notation 'a/b' depends on the sizes of the two matrices 'a' and 'b'. If both of them are scalars, then 'a/b' means 'a' is divided by 'b'. If 'a' and 'b' are two matrices, then 'a/b' is defined to be "$\mathbf{a} \cdot \mathbf{b}^{-1}$" in handwritten form. That is, the result of 'a/b' is a matrix which is equal to the product of the matrix "**a**" and the inverted matrix "$\mathbf{b}^{-1}$". More about matrix inversion will be discussed later.

3.2 Element-by-Element Operations

One important feature of the GAUSS language is that GAUSS has extended the functionality of the arithmetic operators '+', '-', '*' and '/.' Although typically two matrices should have the same dimensions when a matrix is added to or subtracted from another matrix, in GAUSS a scalar (single number) can also be added to or subtracted from a matrix. What GAUSS does is to replicate the scalar to a matrix of the same size as the other matrix and then proceed with the usual calculation. GAUSS also allows a vector to be added to or subtracted from a matrix so long as the dimension of the vector is the same as either column number or row

number of the other matrix. What GAUSS does again is to replicate the vector to a conformable matrix. For example, suppose 'a' is a 1×4 row vector and 'b' is a 6×4 matrix. When computing 'a + b', GAUSS first replicates 'a' six times to form a 6×4 matrix with six identical rows and then adds this matrix to 'b'.

There are two new operators '.*' and './' (i.e., '*' and '/' preceded by a dot) in GAUSS that are referred to as element-by-element multiplication and element-by-element division, respectively. Given that 'a' and 'b' are two matrices of the same dimensions, 'a.*b' means that each element of 'a' is multiplied by the corresponding element in 'b' and 'a./b' means that each element of 'a' is divided by the corresponding element in 'b.'

Since the rule of element-by-element multiplication and element-by-element division about the dimension work is the same as matrix addition, it is also possible for the two matrices under the element-by-element operation to have different dimensions: GAUSS simply expands the matrix of the smaller size before operates on it.

If we are not sure about how the four basic arithmetic operators work, the best way to figure it out is to go ahead to create some simple matrices in GAUSS and then play with them a little. For example,

```
let a[1,3] = 1 2 3;
let b[2,3] = 1 1 1 2 2 2;
c = a.*b;
a;? b;?; c;
```

Viewing results of such experiments should greatly help us understand element-by-element operations.

3.3 Other Arithmetic Operators

The operator for exponentiation is '^'. For example, " 5^3 " is written in GAUSS as

```
5^3;
```

If 'a' is a 4×6 matrix, then the expression

```
b = a^2;
```

creates a 4×6 matrix 'b' whose elements are squares of the corresponding elements in the matrix 'a'.

Matrix transpose is ''. For example, the GAUSS translation of "$z = \mathbf{x}'\mathbf{x}$" is

```
z = x'x;
```

Strictly speaking, the correct expression should be 'x'*x' in the above statement. However, GAUSS allows the abbreviation of the double operators ''*' to simply ''.

Knonecker product such as "$\mathbf{c} = \mathbf{a} \otimes \mathbf{b}$" is expressed in GAUSS as

```
c = a.*.b;
```

If the dimensions of the matrices 'a' and 'b' are $m \times n$ and $p \times q$, respectively, then the resulting 'c' is a blocked matrix containing $m \times n$ blocks. The (i, j)-th block is a matrix of the dimension $p \times q$ which is the product of the (i, j)-th element in the matrix 'a' and the entire matrix 'b'. So the dimension of 'c' is $mp \times nq$.

3.4 Priority of the Arithmetic Operators

If more than one operator appear in the same expression, some operators will be performed prior to the others. For example, matrix multiplication has higher priority than matrix addition: $2 + 3 \cdot 4$ is equal to $2 + (3 \cdot 4)$ instead of $(2 + 3) \cdot 4$. Also, $[(a + b)c]^2$ is different from $a + b \cdot c^2$ because the priority of exponentiation operation is higher than both multiplication and addition. Here, we note that parentheses and brackets help to rearrange the priority of the operations. Generally speaking, the usual priority rule we learn from high school algebra still applies to the GAUSS operation and it is not necessary to memorize any new rule.

Note that the GAUSS expression for computing $[(a+b)c]^2$ is '((a + b)*c)^2'. Since the brackets are not used in GAUSS, we need two layers of parentheses here in the GAUSS expression. The best way to avoid trouble when we are not sure about the priority of some operators is to use parentheses generously.

An Example Given data on quantities q_{ic}, budget shares s_{ic}, and income m_c of 10 commodities in the ASCII files 'volume', 'share', and 'totalexp', we can compute the expenditures e_{ic} and prices p_{ic}, where

$$e_{ic} \equiv p_{ic}q_{ic} = m_c \frac{p_{ic}q_{ic}}{m_c} \equiv m_c s_{ic},$$

from which we can also compute the prices

$$p_{ic} = \frac{p_{ic}q_{ic}}{q_{ic}} \equiv \frac{e_{ic}}{q_{ic}}.$$

In the following GAUSS program we load the ASCII data files, compute the 10 expenditures and prices for 60 countries, and then print the results in an output file named 'comp.out':

```
load q[60,10] = a:\data\volume;
load s[60,10] = a:\data\share;
load m[60,1] =  a:\data\totalexp;

e = m.*s;
p = e./q;

output file = comp.out reset;
format /rd 8,2;

"The Expenditures of 10 Commodities:";?;
e;?;?;

"The Prices of 10 Commodities:";?;
```

```
15  p;?;?;
16
17  output off;
18
19  save q, p, m;
```

The output ASCII file `comp.out` from this simple program will contain two 60×10 matrices: the expenditures and the prices of the 10 commodities from 60 countries. The quantities, prices, and income are also stored as matrix files with name `q.fmt`, `p.fmt`, and `m.fmt`, respectively. Because we do not explicitly specify the subdirectory and drive names, all output files are located at the current subdirectory in drive `c` (which is the default).

Note that the 60×10 matrices '`e`' and '`p`' of expenditures and prices are computed using the element-by-element multiplication and element-by-element division, respectively. In particular, we note that '`m`' is only a 60×1 column vector while '`s`' is a 60×10 matrix. When they are multiplied, GAUSS first expands '`m`' to a 60×10 matrix with identical columns and then multiplies it, element by element, to the matrix '`s`'. These examples show how convenient the element-by-element operations are.

3.5 Matrix Concatenation and Indexing Matrices

One of the most useful features of GAUSS is it allows us to manipulate matrices almost anyway we want. We can combine several matrices by *concatenation* or extract part of a matrix by *indexing*.

If '`a`' is a 3×4 matrix and '`b`' is a 3×2 matrix, then they can be concatenated horizontally to a 3×6 matrix as follows:

```
1  c = a~b;
```

The first four columns of '`c`' come from '`a`' and the last two columns come from '`b`'. Similarly, two matrices '`d`' and '`e`' with the same column numbers can be concatenated vertically as follows:

```
1  f = d|e;
```

If we want to extract the second and fourth rows of a matrix '`a`' to form a new matrix '`b`' of two rows, then we type

```
1  b = a[2 4,.];
```

The two numbers in the brackets before the comma are row indices and the numbers, if any, after the comma are column indices. In the above case, the column indices are replaced by a dot '`.`' which means all columns are selected. If we want to extract the third and fourth columns of a matrix '`a`' to form a new matrix '`b`' of two columns, then we type

```
1  b = a[.,3 4];
```

If we want to extract the first, fourth and second rows, and the fifth column of a matrix 'a' to form a 3×1 matrix 'b', then we type

```
b = a[1 4 2,5];
```

Note that the indices can be in any order so that we can rearrange the elements of a matrix in any order we want.

Let's consider the problem of reading data to the matrices 'y' and 'X' of the equation /* 2.1 */. Suppose the data 'y' and 'X' are stored together in a 5×4 matrix format in the ASCII file 'all.dat', where the first column of the matrix contains the data for 'y' and the last three columns are for 'X'. We use the following statements to read the data into 'y' and 'X':

```
load alldata[5,4] = all.dat;

y = alldata[.,1];
X = alldata[.,2:4];
```

When defining 'X', we use '2:4' to denote the column indices '2 3 4'. The colon mark can be used to abbreviate consecutive indices.

An Example Suppose we are interested in the International Consumption data on Food and we want to list quantities, prices, and budget shares, together with income in one ASCII file food.out. Here we can use the three matrix files q.fmt, p.fmt, and y.fmt created earlier as the inputs.

```
load food_q = q, food_p = p, y;

food_q = food_q[.,1];
food_p = food_p[.,1];
food_s = (food_q.*food_p)./y;

out = food_q~food_p~food_s~y;

output file = food.out reset;
format /rd 10,2;

out;

output off;
```

Note that when we load the matrix files 'q' and 'p', we change their names to 'food_q' and 'food_p', respectively. Since Food data are in the first column of these matrices, the matrix indexing technique is applied to pick these columns. We also note that the resulting column vectors for Food data are again named as 'food_q' and 'food_q', respectively. Such reuse of the matrix names in assignment commands

is perfectly acceptable. However, we should know that after the execution of these commands, the original 60×10 matrices '`food_q`' and '`food_q`' will no longer exist (in the computer memory) because they have been completely replaced by 60×1 column vectors of the Food data. The main reason for adopting such a trick is to economize the number of matrices. Since each matrix occupies some computer memory, it is always desirable to get rid of those matrices which are no longer needed to release the computer memory for other uses.

Another way to clear unwanted matrices is to directly set them to zero. For example, immediately after the '`out`' matrix is defined we can have the following expressions:

```
food_q = 0;
food_p = 0;
food_s = 0;
y = 0;
```

Here are a few interesting questions about the above program:

- What is the size of the matrix '`out`'?
- What is the format of the print-out in the output file '`food.out`'?
- How can we modify the above program so that the output contains data for the first ten and the last ten countries only?

  The answer: first change '`food_q[.,1]`' and '`food_p[.,1]`' in the second and the third expressions to '`food_q[1:15 46:60,1]`' and '`food_p[1:15 46:60,1]`', respectively, and then change '`y`' in the fourth and fifth expressions to '`y[1:15 46:60]`'. Note that, since '`y`' is a column, its second index can be omitted inside the brackets.

- How can we modify the above program to print out the results for a combined commodity of Food and Beverages and Tobacco?

  The answer: simply change '`food_q[.,1]`' and '`food_p[.,1]`' in the second and the third expressions to '`food_q[.,1] + food_q[.,2]`' and '`food_p[.,1] + food_p[.,2]`', respectively.

- Is it possible to print the matrix '`out`' in a way that different columns have different formats?

  The answer is no. To do this we need a special GAUSS command which will be discussed later.

Chapter 4

GAUSS Commands

The strength of GAUSS lies in more than a hundred GAUSS commands whose function covers almost all basic mathematical and statistical operations. In this section we will list the most useful ones.

4.1 Special Matrices

GAUSS provides several special matrices that can be used as building blocks for matrix manipulation:

```
a = zeros(3,5);          /* A 3 x 5 matrix of zeros.                    */
b = ones(2,4);           /* A 2 x 4 matrix of ones.                     */
c = eye(4);              /* A 4 x 4 identity matrix.                    */
d = seqa(2.5,0.25,10);   /* A 10 x 1 column of additive sequence with
                            2.5 as the first element and 0.25 as the
                            increment. So 2.75 = 2.5 + 0.25 is the
                            second element, 3 = 2.75 + 0.25 the third
                            element, etc.                               */
e = seqm(5,2,20);        /* A 20 x 1 column of multiplicative sequence
                            with 5 as the first element and 2 as the
                            multiplication factor. So 10 = 5 x 2 is the
                            second element, 20 = 10 x 2 the third
                            element, etc.                               */
```

4.2 Simple Statistical Commands

GAUSS provides many powerful statistic operations:

```
a1 = sumc(x);            /* The sum.                                    */
a2 = prodc(x);           /* The product.                                */
a3 = meanc(x);           /* The mean.                                   */
a4 = median(x);          /* The median.                                 */
a5 = stdc(x);            /* The standard deviation.                     */
a6 = maxc(x);            /* The maximum.                                */
a7 = minc(x);            /* The minimum.                                */
```

Each of these commands operates on the k columns of the $b \times k$ input matrices ‘x’ and produce a $k \times 1$ column vector. For example, the ‘meanc’ command computes the mean of each of the k columns of ‘x’ and lists the resulting means as a $k \times 1$ *column* vector.

4.3 Simple Mathematical Commands

common mathematical operations are also easy to performed in GAUSS:

```
b1 = exp(x);        /* The exponential function.                          */
b2 = ln(x);         /* The logarithmic function with the natural base.    */
b3 = log(x);        /* The logarithmic function with base 10.             */
b4 = sqrt(x);       /* The square root.                                   */
b5 = abs(x);        /* The absolute value.                                */

b6 = pi;            /* The pi value 3.14159...                            */
b7 = gamma(x);      /* The gamma function.                                */

b8 = sin(x);        /* The sine function of x which is in radians.        */
b9 = cos(x);        /* The cosine function of x which is in radians.      */
b10 = tan(x);       /* The tangent  function of x which is in radians.    */
b11 = arcsin(x);    /* The inverse sine function.                         */
b12 = arccos(x);    /* The inverse cosine function.                       */
b13 = atan(x);      /* The inverse tangent function.                      */

b14 = sortc(x,i);   /* x is sorted based on the i-th column of x; i.e.,
                       the rows of x are rearranged in the ascending
                       order of the elements of the i-th column of x.     */
```

The output matrices from these commands all have the same dimensions as their input matrices.

4.4 Matrix Manipulation

Many matrix operators are easy to implement in GAUSS:

```
r = rows(x);        /* The row number of the matrix x.                    */
c = cols(x);        /* The column number of the matrix x.                 */
d = det(x);         /* The determinant of the square matrix x.            */
g = diag(x);        /* Extracting the diagonal elements of the square
                       matrix x as a column vector.                       */
k = rank(x);        /* The rank of an arbitrary matrix x.                 */
v = rev(x);         /* Reversing the order of rows of the matrix x        */
                    /* Column by column.                                  */
x1 = inv(x);        /* The inverse of the nonsingular matrix x.           */
x2 = invpd(x);      /* The inverse of the positive definite matrix x.     */
x3 = eig(x);        /* The eigenvalues of a square matrix x.              */
x4 = eigh(x);       /* The eigenvalues of a square symmetric matrix x.    */
{x5,x6} = eigv(x);  /* x5 contains the eigenvalues of a square matrix x
                       and x6is a matrix whose columns are the
```

```
                        corresponding eigenvectors of x.                */
{x7,x8} = eighv(x); /* x7 contains the eigenvalues of a square symmetric
                       matrix x and x8 is a matrix whose columns are the
                       corresponding eigenvectors of x.                 */
x9 = diagrv(x,a);   /* x9 is the same as x except the diagonal elements
                       of x9 are replaced by the vector a.              */
```

4.5 Basic Control Commands

The following important commands are not for algebraic operations but for managing the execution of the GAUSS program. They are probably the most used GAUSS commands. In particular, the '`new`' command is always placed at the beginning of the program while the '`end`' command is always at the end of the program.

```
new;  /* This command is placed at the very beginning of a GAUSS program.
         It cleans up and prepares the computer memory for a new program.*/

end;  /* This command is placed at the very end of a GAUSS program. Its
         main function is to close all the opened files in the program.
         However, it does not clear memory. So all the matrices defined
         in the program are still in the computer memory after the
         program terminates. These matrices can still be listed on the
         screen for viewing, for example.                                */

#lineson;   /* This command attaches line number to a program so that if
               errors occur, GAUSS will report the line numbers at which
               errors occur.                                            */

#linesoff;  /* This command stop keeping track of the line number to a
               program so that the execution of the program can be
               speeded up. However, if errors occur during the
               execution, the line numbers at which errors occur will
               not be reported. However, the line numbers at which
               syntax errors occur will always be reported.             */

clear x, y, z;  /* This command clears computer memory occupied by the
                   matrices x, y, z by setting the matrices x, y, z to
                   a scalar zero.                                       */
```

As mentioned earlier, whenever we are unsure about the function of a GAUSS command, we can use On-Line Help in the command mode by pressing Alt-H. For example, if we want to know more about the GAUSS command '`invpd`', we press Alt-H to get a help screen and then press H again to get the prompt '`Help On:`' at the bottom of the screen. If we type '`invpd`', then the on-line help will display information about the GAUSS command '`invpd`'.

4.6 Some Examples

It is possible to use GAUSS to verify many matrix algebra results and it is interesting to see how we can write GAUSS programs to do that.

Given an arbitrary matrix, it is usually not easy to figure out its rank directly. The GAUSS command '`rank`' is quite handy in this regard. Let's use the following GAUSS program to demonstrate this point.

```
new;

let x[5,3] =  1   2   3              /* Defining a matrix arbitrarily. */
              3   6  -1
             -1  -2   5
              2   4   7
             12  24   0;

" The row number of the matrix: " rows(x);
" The column number of the matrix: " cols(x);
" The rank of the matrix: " rank(x);

end;
```

In this very simple but complete GAUSS program, which starts with the standard '`new`' command and ends with the '`end`' command, we first create a 5×3 matrix and then print the row number, column number, and the rank of this matrix on the screen. The outputs of this program on the screen are most likely to be

```
 The row number of the matrix: 5.000000
 The column number of the matrix: 3.000000
 The rank of the matrix:  2.000000
```

Note that, since the first column and the second column of the matrix '`x`' are proportional, the rank of '`x`' is not 3 but 2; i.e., the matrix '`x`' does not have full column rank.

If we do not like seeing so many zero hanging after the decimal point and we know the results are integer numbers, we can add one more command: '`format /rd 5,0`' before the '`row`' command so that no zero will show up after the decimal point.

Another interesting result about the rank is that if the $n \times k$ matrix $\mathbf{X}$ has full column rank (so its row number n must be greater than its column number k), then the $k \times k$ square matrix $\mathbf{X}'\mathbf{X}$ has rank k and is thus a nonsingular matrix (in fact, a positive matrix), while the $n \times n$ square matrix $\mathbf{X}\mathbf{X}'$ will also have rank k and therefore is *not* a nonsingular matrix. A simple GAUSS program can help verify these results.

```
new;

let x[5,3] =  1   2   3              /* Defining a matrix arbitrarily. */
              4   5   6
              7   8   9
```

```
              10  11  12
              13  14  15;

format /rd 7,0;

" The matrix X'X is: "; x'x;?;                /* Printing a 3 x 3 matrix. */
" The rank of X'X:" rank(x'x);?;?;?;

" The matrix XX' is: "; x*x';?;               /* Printing a 5 x 5 matrix. */
" The rank of XX':" rank(x*x');

end;
```

Calculating the eigenvalues (and the corresponding eigenvectors) of a (symmetric) matrix is usually difficult. But GAUSS can do it quite easily. Let's first review a few facts about eigenvalues before we write GAUSS programs to verify them. Suppose $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of a symmetric matrix $\mathbf{A}$.

1. The determinant $|\mathbf{A}| = \prod_{i=1}^{n} \lambda_i$ and the trace $\text{tr}(\mathbf{A}) = \sum_{i=1}^{n} \lambda_i$.
2. The eigenvalues of a nonsingular matrix are all nonzero, and the eigenvalues of a positive definite matrix are all positive.
3. The eigenvalues of $\mathbf{A}^{-1}$ are the inverse of the eigenvalues of $\mathbf{A}$.

The following GAUSS program helps verify the first result:

```
new;

let x[5,5] =  1  2  3  4  5     /* Defining a square matrix arbitrarily. */
              2  3  4  5  6
              3  4  5  6  7
              4  5  6  7  8
              0  1  2  3  4;

format /rd 7,0;

" The determinant of the matrix X is " det(x);
" The product of the eigenvalues of X is " prodc(eigh(x));?;?;

" The trace of the matrix X is " sumc(diag(x));
" The sum of the eigenvalues of X is " sumc(eigh(x));

end;
```

Recall that the trace of a square matrix is the sum of its diagonal elements. Also, it is all-right to put one GAUSS command into another GAUSS command like '`sumc(diag(x))`' which produces the trace of the square matrix '`x`'.

In the following program we will create a positive definite matrix using the fact that the $k \times k$ matrix $\mathbf{X}'\mathbf{X}$ is always positive definite if the $n \times k$ matrix $\mathbf{X}$ has full column rank.

```
new;

let x[5,3] =  1  2  3          /* Defining a matrix of full column rank. */
              4  5  6
              7  8  9
             10 11 12
             13 14 15;

format /rd 10,6;

" The eigenvalues of the positive definite matrix X'X and its inverse, ";
" as well as the reciprocals of the latter:";

eigh(x'x)~eigh(invpd(x'x))~(1./eigh(invpd(x'x)));

end;
```

Since we use horizontal concatenation '~' to put together the three columns of results, a 3×3 matrix will be printed and it should confirm that all the eigenvalues of 'x'x' are positive and that the eigenvalues of 'x'x' and 'invpd(x'x)' are reciprocal.

Note that the last column is the result of element-by-element division '1./eigh(invpd(x'x))'. The reason for an additional pair of parentheses to encircle this expression is to prevent the possibility that the concatenation '~' may have higher priority in execution than the division, in which case the result would be completely messed up. We should use parentheses generously to avoid any potential confusion of this kind.

Also note that we use the 'invpd' command, instead of the 'inv' command, to invert the matrix 'x'x' because we know 'x'x' is positive definite. There are two advantages of using the 'invpd' command to invert a positive definite matrices: First, the 'invpd' command can do the job more efficiently than 'inv', which is applicable to any nonsingular matrix. Secondly, if for whatever reason (e.g., 'x' does not have full column rank) 'x'x' is not positive definite, GAUSS will not execute the 'invpd(x'x)' command and complain about it. This is good for detecting any potential problem of the program.

We now consider two results on the partitioned matrices. The first one involves an important formula for inversion:

$$\begin{bmatrix} \mathbf{A}_1 & \mathbf{B} \\ \mathbf{C} & \mathbf{A}_2 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{X}_1 & -\mathbf{A}_1^{-1}\mathbf{B}\mathbf{X}_2 \\ -\mathbf{A}_2^{-1}\mathbf{C}\mathbf{X}_1 & \mathbf{X}_2 \end{bmatrix},$$

where

$$\mathbf{X}_1 = (\mathbf{A}_1 - \mathbf{B}\mathbf{A}_2^{-1}\mathbf{C})^{-1} \qquad \text{and} \qquad \mathbf{X}_2 = (\mathbf{A}_2 - \mathbf{C}\mathbf{A}_1^{-1}\mathbf{B})^{-1},$$

and $\mathbf{A}_1$ and $\mathbf{X}_1$ are two square matrices of the same dimensions; and $\mathbf{A}_2$ and $\mathbf{X}_2$ are two square matrices of the same dimensions.

To write a GAUSS program to verify this result, we first define an arbitrary nonsingular 5×5 matrix 'big_mac', which consists of four blocks 'a1', 'b', 'c', and'a2'.

```
new;
                              /* Defining the four blocks arbitrarily.     */
let a1[3,3] = 1  2  3  1  3  5  1  -2  4;
let b[3,2] = -1  2 -3  1  0  2;
let c[2,3] =  4  1  0 -1 -2  0;
let a2[2,2] = 3 -1  2 -1;

big_mac = (a1~b)|(c~a2);      /* Combining the four blocks into one big
                                 matrix.                                   */
big_mac = inv(big_mac);

x1 = inv(a1 - b*inv(a2)*c); /* The individual inverse formulas.            */
x2 = inv(a2 - c*inv(a1)*b);
y = -inv(a1)*b*x2;
z = -inv(a2)*c*x1;

big_inv = (x1~y)|(z~x2);      /* Combining the four inverses into one big
                                 matrix.                                   */
format /rd 10,6;

big_mac;?;?;
big_inv;

end;
```

The definition of the blocks ‘a1’, ‘b’, ‘c’, and‘a2’ are arbitrary. Their sizes and contents can be changed as long as the resulting matrix ‘mig-mac’ is nonsingular. The outputs of this program should be two identical 5×5 matrices.

If in this example we create a matrix which is too big to be shown in one screen, then it will be very hard to visually compare the two resulting matrices. A better way to compare big matrices is as follows:

```
format /rd 15,12;

out = maxc(maxc(abs(big_mac - big_inv)));
out;
```

(Question: Why are there two ‘maxc’ commands?) The result ‘out’ is expected to be zero but may not be exactly equal to 0.000000000000: there may be some nonzero digits appearing at the end of the expression. Such small discrepancy between two supposedly equal matrices illustrates the nature of the computer in that all computations are conducted with certain degree of rounding errors. It should however be pointed out that GAUSS is quite good in keeping a high precision level. It can achieve around 16 digits of accuracy which is sufficient for producing *correct* answers to almost all econometric applications.

Note that the above formula for the inverse of the partitioned matrix involves the inverses of the two diagonal blocks $\mathbf{A}_1$ and $\mathbf{A}_2$. It is sometimes possible that one of them may not be invertible, say, $\mathbf{A}_2$, in which case we should replace the two sub-formulas $\mathbf{X}_1 = (\mathbf{A}_1 - \mathbf{B}\mathbf{A}_2^{-1}\mathbf{C})^{-1}$ and $-\mathbf{A}_2^{-1}\mathbf{C}\mathbf{X}_1$ by the equivalent $\mathbf{A}_1^{-1} + \mathbf{A}_1^{-1}\mathbf{B}\mathbf{X}_2\mathbf{C}\mathbf{A}_1^{-1}$ and $-\mathbf{X}_2\mathbf{C}\mathbf{A}_1^{-1}$, respectively.

Let's now consider two formulas for the determinant of the partitioned matrix:

$$\begin{vmatrix} \mathbf{A}_1 & \mathbf{B} \\ \mathbf{C} & \mathbf{A}_2 \end{vmatrix} = |\mathbf{A}_2| \cdot |\mathbf{A}_1 - \mathbf{B}\mathbf{A}_2^{-1}\mathbf{C}| = |\mathbf{A}_1| \cdot |\mathbf{A}_2 - \mathbf{C}\mathbf{A}_1^{-1}\mathbf{B}|.$$

The corresponding GAUSS program for checking the first equality is

```
new;

let a1[3,3] = 1  2  3  1  3  5  1  -2  4;
let b[3,2] = -1  2 -3  1  0  2;
let c[2,3] =  4  1  0 -1 -2  0;
let a2[2,2] = 3 -1  2 -1;

one_det = det((a1~b)|(c~a2));
two_det = det(a2)*det(a1 - b*inv(a2)*c);
out = abs(one_det - two_det);

format /rd 15,12;
out;

end;
```

The above formulas for the partitioned matrices are very useful when we need the inverse or the determinant of a big matrix while the computer memory is not sufficient to handle it. These examples also demonstrate an important trick in dealing with the problem of insufficient memory: we can and should break the trouble-making matrix into smaller pieces and handle them piece by piece.

Let's now use the International Consumption Data to construct additional examples. Given quantities q_{ic} and budget shares s_{ic} for the commodity i in country c, and the country c's income m_c for the 60 countries, we can reorder q_{ic} and s_{ic} according to their income, either in ascending or descending order:

```
new;

load q[60,10] = a:\data\volume;             /*  The quantities.          */
load s[60,10] = a:\data\share;              /*  The budget shares.       */
load m[60,1] = a:\data\totalexp;            /*  The total expenditure.   */

q = sortc(m~q,1);
s = sortc(m~s,1);

```

```
output file = order.out on;
format /rd 10,3;;

" The Ordered Income and Quantities (in Ascending Order):";
q;?;
" The Ordered Income and Budget Shares (in Ascending Order):";
s;?;?;?;

" The Ordered Income and Quantities (in Descending Order):";
rev(q);?;
" The Ordered Income and Budget Shares (in Descending Order):";
rev(s);

end;
```

Four 60×11 matrices will be printed into the ASCII file '`order.out`'. The first two matrices are ordered in the ascending order of the first column, which is the column of income. The last two matrices are in the descending order of the income. It is interesting to see that the Food budget share increases as income decreases.

We can compute many summary statistics for the International Consumption Data. Specifically, we can calculate the averages, sample medians, standard deviations, maxima, minima, and with a little more algebra, the sample covariances and correlation coefficients.

Let's concentrate on the sample covariances and correlation coefficients between m_c and each s_{ic}, $i = 1, \ldots, 10$, which are

$$\widehat{\text{Cov}}(s_i, m) = \frac{1}{60}\sum_{c=1}^{60}(s_{ic} - \bar{s}_i)(m_c - \bar{m}) \qquad \text{and} \qquad \widehat{\text{Corr}}(s_i, m) = \frac{\widehat{\text{Cov}}(s_i, m)}{\sqrt{\widehat{\text{Var}}(s_i)} \cdot \sqrt{\widehat{\text{Var}}(m)}},$$

respectively, where $\bar{s}_i$ and $\bar{m}$ are the sample averages and $\widehat{\text{Var}}(s_i)$ and $\widehat{\text{Var}}(m)$ are the sample variances:

$$\widehat{\text{Var}}(s_i) = \frac{1}{60}\sum_{c=1}^{60}(s_{ic} - \bar{s}_i)^2 \qquad \text{and} \qquad \widehat{\text{Var}}(m) = \frac{1}{60}\sum_{c=1}^{60}(m_c - \bar{m})^2.$$

Suppose $\mathbf{S}$ is the 60×10 matrix of data on s_{ic}, and $\mathbf{m}$ is the 60×1 vector of data on m_c. We can construct a 60×10 matrix $\bar{\mathbf{S}}$ in which each column contains 60 identical numbers which are the sample averages of the budget shares. Thus, the (i, c)-th element of the difference matrix $\mathbf{D} = \mathbf{S} - \bar{\mathbf{S}}$ is precisely $s_{ic} - \bar{s}_i$. We can similarly define a vector $\mathbf{a}$ whose typical element is the difference $m_c - \bar{m}$. Given these definitions, we then have

$$\mathbf{D}'\mathbf{a} = [\,\mathbf{d}_1 \;\; \mathbf{d}_2 \;\; \cdots \;\; \mathbf{d}_{10}\,]'\mathbf{a} = \begin{bmatrix} \mathbf{d}_1' \\ \mathbf{d}_2' \\ \vdots \\ \mathbf{d}_{10}' \end{bmatrix}\mathbf{a} = \begin{bmatrix} \mathbf{d}_1'\mathbf{a} \\ \mathbf{d}_2'\mathbf{a} \\ \vdots \\ \mathbf{d}_{10}'\mathbf{a} \end{bmatrix},$$

which is a 10×1 vector with the ith element being the sample covariance between the ith budget share s_i and income m:

$$\mathbf{d}_i'\mathbf{a} = \sum_{c=1}^{60} d_{ic}a_c = \sum_{c=1}^{60}(s_{ic} - \overline{s}_i)(m_c - \overline{m}), \qquad i = 1, \ldots, 10.$$

This formula can be adopted for efficiently computing the sample covariances in GAUSS. Given that 's' denotes **S** and 'm' denotes **m** in GAUSS, if we define

```
cov_sm = (s - meanc(s)')'(m - meanc(m))./60;
```

then 'cov_sm' is a 10×1 vector of sample covariances between s_{ic} and m_c, for $i = 1, \ldots, 10$. Here, we should note that 'meanc(s)' gives a 10×1 vector of means and its dimension is not the same as 's'. But 's - meanc(s)'' will correctly produce the matrix $\mathbf{D} = \mathbf{S} - \overline{\mathbf{S}}$ because the particular way GAUSS handles subtraction of matrices of unequal sizes. This is a useful trick and it can be used in many occasions.

We can similarly compute the two sample variances of s_{ic} and m_c as follows:

```
varcov_s = (s - meanc(s)')'(s - meanc(s)')./60;
var_s = diag(varcov_s);
var_m = (m - meanc(m)')'(m - meanc(m)')./60;
```

Note that 'varcov_s' is a 10×10 sample variance-covariance matrix of s_{ic}, $c = 1, 2, \ldots, 10$, and its diagonal contains 10 sample variances. Also note that GAUSS provides us with a command 'stdc' which computes the standard deviation (the square root of the sample variance). That is, the vectors 'var_s' should be equal to 'stdc(s)^2', and 'var_m' should be equal to 'stdc(m)^2'. Therefore, the correlation coefficients can be computed by either

```
corr_sm = cov_sm./sqrt(var_s.*var_m);
```

where 'var_s' and 'var_m' are computed as above, or, equivalently,

```
corr_sm = cov_sm./(stdc(s).*stdc(m));
```

Note that the sizes of the matrices 'cov_sm', 'var_s', 'var_m', and 'corr_sm' are all 10.

We now combine all these expressions in one GAUSS program to generate the basic summary statistics for the International Consumption Data. Here, instead of looking at total expenditure m_c, we consider the $\ln m_c$, the logarithmic transformation of m_c. The GAUSS command for the natural log transformation is 'ln'.

```
new;

load q[60,10] = a:\data\volume;
load s[60,10] = a:\data\share;
load m[60,1] = a:\data\totalexp;
```

```
6
7   m = ln(m);                                  /* The log total expenditure. */
8
9   p = (m.*s)./q;                              /* The prices.                */
10
11  mean_s = meanc(s);                          /* The averages.              */
12  mean_m = meanc(m);
13  mean_p = meanc(p);
14
15  std_s = stdc(s);                            /* The standard deviations.   */
16  std_m = stdc(m);
17  std_p = stdc(p);
18
19  /* The sample covariances between shares and log total expenditure.      */
20  cov_sm = (s - mean_s')'(m - mean_m)./60;
21
22  /* The sample correlations between shares and log total expenditure.     */
23  corr_sm = cov_sm./(std_s.*std_m);
24
25  /* The sample covariances between prices and log total expenditure.      */
26  cov_pm = (p - mean_p')'(m - mean_m)./60;
27
28  /* The sample correlations between prices and log total expenditure.     */
29  corr_pm = cov_pm./(std_p.*std_m);
30
31  /* Creating a vector of consecutive numbers from 1 to 10.                */
32  no = seqa(1,1,10);
33
34  out_s = no~mean_s~std_s~maxc(s)~minc(s)~median(s)~cov_sm~corr_sm;
35  out_p = no~mean_p~std_p~maxc(p)~minc(p)~median(p)~cov_pm~corr_pm;
36  out_m = mean_m~std_m~maxc(m)~minc(m)~median(m);
37
38  output file = summary on;
39  format /rd 7,3;
40
41  " The sample averages, standard deviations, maxima, minima, medians, "
42  " of shares; and the sample covariances and the sample correlations "
43  " between shares and log total expenditure:";
44  out_s;?;?;
45
46  " The sample averages, standard deviations, maxima, minima, medians, "
47  " of prices; and the sample covariances and the sample correlations "
48  " between prices and log total expenditure:";
49  out_p;?;?;
50
```

```
51    " The sample averages, standard deviations, maxima, minima, and medians "
52    " of the log total expenditure:";
53    out_m;
54
55    end;
```

The 'no' vector is constructed as a counting device to facilitate the reading of the outputs.

Finally, it should be pointed out that there are GAUSS commands 'vcx(s)' and 'corrx(s)' which compute the 10×10 variance-covariance matrix and the correlation matrix, respectively, from the 60×10 data matrix 's'. The diagonal terms of 'vcx(s)' are sample variances while the off-diagonal elements are sample covariances. Also, the diagonal terms of 'corrx(s)' are all ones. The two matrices 'cov_sm' and 'corr_sm' in the previous program will then be equal to the last columns (except the last elements) of the resulting matrices from 'vcx(s~m)' and 'corrx(s~m)', respectively.

4.7 Character Matrices and Strings

Other than numeric data, GAUSS also allows character data which may be presented in two forms: character matrices and strings.

4.7.1 Character Matrices

The most common form of the character matrix is the character vector, which are used mainly for storing a vector of names for the purpose of identifying rows or columns of a numeric data matrix. For example, an $1 \times k$ character row vector can be created and then concatenated to an $n \times k$ matrix of numbers so that each column of n numbers can be associated with a character name. Consider a more specific example: Suppose the 60×10 matrix 's' contains 60 observations on 10 budget shares. We can store the names of the 10 commodity into a 1×10 character vector, say, 'varname', and concatenate this row vertically to the numeric matrix 's' as follows:

```
1    let varname[1,10] = Food  B_T  Cloth  Rent  House
2                        Medic  Trans  Recre  Ed  Other;
3
4    s1 = varname|s;
```

Here, 's1' is a 61×10 mixed matrix in which the first row are characters and the rest are numbers.

From this simple example we know the way characters are stored in a vector is quite similar to that of a numeric matrix. However, there are a few more rules for handling character vectors: each element of a character vector can contain up to eight characters only. The contents of the above character vector 'varname' will all be uppercase, such as 'FOOD' instead of 'Food'. If the cases of the characters are to be kept exactly as what they are typed, such as 'Food', then they need to be enclosed by quotation marks in the 'let' command as follows:

```
1    let varname[1,10] = "Food"  "B_T"  "Cloth"  "Rent"  "House"
2                        "Medic"  "Trans"  "Recre"  "Ed"  "Other";
```

To print a character vector, we need to add '$' before the matrix name and this is quite different from printing a numeric matrix. For example, to print the character vector 'varname', we type

```
$varname;
```

However, to print the mixed matrix 's1' which contains both numbers and characters, we need a special GAUSS command which will be discussed shortly.

4.7.2 Strings

Other than character vectors, a long string of characters, whose number can be greater than eight (but fewer than 256), may be stored as a single item. For example, we can store the string 'GAUSS is fun' literally in a variable called 'outt' as follows:

```
outt = "GAUSS is fun";
```

where the content of a string should always be enclosed in quotation marks. To print the content of this string variable, we simply type

```
outt;
```

Note that no '$' is needed before the string variable name 'outt'. The result of the above command is a line of string 'GAUSS is fun' on the screen.

Strings can be literally joined using the operator '$+'. For example, with the statements

```
out1 = "Hello, ";
out2 = "how are you?";
out = out1 $+ out2;
```

the content of the scalar variable 'out' is then 'Hello, how are you?'.

A string can also be literally joined (element-by-element) to each element of a character vector. For example, given the previous example of the character vector 'varname', if we type

```
varname1 = varname $+ "z";
```

then the 10 elements of the new character vector 'varname1' will all be attached with 'z' and become 'Foodz', 'B_Tz', 'Clothz', etc. It is important to note that the number of characters in each element still needs to be kept fewer than or equal to eight as a general rule for character vectors.

Two character vectors of the same size can also be joined together by '$+'. If we type

```
let a[3,1] = "GAUSS" "is" "fun";
let b[3,1] = "So" "is" "Econ";

c = a $+ b;
```

then 'c' is also a 3×1 vector with the three elements: 'GAUSSSo, isis, and funEcon.

4.7.3 The Data Type

It is important to understand that strings, as opposed to other data formats such as numeric matrices or character vectors, are a very unique *type* of data. As a matter of fact, all data in GAUSS can be broadly classified as of either *string type* or *matrix type*, while both numeric matrices and character vectors belong to the matrix type. The difference between the string type and the matrix type can be best illustrated in the following problematic statement which invokes the error message of 'Type Mismatch':

```
varname2 = "z" $+ varname;
```

where 'varname' is a 10×1 character vector as defined above. The problem results from a rule in GAUSS that the *type* of the right-hand side operation is decided by the *type* of the first item. In the above example, the first item 'z' on the right-hand side is a string, so the joint operation is considered to be an operation among strings and every item at the right-hand side is expected to be a string. But since the second item is actually a matrix, the syntax error of mismatched type will show up. To fix this problem, we can adopt the following trick:

```
varname2 = 0 $+ "z" $+ varname;
```

Here, the first item '0' on the right-hand side is a number, which is always of the matrix type. So the entire operation on the right-hand side is considered a matrix operation and 'z' is treated as a matrix with a single element instead of a string (the usual element-by-element rule is then applied to the '$+' operations). After the characters are joined, the elements of the character vector 'varname2' become 'zFood', 'zB_T', 'zCloth', etc. Note that the value 0 itself will not be a part of the characters.

4.7.4 Three Useful GAUSS Commands

A couple of problems may arise when we try to combine numeric matrices with character vectors:

1. In many occasions we may want to combine a character vector and a numeric vector to create another character vector of which each entry mixes characters with numbers. But we note characters and numbers are of different nature and we are not sure whether the character joint operator '$+' can be applied to combine characters and numbers.

2. When we print a character vector, we have to precede its variable name with a dollar sign '$' in the print command. It is then unclear how to print a matrix that contains both numbers and characters.

To solve the first problem of jointing numbers with characters or with strings, we use the commands ‘ftocv’ or ‘ftos’ to convert numbers to characters or to strings, respectively. As to the problem of printing a mixed matrix containing both numbers and characters, we need the ‘printfm’ command. These three commands will now be discussed in details.

1. The ‘ftocv’ command converts a numeric vector (or matrix) to a character vector. Given a vector ‘a’ of real numbers, if we want to convert it to a character vector with each element containing 7 characters in which 2 digits after the decimal point as well as the decimal point itself will all appear as characters, we type

```
b = ftocv(a,7,2);
```

   The ‘b’ vector now is a character vector, *even though its contents look like numbers*, and can be joined with any other character vector. If a number is not large enough to fill the space of 7 characters, character 0 will be padded on the left. Also, if we do not want the decimal point and the digits after it to be converted, we replace 2 in the third input of the ‘ftocv’ command by 0.

   Consider another example

```
name = 0 $+ "Var" $+ ftocv(seqa(0,1,21),1,0);
```

   Here, a sequence of 21 consecutive integers starting from 1 are first generated by the ‘seqa(0,1,21)’ command and then converted to a 21×1 character vector by the ‘ftocv’ command. This character vector is then joined to the string ‘Var’. The resulting ‘name’ is a 21×1 character vector containing the characters ‘Var1’, ‘Var2’, . . . Note that the trick of ‘0 $+’ at the right-hand side is used to prevent the ‘Type Mismatch’ problem (see the previous subsection).

2. The ‘ftos’ command converts a scalar to a string. Given a real numbers ‘a’, if we want to convert it to a string of 15 characters in which 4 digits after the decimal point will be included, we type

```
b = ftos(a,"%*.*lf",15,4);
```

   In comparison with the ‘ftocv’ command, we note the ‘ftos’ command has four inputs. The definitions of the first, third, and the fourth inputs are the same as the three inputs of the ‘ftocv’ command, respectively. The second input should be a string enclosed in quotation marks. The specific string ‘%*.*lf’ here will direct GAUSS to right-justify the characters. If ‘a’ is not large enough to fill the space of 15 characters, space (instead of character 0) will be padded on the left. Another common specification is ‘%-*.*lf’, which left-justifies the characters. Moreover, there is a simple way to add other characters literally before and after the character representation of the number: we put these characters inside the quotation marks before and/or after ‘%*.*lf’. For example, if the value of ‘a’ is 1234.567890123 and we type

```
b = ftos(a,"This Trial Produces %*.*lf, Which Are Characters.",12,4);
```

then ‘b’ will be a string with the content:

```
This Trial Produces    1234.5678, Which Are Characters.
```

3. The ‘printfm’ command provides us with the full control on how to print data in a matrix where some columns are numbers and some are characters. In particular, different columns can be printed in different formats. Suppose we want to print an 10×5 matrix ‘x’ in which the first column contains characters and the rest all numbers, then we type:

```
indx = zeros(10,1)~ones(10,4);

let c[5,3] =   "%*.*s"  10  10
              "%*.*lf"  12   4
              "%*.*lf"   8   2
             "%-*.*lf"  10   3
             "%-*.*lf"   6   0;

ok = printfm(x,indx,c);
```

Here, the matrix ‘indx’ has the same dimension as ‘x’ and contains 1 and 0 only. When an element of ‘x’ is a number, then the corresponding element in the ‘indx’ matrix must be the number 1, but if the element of ‘x’ is a character, then the corresponding element in the ‘indx’ matrix must be the number 0. It is possible to make ‘indx’ a 1×5 row vector containing the five numbers 0, 1, 1, 1, and 1. In such a case, the usual element-by-element operation will be applied.

As to the matrix ‘c’, its *row* number must be the same as the *column* number of the data matrix ‘x’. This is because the i-th row in ‘c’ specifies the format for the i-th column of ‘x’. The first element in each row of the ‘c’ matrix is a string. Its content usually is one of ‘%*.*lf’, ‘%-*.*lf’, ‘%*.*s’, ‘%-*.*s’, where the first two are exactly the same as the second input of the ‘ftos’ command and their interpretations are also the same. Note that ‘%*.*lf’ can be abbreviated as ‘lf’. As to the last two commands that contain the character ‘s’, they are for character printing.

The second and the third elements in each row of the ‘c’ matrix specify the number of spaces reserved for printing and the number of digits after the decimal point, respectively. They work like what the second and the third inputs of the ‘ftocv’ command.

It is possible for some column of the ‘x’ matrix to have both numbers and characters at the same time. In such cases the corresponding rows in the ‘c’ matrix should still be either ‘%*.*lf’ or ‘%-*.*lf’. When characters are encountered in printing the column, GAUSS will print them appropriately (as long as the 0-1 specifications in the second input matrix ‘indx’ are correct).

The output of the ‘printfm’ command ‘ok’ is either 1 or 0, depending on whether printing is successful or not. If this information is not needed, we can type

```
call printfm(x,indx,c);
```

Note that the command '`?;`' should be included after the above command. Otherwise, the next thing to be printed will immediately follow the last line of '`x`', instead of starting from a new line.

Chapter 5

GAUSS Program for Linear Regression

In this chapter we will write a GAUSS program for the Ordinary Least Squares (OLS) estimation of a linear regression model after a brief review of the linear regression theory. A full discussion of such a theory can be found in any intermediate econometrics textbook.

5.1 A Brief Review

Given n observations on the dependent variable y_i and k explanatory variables (regressors) $x_{1i}, \ldots, x_{ki}$, we consider the following linear regression model

$$y_i = \beta_1 x_{1i} + \beta_2 x_{2i} + \cdots + \beta_k x_{ki} + \varepsilon_i, \qquad i = 1, , \ldots, n,$$

which can be expressed compactly in matrix form:

$$\underset{n\times 1}{\mathbf{y}} = \underset{n\times k}{\mathbf{X}}\,\underset{k\times 1}{\boldsymbol{\beta}} + \underset{n\times 1}{\boldsymbol{\varepsilon}} \qquad \text{or} \qquad \begin{bmatrix} y_1 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_{11} & x_{21} & \cdots & x_{k1} \\ x_{12} & x_{22} & \cdots & x_{k2} \\ \vdots & \vdots & & \vdots \\ x_{1n} & x_{2n} & \cdots & x_{kn} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_k \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix},$$

where $\beta_1, \beta_2, \ldots, \beta_k$ are k regression coefficients and ε_i are the random disturbance terms. Four assumptions are usually made about the disturbance terms (1) $\mathrm{E}(\boldsymbol{\varepsilon}) = \mathbf{0}$; (2) $\mathrm{Var}(\boldsymbol{\varepsilon}) = \sigma^2\mathbf{I}_n$, where $\mathbf{I}_n$ is an $n \times n$ identity matrix; (3) ε_i is normally distributed; and (4) $\mathbf{X}$ is nonstochastic and is of full column rank, i.e., $\mathrm{rank}(\mathbf{X}) = k$. Note that the second assumption implies the disturbance terms are homoscedastic and unautocorrelated. It is also common to assume that one of the explanatory variables is a constant term. So one column of the matrix $\mathbf{X}$ (usually the first column) is a column of ones.

5.1.1 The Ordinary Least Squares Estimation

The OLS estimator of the parameter $\boldsymbol{\beta}$ is defined as

$$\mathbf{b} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}, \tag{5.1}$$

where $\mathbf{X}'\mathbf{X}$ is invertible (and positive definite) because $\mathbf{X}$ is assumed to be of full column rank. It can also be shown that the OLS estimator $\mathbf{b}$ is the BLUE (best linear unbiased estimator) and $\mathbf{b} \sim \mathcal{N}(\boldsymbol{\beta}, \sigma^2(\mathbf{X}'\mathbf{X})^{-1})$. The result that $\mathbf{b}$ is normally distributed is due to the normality assumption on the disturbance terms.

Given the residuals

$$\mathbf{e} = \mathbf{y} - \mathbf{X}\mathbf{b}, \tag{5.2}$$

the unbiased estimator of the disturbance variance σ^2 is

$$s^2 = \frac{1}{n-k}\mathbf{e}'\mathbf{e}. \tag{5.3}$$

The variance-covariance matrix of the OLS estimator $\text{Var}(\mathbf{b}) = \sigma^2(\mathbf{X}'\mathbf{X})^{-1}$ can be estimated by

$$\widehat{\text{Var}}(\mathbf{b}) = s^2(\mathbf{X}'\mathbf{X})^{-1}. \tag{5.4}$$

The square root of the j-th diagonal element of (5.4)

$$\text{s.e.}(b_j) = \sqrt{s^2(\mathbf{X}'\mathbf{X})^{-1}_{jj}}, \qquad \text{for } j = 1, \ldots, k., \tag{5.5}$$

is the standard errors of the j-th OLS estimator b_j and the corresponding t-ratio is

$$t_j = \frac{b_j}{\text{s.e.}(b_j)}. \tag{5.6}$$

If the true value of β_j is zero, then t_j has a t-distribution with $n-k$ degrees of freedom. This result can be used to test the hypothesis

$$H_0\colon\ \beta_j = 0 \qquad \text{against} \qquad H_1\colon\ \beta_j \neq 0,$$

and to construct the confidence interval for b_j at α level which is

$$b_j \pm t_{\alpha/2}(n-k) \times \text{s.e.}(b_j), \tag{5.7}$$

where $t_{\alpha/2}(n-k)$ is the $\alpha/2$ level critical value from the t-distribution with $n-k$ degrees of freedom.

5.1.2 Analysis of Variance

The total sample variations in the dependent variable is usually measured by the Total Sum of Squares (TSS):

$$\text{TSS} = \sum_{i=1}^{n}(y_i - \overline{y})^2 = (\mathbf{y} - \overline{y}\cdot\mathbf{1}_n)'(\mathbf{y} - \overline{y}\cdot\mathbf{1}_n). \tag{5.8}$$

where $\mathbf{1}_n$ is an n-dimensional column of ones. We note that TSS is simply the sample variance of y_i multiplied by the sample size. Based on the linear regression model we can decompose the TSS into two parts: the part of variations that can be *predicted* or *explained* by a linear combination of explanatory variables is called the Explained Sum of Squares (ESS), while the other part is called the Residual Sum of Squares (RSS). The RSS is generally easier to compute because it equals

$$\text{RSS} = \sum_{i=1}^{n} e_i^2 = \mathbf{e}'\mathbf{e}, \tag{5.9}$$

which measures the variations in the residuals. ESS can be calculated by subtraction:

$$\text{ESS} = \text{TSS} - \text{RSS}. \tag{5.10}$$

A little more algebra can yield some other expressions for ESS. Here, let's concentrate on its interpretation.

The linear regression model is based on the idea that the variations in the dependent variable y_i depends linearly on those of explanatory variables x_{1i}, x_{1i}, ..., x_{ki}. Hence, the performance of a linear regression model can be measured by the proportion of the variations in the dependent variables that can be *explained* by the variations in the linear combination of explanatory variables. Such a proportion is called the coefficient of determination or simply R^2:

$$R^2 = \frac{ESS}{TSS} = 1 - \frac{RSS}{TSS}. \tag{5.11}$$

It is the most common measure of the goodness-of-fit of a linear regression model. Its value lies between 0 and 1 and the larger its value the better. However, there is a problem with the R^2 measure: its value can be increased superficially by including even the most irrelevant explanatory variable into the model. To avoid this problem, we can modify R^2 by making the number of explanatory variables as a counterweight. More precisely, we have the following definition of the adjusted R^2:

$$\overline{R}^2 = 1 - \frac{RSS/(n-k)}{TSS/(n-1)} = 1 - \frac{n-1}{n-k}(1 - R^2), \tag{5.12}$$

whose value can decrease when we add into the model an explanatory variable that has very little contribution in explaining the dependent variable.

Based on ESS and RSS, let's define the following two ratios which are called Explained Mean Square (EMS) and Residual Mean Square (RMS):

$$EMS = \frac{1}{k-1}ESS \qquad \text{and} \qquad RMS = \frac{1}{n-k}RSS.$$

Note that RMS is the same as s^2 defined in (5.3).

Given the normality assumption (the third assumption) on the disturbance terms, it can be shown that the RSS/σ^2 has a χ^2 distribution with $n-k$ degrees of freedom. Furthermore, when the first explanatory variable is a constant term, then the ESS/σ^2 has a χ^2 distribution with $k-1$ degrees of freedom, and is independent of the RSS/σ^2 if the coefficients $\beta_2, \ldots, \beta_k$ are all zero. As a result, the ratio

$$F = \frac{EMS}{RMS}, \tag{5.13}$$

has an F-distribution with $k-1$ and $n-k$ degrees of freedom and can be used to test the hypothesis

$$H_0\colon\ \beta_2 = 0,\ \cdots,\ \beta_k = 0 \qquad \text{against} \qquad H_1\colon\ \beta_2 \neq 0,\ \cdots,\ \beta_k \neq 0.$$

We can present all these results in a table:

5.1.3 Durbin-Watson Test Statistic

As mentioned earlier, the third assumption underlying the linear regression model implies the absence of autocorrelation among the disturbance terms. But for time-series data this assumption usually fails to hold and it is quite common that the disturbance terms follow a first-order autoregressive, or the AR(1), process, which is defined by

$$\varepsilon_i = \rho\, \varepsilon_{i-1} + u_i,$$

Table 5.1: Analysis of Variance

Source	Sum of Squares	d.f.	Mean Square
Explained	$(\mathbf{y}-\overline{y}\cdot\mathbf{1}_n)'(\mathbf{y}-\overline{y}\cdot\mathbf{1}_n)-\mathbf{e}'\mathbf{e}$	$k-1$	EMS
Residual	$\mathbf{e}'\mathbf{e}$	$n-k$	RMS
Total	$(\mathbf{y}-\overline{y}\cdot\mathbf{1}_n)'(y-\overline{y}\cdot\mathbf{1}_n)$	$n-1$	

where $u_i, i = 1,\ldots,n$, are assumed to be uncorrelated and have zero mean. Note that when $\rho = 0$, then $\varepsilon_i = u_i$, which implies the absence of autocorrelation. So when we deal with time-series data, we need to test the hypothesis:

$$H_0\colon\ \rho = 0 \qquad \text{against} \qquad H_1\colon\ \rho \neq 0.$$

The standard test for such a hypothesis is based on the Durbin-Watson statistic

$$d \equiv \frac{\sum_{i=2}^{n}(e_i - e_{i-1})^2}{\sum_{i=1}^{n} e_i^2},$$

where e_i are residuals calculated from (5.2). Given $\mathbf{e}$ is the $n\times 1$ residual vector, suppose $\mathbf{e}_{(1)}$ and $\mathbf{e}_{(n)}$ are two $(n-1)\times 1$ subvectors of $\mathbf{e}$ with its first and its last element, respectively, deleted, then d can be computed by the formula

$$d = \frac{[\mathbf{e}_{(1)} - \mathbf{e}_{(n)}]'[\mathbf{e}_{(1)} - \mathbf{e}_{(n)}]}{\mathbf{e}'\mathbf{e}}. \tag{5.14}$$

5.2 The Program

The International Consumption Data can be used to fit *the Engel curve model* in which consumption is regarded as a function of the income. One useful specification for the Engel curve model is the so-called Working-Leser model where the budget share of a commodity is a linear function of the log income:

$$s_{ic} = \alpha + \beta\cdot\ln m_c + \varepsilon_{ic}, \qquad c = 1,\ldots,60, \quad \text{and} \quad i = 1,\ldots,10, \tag{5.15}$$

where ε_{ic} is the disturbance term. Although the dependent variable is not the quantity consumed but the budget share while the independent variable is not income but log income (here, total expenditure and income are considered synonymous), the equation is an well-defined Engel curve model that is linear (in the parameters).

To write a GAUSS program for the estimation of the Working-Leser Engel curve model, we should first match its notation to that of the standard linear regression model as follows:

$$\mathbf{y} = \begin{bmatrix} s_{i1} \\ s_{i2} \\ \vdots \\ s_{i,60} \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} 1 & \ln m_1 \\ 1 & \ln m_2 \\ \vdots & \vdots \\ 1 & \ln m_{60} \end{bmatrix}, \qquad \text{and} \qquad \boldsymbol{\beta} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}.$$

In the following GAUSS program the Food consumption data are used to estimate the Working-Leser Engel curve model. The first section of the program illustrates how the Food consumption data can be loaded into the appropriate matrices for computation. This GAUSS program produces standard output for the linear regression estimation. Generally, we need to change the first four statements only and the rest can be left as they are.

```
new;

/*****************************************************************
 *                 Reading and Organizing the Data Set           *
 *****************************************************************/

load share[60,10] = a:\data\share;
load totalexp[60,1] = a:\data\totalexp;

y = share[.,1];              /* Forming the vector of the dependent
                                variable which contains the budget share
                                for Food.                          */

x = ones(60,1)~ln(totalexp); /* Forming the matrix for the two
                                explanatory variables: the constant term
                                and log income.                    */

clear share, totalexp;       /* Clearing the matrices that are no longer
                                needed.                            */

output file = ols.out reset; /* Defining the output file name as ols.out.*/

/*****************************************************************
 *                     Regression Estimation                     *
 *****************************************************************/

n = rows(y);                 /* The sample size.                  */
k = cols(x);                 /* The number of explanatory variables.  */

b = invpd(x'x)*x'y;                                          /*  (5.1) */
e = y - x*b;                                                 /*  (5.2) */
s2 = e'e/(n-k);                                              /*  (5.3) */
vb = s2.*invpd(x'x);                                         /*  (5.4) */
seb = sqrt(diag(vb));                                        /*  (5.5) */
tb = b./seb;                                                 /*  (5.6) */

tss = (y - meanc(y))'(y - meanc(y));                         /*  (5.8) */
rss = e'e;                                                   /*  (5.9) */
ess = tss - rss;                                             /* (5.10) */
```

```
r2 = ess/tss;                                                    /* (5.11) */
ar2 = 1 - (rss/(n-k))/(tss/(n-1));                               /* (5.12) */
f = (ess/(k-1))/(rss/(n-k));                                     /* (5.13) */
d = (e[1:(n-1),1] - e[2:n,1])'(e[1:(n-1),1] - e[2:n,1])/(e'e); /* (5.14) */

/*****************************************************************
 *              Printing the Main Estimation Results             *
 *****************************************************************/
name = 0 $+ "BETA" $+ ftocv(seqa(0,1,k),1,0); /*  A character vector of
                                                  parameter names:
                                                  BETA0, BETA1, ...     */
result = name~b~seb~tb;   /* Combining the computation results together. */

"                     REGRESSION ESTIMATION";
"       ------------------------------------------------";
"          PARAMETER     ESTIMATE      S.E.      T-RATIO";
"       ------------------------------------------------";
call printfm(result,0~ones(1,3),"s"~17~8|"lf"~14~4|"lf"~12~4|"lf"~12~4);
"       ------------------------------------------------";
format /rd 9,4;
"        SIGMA SQUARE  "  s2;
"          R SQUARE    "  r2;
"        ADJ. R SQUARE " ar2;
"        DW STATISTIC  "   d;
"       ------------------------------------------------";?;?;

/*****************************************************************
 *                     Analysis of Variance                      *
 *****************************************************************/
essout = ess~(k-1)~(ess/(k-1))~f;
rssout = rss~(n-k)~(rss/(n-k));
tssout = tss~(n-1);
"                      ANALYSIS OF VARIANCE";
"    ---------------------------------------------------------";
"       SOURCE        SS        df          MS          F";
"    ---------------------------------------------------------";
"      EXPLAINED";;
call printfm(essout,ones(1,4),"lf"~11~4|"lf"~7~0|"lf"~15~4|"lf"~12~4);?;
"      RESIDUAL ";;
call printfm(rssout,ones(1,3),"lf"~11~4|"lf"~7~0|"lf"~15~4);?;
"    ---------------------------------------------------------";
"        TOTAL  ";;
call printfm(tssout,ones(1,2),"lf"~11~4|"lf"~7~0);?;
"    ---------------------------------------------------------";?;?;
```

```
84  end;
```

The estimation results will be printed on the screen and into the ASCII file 'ols.out'. Since there are more than one screenful of the results, we have to use the editor to go into the output file ols.out to view the complete results.

An important feature of the above GAUSS program is that the OLS estimation is conducted in a way that is independent of the data it processes. In other words, all we have to do for different applications is to load the appropriate data into the matrices 'y' and 'x', while the main body of the program can be used repeatedly without any modification. This structure of the program demonstrates the power of GAUSS programming. Once a program proves to work well, all the future applications can be made with minimum revisions. To illustrate this point, let's consider another OLS estimation based on the International Consumption Data, in which we extend the Engel curve model by including price terms to get the following *demand model*:

$$s_{ic} = \beta_1 + \beta_2 \ln m_c + \beta_3 \ln p_{1c} + \cdots + \beta_{12} \ln p_{10,c} + \varepsilon_{ic}, \qquad c = 1, \ldots, 60, \tag{5.16}$$

where $\ln p_{ic}$, $i = 1, \ldots, 10$, are the log prices of the 10 commodities. In such a linear demand model the matrix $\mathbf{X}$ includes 12 columns of data: the constant term, the log income, and the 10 log prices. The vector $\boldsymbol{\beta}$ contains 12 regression coefficients.

$$\mathbf{X} = \begin{bmatrix} 1 & \ln m_1 & \ln p_{11} & \cdots & \ln p_{10,1} \\ 1 & \ln m_2 & \ln p_{12} & \cdots & \ln p_{10,2} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \ln m_{60} & \ln p_{1,60} & \cdots & \ln p_{10,60} \end{bmatrix} \quad \text{and} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{12} \end{bmatrix}.$$

To estimate such a linear regression model, all we have to do is to rewrite the first part of the above program as follows:

```
1  load q[60,10] = a:\data\volume;
2  load s[60,10] = a:\data\share;
3  load m[60,1] = a:\data\totalexp;
4
5  p = (m.*s)./q;                    /* Deriving the prices.                */
6  y = s[.,1];
7  x = ones(60,1)~ln(m)~ln(p);
```

These estimations of the Engel curve model and the demand model should demonstrate the flexibility and power of the GAUSS programming.

There is a question about the above program we might ask: except the print-out in the ASCII output file, the program does not seem to save any of the resulting matrices. Obviously, a potentially useful modification of the program is to save some matrices, such as the regression coefficient estimates 'b', in matrix files using the 'save' command.

One of reasons for saving the regression coefficient estimates is that we may want to run the program repeatedly for all 10 commodities and then compare the 10 sets of regression coefficient estimates. Such a

comparison is particularly interesting because in a system of 10 demand equations we expect the following results: the sum of the 10 intercept estimates (b_0) is equal to one, the sum of the 10 slope estimates for the log income is zero, and the sum of the 10 slope estimates for every price is also zero. (Why?) If we save the regression coefficient estimates for all 10 commodities (be sure that they are 10 matrix files with different file names), then it is easy to write a small GAUSS program to verify the above results.

Another reason for saving the regression coefficient estimates stems from the fact that the goal of demand estimation is to obtain estimates of the income and price elasticities. Due to the particular functional form of the demand equation, we cannot read those elasticities directly from the regression coefficients $\boldsymbol{\beta}$. Some additional computation using GAUSS is necessary. Recall that the definition of the income elasticity is

$$\eta = \frac{\partial \ln q}{\partial \ln m} = \frac{m}{q} \cdot \frac{\partial q}{\partial m},$$

where q is the quantity demanded for a particular commodity and m is income. The regression coefficient β_2 of the log income term in our demand equation has the following partial derivative interpretation:

$$\beta_2 = \frac{\partial s}{\partial \ln m} = \frac{\partial (p \cdot q/m)}{\partial \ln m},$$

where s is the budget share and p is the price. With a little algebra we can show that

$$\eta = 1 + \frac{\beta_2}{s}.$$

Note that in deriving this formula the price p is considered exogenous and thus is not a function of income.

Suppose the regression coefficient estimates for Food (i.e., 'b' in the previous program) has been saved as the matrix file `food_est.fmt` in the root subdirectory, the GAUSS program for the income elasticity computation is

```
new;

load share[60,10] = a:\data\share;
load b = a:\food_est;
eta = 1 + b[2]./share[.,1];

" The income elasticity estimates for 60 countries: "; eta;?;?;
" The average income elasticity estimate: "; meanc(eta);

end;
```

The computation results are only shown on the screen and not saved. Note that care must be taken about the indexing of 'b' and 'share' in the definition of 'eta'. A similar program can also be written for the price elasticity and it is left as an exercise.

5.3 The 'ols' Command

Although it is really quite straightforward to write a program for the OLS estimation, GAUSS also has a command – the 'ols' command – for it (so the program we wrote earlier is in fact superfluous). In this subsection we will examine what the 'ols' command does.

We will again use the demand estimation based on the International Consumption Data as the example. It is important to note that, unlike the previous program, we do not include a column of ones in the 'x' matrix here. The whole process of applying the 'ols' command is shown as follows:

```
new;

load q[60,10] = volume;
load s[60,10] = share;
load m[60,1] = totalexp;

p = (s.*m)./q;
y = s[.,1];
x = ln(m)~ln(p);

output file = ols.out reset;
__altnam = "Cnst"|"Ln_m"|"Food"|"Bev_tob"|"Clothing"|"Rent"|"HouseFur"|
           "Medicare"|"Transpt"|"Recreatn"|"Educatn"|"Other"|"Food_S";

call ols(0,y,x);

end;
```

The 'ols" command takes three inputs. The first input was set 0 here and it can be kept as it is in most applications. (Its meaning will be explained in appendix B.) The second and the third inputs of the 'ols" command are the data matrices for the dependent variables and the explanatory variables, respectively. As mentioned above, the data matrix for the explanatory variables does not contain the constant term although the estimation still includes it. We also see a strange new 12×1 character vector '__altnam' has been placed before the 'ols' command. We will explain its meaning later. An ASCII file 'ols.out' is opened right before *calling* the 'ols' command since we want all outputs of the 'ols' command to be printed in this ASCII file. The print-out in the ASCII output file is shown in the next page.

Explanations of this computer print-out are as follows:

1. Most numbers on top of the table and the first four columns in the table are quite self-explanatory. For example, 'Std error of est' is $\hat{\sigma}^2$; 'Rbar-squared' is the adjusted R^2, etc. As to 'F(11,48)', it is the F test statistic with the degrees of freedom $k-1 = 12-1 = 11$ and $n-k = 60-12 = 48$.

2. 'Probability of F' and the fifth column in the table are the so-called p-values for the corresponding F test and t test statistics, respectively. We now present the definition of the p-values for the t test statistics $t_j = b_j/\text{s.e.}(b_j)$. Recall that this statistic is used to test the null hypothesis H_0: $\beta_j = 0$. Given a computed t test statistic t_j^*, instead of comparing it with the critical values from the t distribution table, we can compute its p-value p and then compare the p-value with the size of the test (usually one half of the 5% or 1% in a two-tail test). The definition of p-value is $\Pr(T > |t_j^*|)$ where T is a random variable of the t distribution (with the $n-k$ degrees of freedom). That is, the p-value is the probability for a t random variable T to be greater than the (absolute) value of t_j^*. The null hypothesis will be rejected if p is smaller than the given size of the test. For example, the p-value 0.001 for the 'ln_m' is smaller than either 2.5% or 0.5% so that the null hypothesis of about

```
Valid cases:                 60    Dependent variable:              Food_S
Missing cases:                0    Deletion method:                   None
Total SS:             10706.924    Degrees of freedom:                  48
R-squared:                0.766    Rbar-squared:                     0.712
Residual SS:           2505.546    Std error of est:                 7.225
F(11,48):                14.283    Probability of F:                 0.000

                       Standard               Prob   Standardized  Cor with
Variable    Estimate      Error      t-value    >|t|     Estimate    Dep Var
--------------------------------------------------------------------------------
Cnst       50.848501   10.763414    4.724198   0.000        ---         ---
Ln_m       -8.797151    2.566394   -3.427826   0.001  -1.626372   -0.292045
Food        1.686658    5.044760    0.334339   0.740   0.277730    0.075971
Bev_tob     5.236882    3.612661    1.449591   0.154   0.828485    0.173129
Clothing    1.507999    3.872228    0.389440   0.699   0.251680    0.060411
Rent        0.926101    2.551713    0.362933   0.718   0.153014    0.087702
HouseFur    4.350814    3.660329    1.188640   0.240   0.738807    0.014162
Medicare   -1.606652    3.176194   -0.505842   0.615  -0.276045    0.002089
Transpt    -4.483241    3.197497   -1.402110   0.167  -0.745650    0.069922
Recreatn    6.517217    3.680082    1.770943   0.083   1.075507    0.078659
Educatn    -3.036182    3.312132   -0.916685   0.364  -0.533106   -0.144056
Other      -1.396618    4.391565   -0.318023   0.752  -0.224140    0.094973
```

the regression coefficient for the log income will be rejected in a two-tail test (i.e., the effect of log income is significant). The definition and the use of the p-value for the F test are similar.

There are a number of GAUSS commands which can also be used to evaluate the p-values for the various test statistic with different distributions. For example, to evaluate the p-values of an F test statistic 14.283 computed from the previous program, we use

```
p_value = cdffc(14.283,11,48);
```

In this 'cdffc' command, the first input is the computed F-test statistic, the second and the third inputs are the two degrees of freedom. The more precise definition of the 'cdffc' command is that it gives one minus the value of the cumulative F distribution function at 14.283. Other than F-distribution, we have the following GAUSS command for other distributions:

```
p = cdfnc(z);          /* For the normal distribution.                  */
p = cdftc(t,df);       /* For the t distribution. The second input is
                          the degree of freedom.                        */
p = cdfchic(t,df);     /* For the chi-square distribution. The second
                          input is the degree of freedom.               */
```

Obviously, if we apply the 'cdftc' command to the absolute value of the t-ratio for log income: -3.427826: i.e., 'cdftc(3.427876,48)', the resulting p-value should be 0.001, as shown in the fifth column of the above table.

3. The numbers in the last two columns of the table are two new statistics we haven't considered yet. The last column gives the estimated correlation coefficient between the dependent variable Y_i and each of the explanatory variables X_{ji}, which is defined as

$$\frac{\sum_{i=1}^{n}(Y_{ji}-\bar{Y}_j)(X_{ji}-\bar{X}_j)}{\sqrt{\sum_{i=1}^{n}(Y_i-\bar{Y})^2\cdot\sum_{i=1}^{n}(X_{ji}-\bar{X}_j)^2}}.$$

Given the OLS estimate b_j for the j-th regression coefficient, the standardized estimate in the sixth column is defined by

$$b_j\sqrt{\frac{\sum_{i=1}^{n}(X_{ji}-\bar{X}_j)^2}{\sum_{i=1}^{n}(Y_i-\bar{Y})^2}}.$$

4. The above tabulated estimation results are not the only output from the 'ols' procedure. In fact, eleven additional output matrices are also produced. They are suppressed in the above examples, but can be retrieved if we type

```
{vnam,mmt,b,stb,vb,seb,s2,cor,r2,e,d} = ols(0,y,x);
```

Here in front of the 'ols' command we have 11 matrix names enclosed in braces and an equality sign. Besides the printed results in the output ASCII file, those 11 output matrices are also the computation outputs of the 'ols' command.

Among the 11 output matrices the following seven are something we are familiar with: 'b' (the OLS estimate *b*), 'vb' (the estimated variance-covariance matrix of **b**), 'seb' (the standard errors of **b**), 's2' (σ^2), 'r2' (R^2), 'e' (the residual $\mathbf{e} = \mathbf{y} - \mathbf{Xb}$), and 'd' (the Durbin-Watson statistic). The output vector 'stb' contains the standardized OLS estimates for those non-constant explanatory variables which has just been defined above. The output vector 'vnam' contains the variable names, including those for the constant term (the first element) and the dependent variable (the last element). The matrix 'mmt' gives the cross-product (i.e., $[\,\mathbf{X}\ \ \mathbf{y}\,]'[\,\mathbf{X}\ \ \mathbf{y}\,]$) and 'cor' contains the correlation coefficient matrix for the non-constant explanatory variables and the dependent variable (which is included as the last variable). In general the residual vector 'e' and the scalar 'd' for the Durbin-Watson statistic are 0 unless we reset the *switch* '_olsres' from its default value 0 to 1 before executing the 'ols' command. The explanation of *switch* will be given below.

Besides the print-out in the output ASCII file, why do we need those 11 output matrices? The reason is that we may use those matrices for additional computation. For example, the residual vector 'e' may be used to compute additional test statistics. This computation can proceed directly after the 'ols' command in the same program.

5. We now go back to the explanation of the new expression '__altnam' in the program. The 'ols" command offers six *switches* (their formal names are *global variables* which will be discussed more fully in section 8.2) with which we can control some aspects of the OLS estimation. These switches are simply matrix variables whose values are to be set by us. The 'ols" command will check the values of these switches before implementing its calculation. These switches all have default values which can usually be left as they are. Nevertheless, we can easily change those aspects of the 'ols' command that are controlled by the switches by assigning different values to the switches. Here, we will discuss three switches only: the matrix variables '__con', '__altnam', and '_olsres'. Note that the names of the first two variables are prefixed with two underlines. This unique form distinguishes them from other variable names. Let's now consider the meanings of these two switches:

   (1) The switch '__con':
   If we do not want to include the constant term in the OLS estimation, then we should set the switch '__con' to 0 before calling the 'ols' procedure. The default value of '__con' is 1, with which the constant term will be included in the OLS estimation. In the above program the redefinition of the switch '__con' is not included, so the estimation will include the constant term (even if the data matrix 'x' does not contain a column for the constant term.)

   (2) The switch '__altnam':
   If we want to give specific names to the explanatory variables and the dependent variable, we need to assign a character vector of names to the switch '__altnam', where the first one should always be the name for the constant term if it is included and the last one should be the name for the dependent variable. This is exactly how we did in the above program. So the name for the constant term is 'Cnst', the name for the log income variable is 'Ln_m', etc. The name for the dependent variable, specified as the last character, is 'Food_S'. Note that the size of the '__altnam' vector is 13×1. If the switch '__altnam' is not included in the program, then the default names for the explanatory variables are 'CONSTANT', 'X1', 'X2', 'X3', ...

(3) The switch '_olsres':

As mentioned above, the residual vector 'e' and the scalar 'd' for the Durbin-Watson statistic in the command '{vnam,mmt,b,stb,vb,seb,s2,cor,r2,e,d} = ols(0,y,x)' are set at 0 unless the switch '_olsres' is altered from its default value 0 to 1.

5.4 Linear Restrictions

Given the standard linear regression model $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$, if the parameter $\boldsymbol{\beta}$ is subject to J linear restrictions which can be expressed as

$$\mathbf{R}\boldsymbol{\beta} = \mathbf{q},$$

where $\mathbf{R}$ is a $J \times k$ matrix of full row rank, $\mathbf{q}$ is a $J \times 1$ vector, and both are known matrices, then the restricted OLS estimator of $\boldsymbol{\beta}$ is

$$\mathbf{b}^* = \mathbf{b} - (\mathbf{X}'\mathbf{X})^{-1}\mathbf{R}'[\mathbf{R}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{R}']^{-1}(\mathbf{R}\mathbf{b} - \mathbf{q}),$$

where $\mathbf{b}$ is the usual unrestricted OLS estimator. The corresponding restricted residual is

$$\mathbf{e}^* = \mathbf{y} - \mathbf{X}\mathbf{b}^* = \mathbf{y} - \mathbf{X}\mathbf{b} + \mathbf{X}(\mathbf{b} - \mathbf{b}^*) = \mathbf{e} + \mathbf{X}(\mathbf{b} - \mathbf{b}^*),$$

where $\mathbf{e}$ is the usual unrestricted residual. Since $\mathbf{X}'\mathbf{e} = \mathbf{0}$, we have

$$\mathbf{e}^{*\prime}\mathbf{e}^* = \mathbf{e}'\mathbf{e} + (\mathbf{b} - \mathbf{b}^*)'\mathbf{X}'\mathbf{X}(\mathbf{b} - \mathbf{b}^*) = \mathbf{e}'\mathbf{e} + (\mathbf{R}\mathbf{b} - \mathbf{q})'[\mathbf{R}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{R}']^{-1}(\mathbf{R}\mathbf{b} - \mathbf{q}).$$

Recall that the OLS estimator $\mathbf{b}$ is distributed as $\mathcal{N}(\boldsymbol{\beta}, \sigma^2(\mathbf{X}'\mathbf{X})^{-1})$. So if the linear restrictions $\mathbf{R}\boldsymbol{\beta} = \mathbf{q}$ is true, then

$$\frac{(\mathbf{R}\mathbf{b} - \mathbf{q})'[\mathbf{R}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{R}']^{-1}(\mathbf{R}\mathbf{b} - \mathbf{q})}{\sigma^2} = \frac{\mathbf{e}^{*\prime}\mathbf{e}^* - \mathbf{e}'\mathbf{e}}{\sigma^2},$$

has a χ^2 distribution with J degrees of freedom and is independent of

$$\frac{\mathbf{e}'\mathbf{e}}{\sigma^2} = \frac{(n-k)s^2}{\sigma^2},$$

which also has a χ^2 distribution with $n-k$ degrees of freedom. Recall that $\mathbf{e}'\mathbf{e}$ is referred to as RSS (residual sum of squares) before. We can denote $\mathbf{e}^{*\prime}\mathbf{e}^*$ as RSS*. Consequently, if the linear restrictions $\mathbf{R}\boldsymbol{\beta} = \mathbf{q}$ is true, then the ratio

$$\frac{(\mathbf{R}\mathbf{b} - \mathbf{q})'[\mathbf{R}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{R}']^{-1}(\mathbf{R}\mathbf{b} - \mathbf{q})/J}{s^2} = \frac{(\mathrm{RSS}^* - \mathrm{RSS})/J}{\mathrm{RSS}/(n-k)}, \tag{5.17}$$

has an F-distribution with J and $n-k$ degrees of freedom, a fact that can be used to test the hypothesis

$$H_0\colon\ \mathbf{R}\boldsymbol{\beta} = \mathbf{q} \qquad \text{against} \qquad H_1\colon\ \mathbf{R}\boldsymbol{\beta} \neq \mathbf{q}.$$

Although the left-hand side expression of (5.17) may appear straightforward to compute, the right-hand side expression is more readily to generalize as can be seen in section 5.5.

An Example The difference between an Engel curve model (5.15) and a demand model (5.16) is that the latter includes prices. Which of the two models is more suitable for the International Consumption Data depends on how important the price effects are. The F test is needed to decide whether the price effects are jointly significant. To conduct such a test, we start with the demand model which can be represented by the general notation $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$, with $\mathbf{X}$ including a constant term, log income, and ten log prices. Among the twelve elements of $\boldsymbol{\beta}$, what we try to decide is whether the last ten (the coefficients for the log prices) are equal to zero or not. In other words, the hypothesis is whether the ten price coefficients are equal to zero or not. Formally, this null hypothesis can be expressed as a set of ten linear restrictions on the regression coefficient vector $\boldsymbol{\beta}$ as follows:

$$\mathbf{R}\boldsymbol{\beta} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \beta_{12} \end{bmatrix} = [\, \mathbf{0} \;\; \mathbf{0} \;\; \mathbf{I}_{10} \,] \cdot \boldsymbol{\beta} = \begin{bmatrix} \beta_3 \\ \beta_4 \\ \beta_5 \\ \vdots \\ \beta_{12} \end{bmatrix} = \mathbf{0}.$$

where $\mathbf{I}_{10}$ is a 10×10 identity matrix and $\mathbf{0}$ is a 10×1 vector of zeros. Given this setup, then it is fairly straightforward to write a program to test the significance of the price effects:

```
new;

load q[60,10] = volume;
load s[60,10] = share;
load m[60,1] = totalexp;

p = (m.*s)./q;

y = s[.,1];
x = ones(60,1)~ln(m~p);
n = rows(y);
k = cols(y);

e = y - x*invpd(x'x)*x'y;
s2 = e'e/(n-k);
r = zeros(10,2)~eye(10);
f = (r*b)'invpd(r*invpd(x'x)*r')*r*b/(10*s2);
pv = cdffc(f,10,n-k);

output file = test.out on;
format /rd 8,4;
" The F test statistic is " f;;" with a p-value " pv;

end;
```

5.5 Chow Test for Structural Changes

One important implicit assumption about the linear regression model $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$ is that the parameter $\boldsymbol{\beta}$ is constant over the entire sample so that it can be estimated uniquely by the OLS estimation. However, in many applications the sample may contain structural changes which can cause the true values of some or all elements of the $\boldsymbol{\beta}$ vector to vary. As a result, we have to estimate more than one set of $\boldsymbol{\beta}$. This can be accomplished by dividing the sample into subsamples and assuming that the value of $\boldsymbol{\beta}$ is constant in each of these subsamples but different across subsamples. Here, we only consider the simplest case where there is a single structural change so that the true values of some or all elements of the $\boldsymbol{\beta}$ vector for the first n_1 observations are different from those for the rest $n - n_1$ observations. More specifically, we consider the following two specifications:

Case 1: only the intercept differs across the two subsamples so that the regression model becomes

$$\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{1}_1 & \mathbf{0} & \tilde{\mathbf{X}}_1 \\ \mathbf{0} & \mathbf{1}_2 & \tilde{\mathbf{X}}_2 \end{bmatrix} \begin{bmatrix} \beta_{1a} \\ \beta_{1b} \\ \boldsymbol{\beta}_2 \end{bmatrix} + \boldsymbol{\varepsilon},$$

where $\mathbf{1}_1$ and $\mathbf{1}_2$ are two vectors of ones whose dimensions are $n_1 \times 1$ and $(n - n_1) \times 1$, respectively; and $\tilde{\mathbf{X}}_1$ and $\tilde{\mathbf{X}}_2$ are $n_1 \times (k-1)$ and $(n - n_1) \times (k-1)$ matrices, respectively, containing observations on the explanatory variables excluding the constant term. β_{1a} and β_{1b} are two intercepts that reflect the effect of the structural change, while the $(k-1) \times 1$ vector $\boldsymbol{\beta}_2$ contains the parameters that are not affected by the structural change and remain the same across the two subsamples. The number of parameters in this extended model is $k + 1$ because an additional intercept.

Case 2: all k parameters in $\boldsymbol{\beta}$ differ across the two subsamples so that the regression model becomes

$$\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{X}_1 & \mathbf{O} \\ \mathbf{O} & \mathbf{X}_2 \end{bmatrix} \begin{bmatrix} \boldsymbol{\beta}_1 \\ \boldsymbol{\beta}_2 \end{bmatrix} + \boldsymbol{\varepsilon},$$

where $\mathbf{X}_1$ and $\mathbf{X}_2$ are $n_1 \times k$ and $(n - n_1) \times k$ matrices, respectively, and $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$ are two sets of parameter vectors that reflect the effect of the structural change. The number of parameters is $2k$ because there are two full sets of $\boldsymbol{\beta}$.

Given the two generalized models in case 1 and case 2, the original model is in fact a *restricted model* which is based on the hypothesis that there is no structural change and there is only one set of parameter $\boldsymbol{\beta}$ in

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon} \qquad \text{or} \qquad \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{1}_1 & \tilde{\mathbf{X}}_1 \\ \mathbf{1}_2 & \tilde{\mathbf{X}}_2 \end{bmatrix} \boldsymbol{\beta} + \boldsymbol{\varepsilon} \qquad \text{or} \qquad \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{bmatrix} \boldsymbol{\beta} + \boldsymbol{\varepsilon}.$$

Here, the number of parameters is k as usual. We note the numbers of parameters in case 1 and case 2: $k+1$ and $2k$, respectively, are both greater than k, reflecting the original model is indeed a restricted model with fewer parameters.

We can fit the model of no structural change, as opposed to case 1 and case 2 structural changes, into the framework of the general linear restrictions. Given the case 1 structural change, the *unrestricted* parameter vector contains β_{1a}, β_{1b}, and $\boldsymbol{\beta}_2$, while the $\mathbf{R}$ matrix corresponding to the restricted model of no structural change is a $1 \times (k+1)$ vector $[1 \; -1 \; \mathbf{0}']$, where $\mathbf{0}$ is a $k \times 1$ vector of zeros. So the restriction $\mathbf{R}\boldsymbol{\beta} = \mathbf{q}$ can be written as

$$\begin{bmatrix} 1 & -1 & \mathbf{0}' \end{bmatrix} \begin{bmatrix} \beta_{1a} \\ \beta_{1b} \\ \boldsymbol{\beta}_2 \end{bmatrix} = 0.$$

Similarly, in case 2 the restricted model of no structural change imposes the following restriction on the parameter vector:

$$\begin{bmatrix} \mathbf{I}_k & -\mathbf{I}_k \end{bmatrix} \begin{bmatrix} \boldsymbol{\beta}_1 \\ \boldsymbol{\beta}_2 \end{bmatrix} = \mathbf{0},$$

where $\mathbf{I}_k$ is a $k \times k$ identity matrix.

The above expressions of the $\mathbf{R}$ matrix can be used to formulate statistics for testing the null hypothesis of no structural change against the alternative hypothesis of case 1 and case 2, respectively. However, an easier way to construct the test statistics is based on residuals and the corresponding RSS from each of the three models. Let RSS be the residual sum of squares from the restricted model (i.e., the original model) and RSS_1 and RSS_2 from the case 1 model and the case 2 model, respectively.

1. The F-test statistic against case 1 structural change is

$$F_1 = \frac{(\text{RSS} - \text{RSS}_1)/1}{\text{RSS}_1/(n-k-1)},$$

   with degrees of freedom 1 and $n-k-1$.

2. The F-test statistic against case 2 structural change is

$$F_2 = \frac{(\text{RSS} - \text{RSS}_2)/k}{\text{RSS}_2/(n-2k)},$$

   with degrees of freedom k and $n-2k$.

3. Finally, we note case 1 is in fact a special case of case 2. So we can test case 1 against case 2 using the following test statistic:

$$F = \frac{(\text{RSS}_1 - \text{RSS}_2)/(k-1)}{\text{RSS}_2/(n-2k)}.$$

   The degrees of freedom are $k-1$ and $n-2k$.

These F-tests, particularly F_2, are usually referred to as the Chow test. The GAUSS program for Chow tests is quite straightforward. What needs to be done is simply the calculation of three types of residuals and the corresponding RSS, as well as the degrees of freedom. From equations (5.2) and (5.9) we know RSS is defined by

$$\text{RSS} = \mathbf{e}'\mathbf{e} = (\mathbf{y} - \mathbf{X}\mathbf{b})'(\mathbf{y} - \mathbf{X}\mathbf{b}) = \mathbf{y}'\mathbf{y} - \mathbf{y}'\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}.$$

Also, each RSS is associated with a particular degree of freedom, which is equal to the sample size (i.e., the row number of the **X** matrix) minus the number of parameters (i.e., the column number of the **X** matrix). Furthermore, by a closer inspection of each of the Chow test statistics, we find that the denominator is an RSS divided by its associated degree of freedom, while the numerator is the difference between two RSS's, divided by the difference between the two associated degrees of freedom. This general pattern helps simplifying the GAUSS program.

Given the $n \times k$ matrix of the explanatory variables 'x' from the original model (i.e., the restricted model), in which the first 'n1' observations belong to the first subsample and the rest 'n - n1' observations belong to the second subsample, we have to carefully organize the $n \times (k+1)$ and $n \times 2k$ matrices 'x1' and 'x2' for the two unrestricted models – case 1 and case 2, respectively. Once the three matrices of explanatory variables for the three models are defined, then we can use the same formulas to compute the RSS and the associated degrees of freedom for the three models, followed by the corresponding Chow test statistics, 'f1', "f2', and 'f'. The p-values of these test statistics can also be computed easily.

```
/* The expanded matrix of explanatory variables for the case 1 model.    */
x1 = (ones(n1,1)~zeros(n1,1)~x[1:n1,2:k])|
     (zeros(n-n1,1)~ones(n-n1,1)~x[(n1+1):n,2:k]);

/* The expanded matrix of explanatory variables for the case 2 model.     */
x2 = (x[1:n1,.]~zeros(n1,k))|(zeros(n-n1,k)~x[(n1+1):n,.]);

rss = y'y - y'x*invpd(x'x)*x'y;      /* RSS from the restricted model.   */
df = rows(x) - cols(x);              /* The degree of freedom of 'rss'.  */

rss1 = y'y - y'x1*invpd(x1'x1)*x1'y; /* RSS from the case 1 model.       */
df1 = rows(x1) - cols(x1);           /* The degree of freedom of 'rss1'. */

/* The F test statistic for testing case 1 against the restricted model. */
f1 = ((rss - rss1)/(df1 - df)/(rss1/df1);
/*  The p-value of the test statistic.     */
pv1 = cdffc(f1,df1-df,df1);

rss2 = y'y - y'x2*invpd(x2'x2)*x2'y; /* RSS from the case 2 model.       */
df2 = rows(x2) - cols(x2);           /* The degree of freedom of 'rss2'. */
f2 = ((rss - rss2)/(df2 - df)/(rss2/df2);
pv2 = cdffc(f2,df2-df,df2);

/*  The F test statistic for testing case 2 against case 1.              */
f = ((rss1 - rss2)/(df2 - df1)/(rss2/df2);
pv = cdffc(f,df2-df1,df2);
```

Recall that the null hypothesis will be rejected if the p-value is smaller than the designated size (usually 5% or 1%). The most interesting feature of the above program is the similarity among the three sets of commands for computing the three test statistics.

Chapter 6

Relational Operators and Logic Operators

Other than the usual arithmetic operators, there are relational and logic operators in GAUSS which produce results that can have only two values: either "true" (recorded as the number 1) or "false" (recorded as the number 0).

6.1 Relational Operators

In GAUSS there are six relational operators. Each of these relational operators has two equivalent notation:

1. '<' (or 'lt'), which means "smaller than".
2. '<=' (or 'le'), which means "smaller than or equal to".
3. '==' (or 'eq'), which means "equal to".
4. '/=' (or 'ne'), which means "not equal to".
5. '>' (or 'gt'), which means "greater than".
6. '>=' (or 'ge'), which means "greater than or equal to".

Consider the example:

```
a = (x /= y);
```

If 'x' and 'y' are two scalar variables with different values, then the right-hand side relational operation will be true and the value of 'a' will be '1'. If 'x' and 'y' have the same values, then the result on the right-hand side relational operation is false and the value of 'a' will be '0'. Note that the relational operator for "equal to" consists of two equal signs '==', which is very different in meaning from the single equal sign '='.

If the relational operators are preceded by a dot '.' then they become element-by-element operators. For example, if 'x' and 'y' are two $n \times 1$ vectors and

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

then 'a' will be an $n \times 1$ vector of 0 and 1, representing the results of n relational operations between the n corresponding elements of 'x' and 'y'. Also, if the dimensions of 'x' and 'y' are not the same, then the usual rule for element-by-element operations will apply.

The relational operators, particularly '==' and '/=', can also be applied to character vectors or strings with each operator preceded by '$'. For example,

```
a = ("old" $/= "young");
b = (region .$== "SOUTH");
```

The value of ‘a’ is 1 since the relation on the right-hand side is always true. The dimension of ‘b’ will be the same as that of the character vector ‘region’. Whether the value of an element of ‘b’ is 1 or 0 depends on whether the corresponding element in ‘region’ contains characters ‘SOUTH’ or not.

6.2 Logic Operators

There are five logic operators in GAUSS: ‘and’, ‘or’, ‘not’, ‘xor’, and ‘eqv’. Among them the first three are used most often.

```
a = (x and y);
b = (x or y);
c = not x;
e = (x eqv y);
f = (x xor y);
```

Here, the values in both scalars ‘x’ and ‘y’ must be either 0 and 1; that is, they themselves may be the true/false results of some relational or logical operations. The value of ‘a’ is 1 when both ‘x’ and ‘y’ are 1; ‘a’ is 0 for all other cases. The value of ‘b’ is 0 when both ‘x’ and ‘y’ are 0; ‘b’ is 1 for all other cases. The value of ‘c’ is just opposite to that of ‘x’: if ‘x’ is 1, then ‘c’ is 0; and vice versa. The value of ‘e’ is 1 when ‘x’ and ‘y’ are both 1 or both 0. The value of ‘f’ is 1 when ‘x’ and ‘y’ have opposite values. The logical operators can also be made as element-by-element operators if they are preceded by a dot ‘.’ So ‘.and’, ‘.or’, ‘.not’, ‘.eqv’, and ‘.xor’ are all element-by-element logical operators.

6.3 Conditional Statements

The true/false results of relational or logic operations are usually used as conditions for determining whether to execute a set of commands through the following design:

```
if (true/false statement A);
   expressions I;
endif;
expressions II;
```

If the ‘true/false statement A’ is true, then ‘expressions I’ will be executed, followed by ‘expressions II’ after the ‘endif’ expression. If the ‘true/false statement A’ is false, then ‘expressions I’ will be skipped and only ‘expressions II’ will be executed. Each ‘if’ command must be paired with an ‘endif’ command. Here is a simple example:

```
if a <= 0;
   "a is not positive";
   end;
endif;
b = sqrt(a);
```

The 'if...endif' statement here can be regarded as a safety device to avoid taking a square root of a non-positive number. If the condition 'a <= 0' holds, then the program will print a message 'a is not positive' and terminate immediately. Otherwise, it will move on to the expression after the 'endif' command and take square root of the value of 'a'.

The 'if...endif' command can be extended to include a few 'elseif' commands and/or a 'else' in between for more options as follows:

```
if (true/false statement A);
   (expressions I)
elseif (true/false statement B);
   (expressions II)
else;
   (expressions III)
endif;
   (expressions IV)
```

If the 'true/false statement A' is true, then 'expressions I' will be executed, followed by 'expressions IV' after the 'endif' expression. If the 'true/false statement A' is false and 'true/false statement B' is true, then 'expressions II' will be executed, followed by 'expressions IV'. If both 'true/false statement' 'A' and 'B' are false, then 'expressions III' will be executed, followed by 'expressions IV'.

Let's consider an example:

```
if a < 0;
   x = 1;
elseif a == 0;
   x = 2;
elseif a > 0 .and a < 1;
   x = 3;
else;
   x = 4;
endif;
```

The value of 'x' is 1 if 'a' is a negative number; 'x' is 2 if 'a' is zero; 'x' is 3 if 'a' lies between 0 and 1, and 'x' is 4 for all other values of 'a'.

6.4 Row-Selectors: the 'selif' and 'delif' Commands

A useful technique for data rearrangement is to select rows (observations) from a data matrix based on the values of the corresponding elements of a column vector. For example, given the 60×10 matrix 's' of budget share data on 10 commodities for 60 countries, we want to select those observations with the Food budget share (the first column) lying between 0.3 and 0.8. In other words, the process of selecting observations is based on a relational operation on the single variable – the Food budget share. Situations like this can be easily handled using the powerful 'selif' or 'delif' commands:

```
e = (s[.,1] .>= 0.3 .and s[.,1] .<= 0.8);
s1 = selif(s,e);
```

Here, 'e' is a 60×1 vector of 1 and 0, representing the results of element-by-element relational operations. The 'selif' command will pick those rows of the 's' matrix which correspond to all the "1" in the 'e' vector, delete those rows of the 's' matrix which correspond to all the "0" in the 'e' vector, and then form a smaller matrix 's1'. A similar command is

```
f = (s[.,1] .< 0.3 .or s[.,1] .> 0.8);
s2 = delif(s,f);
```

where 'f' is again a 60×1 vector of 1 and 0, representing the results of element-by-element relational operations which are exactly opposite to the 'e' vector before. The 'delif' command will delete those rows of the 's' matrix which correspond to all the "1" in the 'e' vector, and then form the resulting 's2' matrix. Obviously, the two resulting matrices 's1' and 's2' are identical.

The two arguments of the 'selif' and 'delif' commands should have the same row number while the second argument must be a column vector of 0 and 1, which usually are the results of element-by-element relational operations.

6.5 Dummy Variables in Linear Regression Models

In this section we will first review how dummy variables work in a simple linear regression model:

$$y_i = \alpha + \beta x_i + \varepsilon_i,$$

we then suggest how dummy variables can be constructed in GAUSS. We will find that the element-by-element relational operators and logical operators are very useful in generating dummy variables.

6.5.1 Binary Dummy Variables

A dummy variable, say, d_i, is used to characterize two mutually exclusive categories. Based on which of the two categories an observation i belongs to, we assign a value, either 0 or 1, to the dummy variable d_i. Including a dummy variable in a linear regression model allows us to examine how the difference between the two categories may affect the dependent variable. For example, the sampled countries of the International Consumption Data can be divided into two categories: North American/European and the

others. The North American/European countries can be broadly referred to as the developed countries while all the others the developing countries. When we estimate the Engel curve model using the International Consumption Data, we may wonder the consumption behavior of the richer developed countries may differ from that of the developing countries. To incorporate this idea into the estimation, we need to characterize the developed/developing difference by a dummy variable and include it in the Engel curve model. Such a dummy variable can be defined as follows:

$$d_i = \begin{cases} 1, & \text{if the } i\text{th country is developed,} \\ 0, & \text{if the } i\text{th country is developing.} \end{cases}$$

The way we assign the two values 0 and 1 to the two categories is arbitrary and it can be changed.

There are two approaches to incorporating a dummy variable into the simple linear regression model:

1. *The constant-slope case*: the two categories (the developed countries and the developing countries) are assumed to have different intercepts but the same slope β:

$$y_i = \alpha + \beta x_i + \gamma d_i + \varepsilon_i, = \begin{cases} (\alpha + \gamma) + \beta x_i + \varepsilon_i, & \text{if } d_i = 1, \\ \alpha + \beta x_i + \varepsilon_i, & \text{if } d_i = 0, \end{cases}$$

   where the parameter γ characterizes the difference in the intercepts for the two categories of countries. For example, if the coefficient γ has a positive value, then the intercept, which is $\alpha + \gamma$ for those countries whose dummy variable value is 1 (i.e., the developed countries) will be larger than the intercept, which is only α, of other countries. We note that in this formulation the slope estimates are necessarily the same for both categories of countries.

2. *The varying-slope case*: the two categories are assumed to have different slopes as well as different intercepts:

$$y_i = \alpha + \beta x_i + \gamma d_i + \delta x_i d_i + \varepsilon_i, = \begin{cases} (\alpha + \gamma) + (\beta + \delta) x_i + \varepsilon_i, & \text{if } d_i = 1, \\ \alpha + \beta x_i + \varepsilon_i, & \text{if } d_i = 0, \end{cases}$$

   where an *interaction term* – the product of the dummy variable and the explanatory variable – is included. The parameter γ represents the difference in the intercepts while δ represents the difference in the slopes.

   We should note that this specification is almost the same as running two completely different linear regression models for the two categories separately, except that in the present dummy variable formulation the variance of the error term is assumed to be the same for both categories of observations (there will be two error variances if two completely separate regression models are estimated).

Dummy variables in the 0-1 form usually need to be constructed from the original data where variables may take various numeric and character forms. For example,

1. The International Consumption Data contains a variable, says, 'ctt' which indicates whether a country belongs to one of the five continents: Africa (AF), Asia (AS), North America (NA), South and Central America (SA), and Europe (EU). The codes AF, AS, NA, SA, and EU are characters.

2. The variable like ‘gender’ in many data sets is usually coded in characters ‘Male’ and ‘Female’, instead of 0 and 1.

3. The two categories of marital status – single and married – need to be derived from a more detailed coded variable ‘Marital Status’ which may have four numeric values: 0 for ‘Never Married,’ 1 for ‘Married,’ 2 for ‘Divorced,’ and 3 for ‘Widowed.’

4. The two periods of time – before the World War II (WW II) and after – need to be derived from a numeric variable ‘calendar year.’

5. The two categories – having college education and not having college education – need to be derived from the variable ‘the Years of Schooling’ which may have values from 0 to 22 or more.

6. The two categories – households without children and households with children – need to be derived from the variable ‘Number of Children’ which may have values ranging from 0 to 10.

So it is generally necessary to transform the original variables to the format of dummy variables before they can be included in the linear regression model. Let’s consider the above six examples and assume that the names of the six original variables in GAUSS are ‘ctt’, ‘gender’, ‘marital’, ‘year’, ‘school’, and ‘child_no’, respectively. The GAUSS statements for deriving the corresponding dummy variables from these six variables are as follows:

```
dum1 = (ctt .$== "NA" .or ctt .$== "EU" );
                                /* dum1 = 1, for developed countries;
                                        0, for developing countries.  */

dum2 = (gender .$== "female");  /* dum2 = 1, for female;
                                        0, for male.                  */

dum3 = (marital .== 1);         /* dum3 = 1, for married persons;
                                        0, otherwise.                 */

dum4 = (year .> 1945);          /* dum4 = 1, for after WW II;
                                        0, otherwise.                 */

dum5 = (school .> 12 .and school .<= 16);
                                /* dum5 = 1, for college graduates;
                                        0, otherwise.                 */


dum6 = (child_no ./= 0);        /* dum6 = 1, for families with children
                                        0, otherwise.                 */
```

Some remarks:

1. We exploit the element-by-element relational operations to define dummy variables. Recall that the result of a relational operation is either 0 or 1, depending on whether the result is false or true.

2. All the operations are element-by-element so that the resulting dummy variable has the same dimension as that of the original variable.

3. As in the first and the fifth example, it is often necessary to have logic operators like '`.and`' and '`.or`' when defining dummy variables.

4. When original variables are recorded in characters, all relational operators need to be preceded by '`$`'.

5. The parentheses on the right-hand side of the equality sign are not really necessary. They are there to help us to read the expressions more clearly.

6. Numeric categorical variables like '`marital`' in their original forms cannot be included in the linear regression model directly. This is because the particular codes of these variables may imply metrics for different categories that do not make sense. For example, including the variable '`marital`' directly in a simple linear regression model implies the following specification:

$$y_i = \alpha + \beta x_i + \gamma \times \text{marital}_i + \varepsilon_i, = \begin{cases} \alpha + \beta x_i + \varepsilon_i, & \text{for the 'Never Married,'} \\ (\alpha + 1\gamma) + \beta x_i + \varepsilon_i, & \text{for the 'Married,'} \\ (\alpha + 2\gamma) + \beta x_i + \varepsilon_i, & \text{for the 'Divorced,'} \\ (\alpha + 3\gamma) + \beta x_i + \varepsilon_i, & \text{for the 'Widowed.'} \end{cases}$$

   so that the difference in intercept between the 'Never Married' and the 'Married,' is the same as that between the 'Married' and the 'Divorced,' a result that does not make much sense in any application.

An Example Let's go back to the analysis of International Consumption Data. To examine how the consumption behavior of the richer developed countries differs from that of the poorer developing countries in the Engel curve model, we include a dummy variable as follows:

$$s_{ic} = \alpha + \beta \ln m_c + \gamma d_c + \varepsilon_{ic}, \qquad c = 1, 2, \ldots, 60.$$

Given such an extended Engel curve model, the matrix of explanatory variables $\mathbf{X}$ includes 3 columns: the constant term, the log income, and the dummy variable. The regression coefficient vector $\boldsymbol{\beta}$ contains 3 elements.

$$\mathbf{X} = \begin{bmatrix} 1 & \ln m_1 & d_1 \\ 1 & \ln m_2 & d_2 \\ \vdots & \vdots & \vdots \\ 1 & \ln m_{60} & d_{60} \end{bmatrix} \qquad \text{and} \qquad \boldsymbol{\beta} = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}.$$

In writing a GAUSS program for estimating this extended Engel curve model the key is the construction of the dummy variable. In addition to the three data sets on budget shares, volumes, and total expenditure, the International Consumption Data also include an ASCII file '`country`' with two columns of characters indicating the country names (the first column) and the continent each country belongs to (the second column). As mentioned earlier, the continent indicator can be used to define the dummy variable for developed/developing countries. The five continent codes are '`AF`' (for Africa), '`AS`' (for Asia), '`NA`' (for North

America), 'SA' (for South America), and 'EU' (for Europe). Countries with codes 'NA' and 'EU' will be defined as the developed countries and the rest developing countries.

```
new;

load s[60,10] = share;
load m[60,1] = totalexp;
load c[60,2] = country;

d = (c[.,2] ./= "NA" .and c[.,2] ./= "EU");

y = s[.,1];
x = ones(60,1)~ln(m)~d;              /*  or  x = ln(m)~d;                */

output file = dummy.out reset;
__altnam = "Cnst"|"Ln_m"|"Dummy"|"Share";
call ols(0,y,x);

end;
```

In the above definition of dummy variable 'd' the values are 1 for the developing countries and are 0 for the developed countries. From the estimation results in the output file `dummy.out` we should be able to tell whether the consumption difference between the developed countries and the developing countries is statistically significant based on the coefficient estimates for the dummy variable.

The above analysis of the dummy variable is the so-called constant-slope approach since the slope coefficient β are the same for both types of countries. We can also consider the varying-slope approach, where both the intercept α and the slope β of the Engel curve model are allowed to be different between the two types of countries. The key to this new approach is the inclusion of the interaction term, which is the product of the log income and the dummy variable:

$$s_{ic} = \alpha + \beta \ln m_c + \gamma d_c + \delta (\ln m_c \times d_c) + \varepsilon_{ic}, \qquad c = 1, 2, \ldots, 60.$$

In this general Engel curve model the matrix $\mathbf{X}$ includes 4 columns: the constant term, the log income, the dummy variable, and the interaction term. The vector $\boldsymbol{\beta}$ contains 4 regression coefficients.

$$\mathbf{X} = \begin{bmatrix} 1 & \ln m_1 & d_1 & (\ln m_1 \times d_1) \\ 1 & \ln m_2 & d_2 & (\ln m_2 \times d_2) \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \ln m_{60} & d_{60} & (\ln m_{60} \times d_{60}) \end{bmatrix} \quad \text{and} \quad \boldsymbol{\beta} = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}.$$

The GAUSS program for estimating this model is

```
new;

```

```
load s[60,10] = share;
load m[60,1] = totalexp;
load c[60,2] = country;

d = (c[.,2] ./= "NA" .and c[.,2] ./= "EU");

y = s[.,1];
x = ones(60,1)~ln(m)~d~(ln(m).*d);

output file = dummy.out on;
__altnam = "Cnst"|"Ln_m"|"Dummy"|"Interact"|"Share";
call ols(0,y,x);

end;
```

Note that the construction of the interaction term '`d.*ln(m)`' in the data matrix '`x`' is quite easy because of element-by-element multiplication.

6.5.2 The Polychotomous Case

Variables like '`ctt`' and '`marital`' are referred to as polychotomous categorical variables because they have more than two possible values. The way we have handled them is to construct dummy variables from them. However, information in these variables is usually more than what a single dummy variable can convey. To fully reflect the information in a polychotomous categorical variable, we need to construct a set of dummy variables. Furthermore, it is sometimes possible to derive a set of dummy variables from "continuous" variables like '`year`,' '`school`,' and '`child_no`'.

Generalizing from the two approaches in the single dummy variable case, we have the following two approaches to including multiple dummy variables into a simple linear regression model:

1. *The constant-slope case*: It is very important to know that only $J-1$ dummy variables are needed for the J categories when the intercept term is included in the linear regression model. The specification of the model is as follows:

$$y_i = \alpha + \beta x_i + \sum_{j=1}^{J-1} \gamma_j d_{ji} + \varepsilon_i,$$

$$= \begin{cases} (\alpha + \gamma_j) + \beta x_i + \varepsilon_i, & \text{if } d_{ji} = 1 \text{ and } d_{ki} = 0, \text{ for all } k \neq j, \text{ and } j = 1, \ldots, J-1, \\ \alpha + \beta x_i + \varepsilon_i, & \text{if } d_{ji} = 0, \text{ for all } j, \end{cases}$$

where the category with all $d_{ji} = 0$ is considered as the *base*. γ_j indicates the difference in the intercept between the base category and the category specified by $d_{ji} = 1$ and $d_{ki} = 0$, for all $k \neq j$.

Consider the example of the four-category '`marital`' variable. three dummy variables are needed to carry all the information in '`marital`' and they can be defined in GAUSS as follows:

```
m_dum1 = (marital .== 1);
m_dum2 = (marital .== 2);
m_dum3 = (marital .== 3);
```

Here, we note a variable 'm_dum0' which may be defined by '(marital .== 0)' is deliberately missed because the category of the 'Never Married' is chosen as the base. For the 'Never Married' the values of the three dummy variables 'm_dum1', 'm_dum2', and 'm_dum3' are all 0. As to the 'Married,' the value of the dummy variable 'm_dum1' is 1 and the other two are 0. The specifications for other categories are similar to that of the 'Married.'

It is possible to use a single GAUSS statement to define an $n \times 3$ matrix that includes all four dummy variables 'm_dum1', 'm_dum2', and 'm_dum3':

```
m_dum = (marital .== (1~2~3));
```

The resulting matrix 'm_dum' is equivalent to the horizontal concatenation of the four columns of 'm_dum1', 'm_dum2', and 'm_dum3' in the previous definition.

Let's now consider another example of generating multiple dummy variables from the *continuous* variable 'school'. Suppose we want to create four dummy variables for the five categories: (1) some high school education; (2) high school graduates; (3) some college education; (4) college graduates; and (5) some postgraduate education; where the second category of high school graduates is taken as the base. The GAUSS statements are

```
ed_dum1 = (school .<= 11);
ed_dum3 = ((12 .< school) .and (school .<= 15));
ed_dum4 = ((15 .< school) .and (school .<= 16));
ed_dum5 = (16 .< school);
```

There seems no simple relational operations to create an $n \times 4$ matrix that contains all four dummy variables at one stroke. But there is a GAUSS command designed specifically for this purpose:

```
y = dummydn(x,e,j);
```

What this command does is following:

(1) Using elements e_1, e_2, . . ., of the second input 'e', which must be a $k \times 1$ column of numbers in ascending order, to partition the real line into $k+1$ open-closed intervals: $(-\infty, \ e_1]$, $(e_1, \ e_2]$, . . . $(e_{k-1}, \ e_k]$, $(e_{k-1}, \ \infty]$.

(2) Evaluating each element in the first input 'x', which is an $n \times 1$ column, to determine which of the above $k+1$ open-closed intervals this element belongs to;

(3) Creating an $n \times (k+1)$ temporary matrix, say 'z', of dummy variables, each row contains a 1 and k 0's. Among the $k+1$ element of the i-th row of 'z', which element is 1 depends on which of the above $k+1$ open-closed intervals contains the value of the i-th element of 'x'.

(4) The third input 'j', which must be an integer between 1 and $k+1$, specifies the column to be deleted from the $n \times (k+1)$ matrix 'z' in order to produce the $n \times k$ output matrix 'y'. The deleted column corresponds to the category that is designated as the base.

So in our example we should type

```
e = 11|12|15|16;
ed_dum = dummydn(school,e,2);
```

The resulting $n \times 4$ matrix 'ed_dum' is equivalent to the horizontal concatenation of the four columns 'ed_dum1', 'ed_dum3', 'ed_dum4', and 'ed_dum5' in the earlier definition. Obviously, with the four break points in the second input 'e' it is possible to create five categories from the input 'school', while the third input of the 'dummydn' command, which is 2 in our example, indicates which among the five categories is designated as the base so that the corresponding column will be deleted.

2. *The varying-slope case*: Again, $J-1$ dummy variables are needed for J categories and the model becomes

$$y_i = \alpha + \beta x_i + \sum_{j=1}^{J-1} \gamma_j d_{ji} + \sum_{j=1}^{J-1} \delta_j x_i d_{ji} + \varepsilon_i,$$

$$= \begin{cases} (\alpha+\gamma_j) + (\beta+\delta_j)x_i + \varepsilon_i, & \text{if } d_{ji} = 1 \text{ and } d_{ki} = 0, \text{ for all } k \neq j, \text{ and } j = 1, \ldots, J-1, \\ \alpha + \beta x_i + \varepsilon_i, & \text{if } d_{ji} = 0, \text{ for all } j, \end{cases}$$

where the category with all $d_{ji} = 0$ is considered the base. γ_j indicates the difference in the intercept and δ_j indicates the difference in the slope between the base category and the category with $d_{ji} = 1$ and $d_{ki} = 0$, for all $k \neq j$.

In the previous example about the variable 'school' we have created an $n \times 4$ matrix 'ed_dum' of dummy variables. Now suppose we have a column of data 'x0' on the single continuous explanatory variable x_i, then the new data matrix 'x' of all explanatory variables for the present varying-slope case is formed by

```
x = ones(n,1)~x0~ed_dum~(x0.*ed_dum);
```

The dimension of the matrix 'x' is $n \times 10$.

As demonstrated by the example of generating multiple dummy variables from the *continuous* variable 'school', we note essentially any continuous variable can be transformed to a set of dummy variables. Doing so may be necessary in some applications because we are not always certain about whether the

regression coefficients of those continuous explanatory variables are constant over their entire ranges. In the next subsection, we will explore another approach to the problem of non-constant regression coefficients which is based on another kind of categorization of the information from the continuous variables.

An Example Extending from the previous simple dummy variable analysis for the International Consumption data, we now study how the consumption behavior differs among the developed countries and the three groups – the African, Asian, and South American countries – of the developing countries. That is, we are going to divide the 60 sampled countries into four categories. For four categories we need three dummy variables, says, d_1, d_2, d_3, whose exact definitions will depend on the choice of the base category. Here, we arbitrarily choose the developed countries as the base. The specific definitions of the dummy variables are as follows: For the developed countries the values of the three dummy variables are all zeros. The values of d_1, d_2, and d_3 for the African countries are 1, 0, and 0, respectively. For the Asian countries, the values are 0, 1, and 0, respectively; and for the South American countries, the values are 0, 0, and 1, respectively.

We include the three dummy variables in the Engel curve model as follows:

$$s_{ic} = \alpha + \beta \ln m_c + \gamma_1 d_{1c} + \gamma_2 d_{2c} + \gamma_3 d_{3c} + \varepsilon_{ic}, \qquad c = 1, 2, \ldots, 60.$$

In writing a GAUSS program for estimating this extended Engel curve model, the key is again the construction of the dummy variable.

```
new;

load s[60,10] = share;
load m[60,1] = totalexp;
load c[60,2] = country;

d = (c[.,2] .== "AF"~"AS"~"SA");

y = s[.,1];
x = ones(60,1)~ln(m)~d;

output file = dummy.out reset;
__altnam = "Cnst"|"Ln_m"|"Africa"|"Asia"|"S._Amer."|"Share";
call ols(0,y,x);

end;
```

We should note that 'd' here is not a column but a 60×3 matrix containing the 60 observations for the three dummy variables d_1, d_2, and d_3.

From the estimation results in the output file `dummy.out` we should be able to test the significance of the consumption difference between the base countries (i.e., the developed countries) and each group of the developing countries, based on the coefficient estimates for the three dummy variables. However, to infer the consumption difference between, says, African countries and Asian countries, we must get the difference between the coefficient estimates of d_1 and d_2. That is, the coefficient estimates for the three dummy variables only show the differences between the base countries and each group of developing countries.

A more difficult question is about how to test the significance of consumption difference between African countries and Asian countries. Subtracting the coefficient estimates of the corresponding dummy variables only gives us the coefficient difference. We need a standard error for this difference in order to conduct the significance test. Obviously, additional computation is needed to answer the question. However, a short-cut answer can be come by given that we are able to do the OLS estimation easily. To find out whether African countries are significantly different from Asian countries, we can simply change the base category from the developed countries to the African countries, redefine the three dummy variables, and then rerun the program. The coefficient estimates from this new setup can then answer our question directly.

Let's now turn to the varying-slope case for four categories of countries, we need three additional interaction terms:

$$s_{ic} = \alpha + \beta \ln m_c + \gamma_1 d_{1c} + \gamma_2 d_{2c} + \gamma_3 d_{3c} + \delta_1 (\ln m)(d_{1c}) + \delta_2 (\ln m)(d_{2c}) + \delta_3 (\ln m)(d_{3c}) + \varepsilon_{ic}, \qquad c = 1, 2, \ldots, 60.$$

In this general Engel curve model the matrix $\mathbf{X}$ includes 8 columns: the constant term, the log income, the three dummy variables, and the three interaction terms. The vector $\boldsymbol{\beta}$ contains 8 regression coefficients. The GAUSS program for estimating such a model is

```
new;

load s[60,10] = share;
load m[60,1] = totalexp;
load c[60,2] = country;

d = (c[.,2] .== "AF"~"AS"~"SA");

y = s[.,1];
x = ones(60,1)~ln(m)~d~(ln(m).*d);

output file = dummy.out on;

__altnam = "Cnst"|"Ln_m"|"Africa"|"Asia"|"S._Amer."|
           "AFxLn_m"|"ASxLn_m"|"SAxLn_m"|"Share";
call ols(0,y,x);

end;
```

The interaction term '`d.*ln(m)`' is a 60×3 matrix after element-by-element multiplication.

6.5.3 The Piecewise Linear Regression Model

In the piecewise linear regression model, we assume the regression line can be broken into $J + 1$ pieces at the J break points $x = c_j$, $j = 1, \ldots, J$, where $c_1 < c_2 < \ldots < c_J$. That is, the slope of the regression line is not constant but changes at those J break points.

To construct a piecewise linear regression model, we need to define J dummy variables as follows:

$$d_{ji} = \begin{cases} 1, & \text{for } j \le k, \\ 0, & \text{for } j > k, \end{cases} \qquad \text{if} \quad c_k < x_i \le c_{k+1}, \quad \text{for} \quad k = 0, \ldots, J,$$

where $c_0 = -\infty$ and $c_{J+1} = \infty$.

Given a column vector of data 'x0' on the single explanatory variable x_i and a column vector 'c' of J break points, the $n \times J$ matrix containing all J dummy variables $d_{1i}, d_{2i}, \ldots, d_{Ji}$ can be created quite easily in GAUSS as follows:

```
dum = (x0 .>= c');
```

Here, if the i-th element of the vector 'x0' is greater than the j-th element but not greater than the $j+1$-th element of the vector 'c' , then the first j elements in the i-th row of the matrix 'dum' are all 1's and the last $J - j$ elements are all 0's.

1. If we assume the piecewise regression line is continuous, then

$$y_i = \alpha + \beta x_i + \sum_{j=1}^{J} \gamma_j (x_i - c_j) d_{ji} + \varepsilon_i,$$

$$= \begin{cases} \alpha + \beta x_i + \varepsilon_i, & \text{if } x_i \le c_1, \\ \left(\alpha - \sum_{j=1}^{k} \gamma_j c_j\right) + \left(\beta + \sum_{j=1}^{k} \gamma_j\right) x_i + \varepsilon_i, & \text{if } c_k < x_i \le c_{k+1}, \text{ for } k = 1, \ldots, J. \end{cases}$$

   Given the matrix 'dum' containing the J dummy variables $d_{1i}, d_{2i}, \ldots, d_{Ji}$, the matrix 'x' of all explanatory variables for the present case is defined to be:

```
x = ones(n,1)~x0~((x0 - c').*dum);
```

   which is an $n \times (J+2)$ matrix.

2. If we allow the piecewise regression line to be disconnected at those J break points, then

$$y_i = \alpha + \beta x_i + \sum_{j=1}^{J} \gamma_j (x_i - c_j) d_{ji} + \sum_{j=1}^{J} \delta_j d_{ji} + \varepsilon_i,$$

$$= \begin{cases} \alpha + \beta x_i + \varepsilon_i, & \text{if } x_i \le c_1, \\ \left(\alpha - \sum_{j=1}^{k} \gamma_j c_j + \sum_{j=1}^{k} \delta_j\right) + \left(\beta + \sum_{j=1}^{k} \gamma_j\right) x_i + \varepsilon_i, & \text{if } c_k < x_i \le c_{k+1}, \text{ for } k = 1, \ldots, J, \end{cases}$$

   where the parameter δ_j indicates the distance between the adjacent two pieces at the break point $x = c_j$. Given the matrix 'dum', the matrix 'x' of all explanatory variables for the present case is

```
x = ones(n,1)~x0~((x0 - c').*dum)~dum;
```

which is an $n \times (2J + 2)$ matrix.

Chapter 7

Iteration with Do-Loops

When we implement econometric methods in GAUSS programming, a very common situation is that we need to repeat similar computation with almost identical sets of GAUSS commands. The process of iterating the same set of commands can be done by a mechanism called do-loop in GAUSS. We will explain the concept of a do-loop through a simple example.

7.1 Do-loops

Consider the product of two column-vectors, says, 'a' and 'b', both with 5 elements. It is defined to be the sum of the five products of the corresponding elements in 'a' and 'b'. Suppose we want to use this definitional formula to compute the product 'c' of two vectors, then the most straightforward way to do it is

```
c = a[1]*b[1] + a[2]*b[2] + a[3]*b[3] + a[4]*b[4] + a[5]*b[5];
```

This computation involves adding terms repeatedly with a sequential indices running from 1 to 5. This kind of arithmetic can always be written in a *recursive* fashion as follows:

```
c = 0;
c = c + a[1]*b[1];
c = c + a[2]*b[2];
c = c + a[3]*b[3];
c = c + a[4]*b[4];
c = c + a[5]*b[5];
```

We note that the last five expressions are identical except the indices running from 1 to 5. In GAUSS any group of recursive expressions can be replaced by a do-loop.

A do-loop starts with the command 'do until', followed by a condition (the so-called *do-loop condition*), then a set of commands to be repeated, and then ends with the 'endo' command. The GAUSS statements between the 'do until' command and the 'endo' command will be executed repeatedly. After each iteration the do-loop condition will be checked to decide whether to continue another round of iteration or to stop. Iteration will be continued as long as the do-loop condition *is not* satisfied. For the previous example, we can use the following do-loop representation:

```
c = 0;
i = 1;
do until i > 5;
   c = c + a[i]*b[i];
```

```
    i = i + 1;
endo;
```

There is another kind of do-loops that start with the command 'do while'. Though the basic structure is the same as that of the 'do until' command, the GAUSS statements between the 'do while' command and the 'endo' command will be executed repeatedly as long as the do-loop condition *is* still satisfied. With this command, the previous example can be written as

```
c = 0;
i = 1;
do while i <= 5;
   c = c + a[i]*b[i];
   i = i + 1;
endo;
```

Generally speaking, there are two kinds of do-loop conditions: one involves an index, such as 'i' in the previous examples, whose value has to be initiated before the do-loop and will change sequentially inside the do-loop. Iteration will stop when the index attains a prescribed value. The second type of do-loop conditions involve the computation result directly from each iteration, in which case iteration will stop when the computation result satisfies a prescribed criterion. For example, the iteration terminates when a result inside the do-loop becomes smaller than 0.00001.

Do-loops can be nested; i.e., a do-loop can be placed inside another do-loop. Let's consider the example of multiplying a matrix with a vector: $\mathbf{Ab} = \mathbf{c}$, where $\mathbf{A}$ is an $n \times m$ matrix, $\mathbf{b}$ an $m \times 1$ column vector, and the result $\mathbf{c}$ is an $n \times 1$ column vector. If we want to conduct this computation with do-loops, then nested do-loops are needed. Other than computing the sum of m products for each row of $\mathbf{A}$ which requires one do-loop (the inner do-loop), we need another layer of do-loop (the outer do-loop) for the n rows of $\mathbf{A}$. Given the matrix 'a' and the vector 'b', we compute the vector 'c' as follows:

```
c = zeros(n,1);
i = 1;
do until i > n;
   j = 1;
   do until j > m;
      c[i] = c[i] + a[i,j]*b[j];
      j = j + 1;
   endo;
   i = i + 1;
endo;
```

It is important to note that in the very first statement we define the dimension of the 'c' vector to "reserve" the space for the iteration results. Such a technique is needed whenever we need to retain results from each iteration. An alternative way of retaining results from each iteration, without the knowledge of how much space to be reserved, is as follows:

```
c = 0;
i = 1;
do until i > n;
   j = 1;
   temp = 0;
   do until j > m;
      temp = temp + a[i,j]*b[j];
      j = j + 1;
   endo;
   c = c|temp;
   i = i + 1;
endo;
c = c[2:rows(c)];
```

Here, a scalar 'c' is initiated with the value 0 right before the do-loop. We use this 'c' as a base and we will attach the result form each iteration to it. Each time the inner do-loop for computing the sum of m products is completed, the result is attached to 'c'. After all n computations are finished, we have a $(n+1) \times 1$ column vector 'c' whose first element needs to be deleted as is done in the statement right after the 'endo' command. Note that if the result from each iteration is a vector (instead of a scalar as in the above example), then a vector has to be initiated before the do-loop so that the resulting vector from each iteration can be properly attached to this initial vector.

A General Principle in Using Do-loop Do-loop is a useful tool in many situations. As a matter of fact, all the matrix operations, such as matrix multiplication in the previous examples, can be replaced by do-loops. However, in most matrix operations we should use the matrix operators instead of do-loops. This is because matrix operators are easier to use and are executed with much more efficiency. A general principle for GAUSS programming is that *the use of do-loops should be avoided unless there is absolutely no alternative*. Replacing do-loops with ingenious matrix operations, if possible, always save us tremendous amount of computing time.

An Example Our previous program for the demand estimation can deal with one commodity only. We need to rerun that program ten times for the ten commodities of the International Consumption Data. Here, by using do-loops we can easily conduct the estimations for all ten commodities in one program. In such a program it is then straightforward to collect all the regression coefficient estimates and perform additional computations with them. In particular, we can compute income elasticities for all ten commodities and for all 60 countries in one stroke. We can also check whether the sum of the ten intercept estimates is equal to one and whether the sums of the ten sets of all other coefficient estimates are equal to zero.

```
new;

load q[60,10] = volume;
load s[60,10] = share;
load m[60,1] = totalexp;
```

```
6
7   p = (s.*m)./q;
8   x = ln(m)~ln(p);
9
10  output file = all.out reset;
11
12  __altnam = "Cnst"|"Ln_m"|"Food"|"Bev_tob"|"Clothing"|"Rent"|"HouseFur"|
13             "Medicare"|"Transpt"|"Recreatn"|"Educatn"|"Other"|"Food_S";
14
15  bout = zeros(1,12);     /*  Initializing a row vector to which all the
16                              regression coefficient estimates from the
17                              do-loop can attach.                          */
18
19  eout = zeros(60,1);     /*  Initializing a column vector to which all
20                              the elasticities estimates from the do-loop
21                              can attach.                                  */
22
23  i = 1;                  /*  Initializing an index for counting.          */
24  do until i > 10;        /*  Do-loop will run ten times.                  */
25
26     y = s[.,i];          /*  The index i picks one commodity in turn.     */
27     {vnam,mmt,b,stb,vb,seb,s2,cor,r2,e,d} = ols(0,y,x);
28     eta = 1 + b[2]./s[.,i];    /*  The computation of the elasticities.   */
29     bout = bout|b';  /*  Collecting the regression estimates row by row.  */
30     eout = eout~eta; /*  Collecting the elasticity estimates column by
31                         column.                                           */
32     i = i + 1;
33  endo;
34  bout = bout[2:rows(bout),.];        /*  Deleting unwanted first row.     */
35  eout = eout[.,2:cols(eout)];        /*  Deleting unwanted first column.  */
36
37  eout = seqa(1,1,60)~eout;           /*  Attaching a column of counters.  */
38
39  " The income elasticities for the ten commodities and 60 countries:";?;
40  call printfm(eout,ones(1,11),"*.*lf"~4~0|(ones(10,1).*("*.*lf"~6~3));?;?;
41
42  " The sum of ten sets of regression coefficient estimates:";?;
43  format /rd 15,8; sumc(bout);
44
45  end;
```

Note that after the ‘eout’ matrix is generated toward the end of the program, a column of 60 consecutive numbers 1,2,. . .,60, which serve as counters are generated and attached to the ‘eout’ matrix to facilitate its reading. We then print these counters in the first column of ‘eout’ as integers while the rest 60×10 matrix of elasticity estimates as real numbers. Because different columns of ‘eout’ are to be printed differently, we

must use a 'printfm' command, in which the second argument, a row of 11 ones, indicates all 11 columns of 'eout' are numbers (instead of characters). The third argument of the 'printfm' command is a 11×3 formatting matrix. The first row of the formatting matrix, '*.*lf~4~0', is associated with the first column of 'eout', which contains the counters, and the other 10 rows are all identically equal to '(*.*lf~6~3)' since the last ten columns of 'eout' are to be printed in the same format. Note that the way we construct the last ten rows of the formatting matrix is based on an element-by-element trick for duplication. This trick can be quite useful in many occasions.

Breaking Away from Do-loop Sometimes we need to break away from the do-loop before the do-loop condition is satisfied by using a supplementary GAUSS statement 'break'. For a trivial example, if we only want to add the first three product terms from the original 5-iteration do-loop condition, then we put an 'if' statement inside the loop as follows:

```
c = 0;
i = 1;
do until i > 5;
   if i > 3;
      break;
   endif;
   c = c + a[i]*b[i];
   i = i + 1;
endo;
```

When the 'if' condition is satisfied (i.e., after three iterations are completed and the index 'i' becomes 4), then the 'break' statement is executed and the program will jump out of the loop and go to the statement immediately after the 'endo' statement.

There is another supplementary GAUSS statement – the 'continue' statement – that help bypass some of the statements inside the do-loop but still continue the do-loop iteration as illustrated by the following example in which the third product term is to be skipped from the summation:

```
c = 0;
i = 1;
do until i > 5;
   if i == 3;
      continue;
   endif;
   c = c + a[i]*b[i];
   i = i + 1;
endo;
```

When the 'if' condition is satisfied (i.e., after two iterations are completed and the index 'i' becomes 3) so that the 'continue' statement is executed, the program will skip the rest of the statements inside the loop and jump to the top of the loop to evaluate the do-loop condition to decide whether to continue the iteration. The above program is equivalent to

```
c = a[1]*b[1] + a[2]*b[2] + a[4]*b[4] + a[5]*b[5];
```

where the third product is omitted.

7.2 Some Statistics Related to the OLS Estimation

In this section we briefly introduce a few estimators and test statistics that are used to deal with some common problems with the OLS Estimation when data cannot really satisfy the standard assumptions required by the linear regression model. We then present GAUSS programs for those estimators and test statistics. In particular, we demonstrate how do-loops are necessary in many of these programs.

7.2.1 The Heteroscedasticity Problem

Heteroscedasticity is a problem with a linear regression model $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$ in that $\text{Var}(y_i) = \text{Var}(\varepsilon_i) = \sigma_i^2$ are not constant across i so that

$$\text{Var}(\boldsymbol{\varepsilon}) = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n^2 \end{bmatrix} \equiv \boldsymbol{\Sigma}.$$

Heteroscedasticity usually occurs to the cross-section data.

White's Estimator: The main problem with the OLS estimator $\mathbf{b}$ under heteroscedasticity is that its variance-covariance matrix becomes more complicated:

$$\text{Var}(\mathbf{b}) = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\boldsymbol{\Sigma}\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1} = (\mathbf{X}'\mathbf{X})^{-1}\left(\sum_{i=1}^{n}\sigma_i^2\mathbf{x}_i\mathbf{x}_i'\right)(\mathbf{X}'\mathbf{X})^{-1},$$

where $\mathbf{x}_i'$ is the ith row of the $n \times k$ matrix $\mathbf{X}$. However, the above expression can be consistently estimated by the so-called White's heteroscedasticity-consistent estimator:

$$\widehat{\text{Var}}(\mathbf{b}) = (\mathbf{X}'\mathbf{X})^{-1}\left(\sum_{i=1}^{n}e_i^2\mathbf{x}_i\mathbf{x}_i'\right)(\mathbf{X}'\mathbf{X})^{-1},$$

where e_i are the OLS residuals.

Given the $n \times k$ data matrix 'x' of the explanatory variables and the n-vector of OLS residuals 'e', White's heteroscedasticity-consistent estimator can be calculated by the GAUSS statement:

```
w_vb = invpd(x'x)*x'diagrv(eye(n),e)*x*invpd(x'x);
```

where the GAUSS command 'diagrv' takes two inputs – the first input is a square matrix and the second one is a column vector – and produces a square matrix that is the same as the first input except the diagonal elements are those of the second input. Recall that the GAUSS command 'eye(n)' gives an $n \times n$ identity matrix.

It is easy to verify that, given any $n \times 1$ column vector 'a' and $n \times n$ matrix 'b', the expression 'a.*b' yields the same result as that of 'diagrv(eye(n),a)*b'. Consequently, White's heteroscedasticity-consistent estimator can be calculated by the following alternative GAUSS program:

```
w_vb = invpd(x'x)*x'(e.*x)*invpd(x'x);
```

White's Test for Heteroscedasticity Given R^2 of the special regression of the squared OLS residuals e_i^2 on all the explanatory variables and their squared and cross product terms, White's Test is based on the statistic

$$S_w = n \cdot R^2,$$

where n is the sample size. Suppose the number of the explanatory variables in such a regression is $k_\circ$, then the test statistic S_w has a $\chi^2(k_\circ - 2)$ distribution under the null hypothesis of *no* heteroscedasticity.

The GAUSS program for calculating White's test statistic is somewhat complicated. Given the sample size 'n', the $n \times k$ data matrix 'x' of the explanatory variables including the constant term, and the n-vector of OLS residuals 'e', we use

```
j = 1;
do until j > k;
   x0 = x0~(x[.,j].*x[.,j:k]);
   j = j + 1;
endo;

y0 = e^2;
e0 = y0 - x0*invpd(x0'x0)*x0'y0;
tss0 = (y0 - meanc(y0))'(y0 - meanc(y0));
rss0 = e0'e0;
r2 = (tss0 - rss0)/tss0;

s_w = n*r2;
```

Here, a do-loop is necessary in the construction of the 'x0' matrix which contains the data on all the explanatory variables in 'x' and their squared and cross product terms. The five statements after the do-loop represent the standard procedure to compute R^2 where the particular dependent variable 'y0' in this computation is the squared OLS residual e_i^2. White's test statistic is in 's_w'. If the computed value of White's test statistic is greater than the $(1-\alpha)$% critical point from the $\chi^2(k_\circ - 2)$ distribution, where $k_\circ$ is the column number of 'x0', then the null hypothesis of *no* heteroscedasticity will be rejected.

Breusch-Pagan's Test for Heteroscedasticity If heteroscedasticity follows the form

$$\sigma_i^2 = \sigma^2 \cdot h(\mathbf{z}_i'\boldsymbol{\gamma}),$$

for some known function h of some vector $\mathbf{z}_i$ of p observable variables which includes the constant term and can overlap with $\mathbf{x}_i$, then Breusch-Pagan's test statistic is

$$S_{BP} = \frac{(\mathbf{w} - \mathbf{1}_n)'\mathbf{Z}(\mathbf{Z}'\mathbf{Z})^{-1}\mathbf{Z}'(\mathbf{w} - \mathbf{1}_n)}{(\mathbf{w} - \mathbf{1}_n)'(\mathbf{w} - \mathbf{1}_n)/n}.$$

where $\mathbf{1}_n$ is n dimensional vector of ones, $\mathbf{w}$ and $\mathbf{Z}$ are the $n \times 1$ vector and the $n \times p$ matrix containing all the observations of w_i and $\mathbf{z}_i$, respectively. Here, w_i are normalized squared residuals

$$w_i = \frac{e_i^2}{\frac{1}{n}\sum_{j=1}^{n} e_j^2}.$$

S_{BP} has the $\chi^2(p-1)$ distribution under the null hypothesis and the normality assumption.

Given the $n \times k$ data matrix '`x`' of the explanatory variables, the $n \times p$ data matrix '`z`' of the variables $\mathbf{z}_i$, and the n-vector of OLS residuals '`e`', the GAUSS program for calculating Breusch-Pagan's test statistic is

```
w = e^2./meanc(e^2);
s_bp = (w - 1)'z*invpd(z'z)*z'(w - 1)/((w - 1)'(w - 1)/n);
```

We note that although '`w`' is an n-dimensional vector, we can still subtract a scalar 1 from it since GAUSS will automatically expand 1 to an n-vector of ones which becomes conformable to '`w`'.

7.2.2 The Autocorrelation Problem

In a linear regression model for time-series data (here, we change the subscript from i to t to emphasize the nature of data is time series):

$$y_t = \mathbf{x}_t'\boldsymbol{\beta} + \varepsilon_t, \quad t = 1, \ldots, n,$$

we usually expect nonzero covariances of the disturbances across time

$$\mathrm{Cov}(y_s, y_t) = \mathrm{Cov}(\varepsilon_s, \varepsilon_t) \equiv c_{st} \neq 0, \qquad \text{for some } s,\ t = 1, 2, \ldots, n.$$

Because of these nonzero covariances the OLS estimation has the problem that the variance of the OLS estimator $\mathbf{b}$ changes from the formula $\sigma^2(\mathbf{X}'\mathbf{X})^{-1}$ to

$$\mathrm{Var}(\mathbf{b}) = (\mathbf{X}'\mathbf{X})^{-1}\left(\sum_{s=1}^{n}\sum_{t=1}^{n} c_{st}\mathbf{x}_s\mathbf{x}_t'\right)(\mathbf{X}'\mathbf{X})^{-1},$$

where $\mathbf{x}_t'$ is the tth row of the $n \times k$ matrix $\mathbf{X}$.

Newey-West's Estimator: Similar to White's estimator of Var(**b**) which is used in the presence of heteroscedasticity of an unknown form, we use the follwoing Newey-West's estimator[1] for Var(**b**) when we believe covariances of the disturbances across time are not zero and do not have any specific forms:

$$\widehat{\text{Var}}(\mathbf{b}) = (\mathbf{X}'\mathbf{X})^{-1}\left[\sum_{t=1}^{n} e_t^2\mathbf{x}_t\mathbf{x}_t' + \sum_{j=1}^{m}\sum_{t=j+1}^{n} w_j e_t e_{t-j}(\mathbf{x}_t\mathbf{x}_{t-j}' + \mathbf{x}_{t-j}\mathbf{x}_t')\right](\mathbf{X}'\mathbf{X})^{-1},$$

where w_j are weights defined by $w_j = j/(m+1)$, $j = 1, 2, \ldots, m$, for a given number m, which is usually a small integer but can be increased as the sample size n increases.

Given an integer '`m`' (which represents the number "m" in the above definition), the $n \times k$ data matrix '`x`' of the explanatory variables, and the n-vector of OLS residuals '`e`', the GAUSS program for calculating Newey-West's estimator is

```
nw = 0;
j = 1;
do until j > m;
   t = j + 1;
   do until t > n;
      nw = nw + ((j/(m+1))*e[t]*e[t-j]).*(x[t,.]'x[t-j,.] + x[t-j,.]'x[t,.]);
      t = t + 1;
   endo;
   j = j + 1;
endo;

nw =  invpd(x'x)*(x'(e.*x)+ nw)*invpd(x'x);
```

We should note that the particular matrix indexing '`x[t,.]`' gives the tth row of the data matrix '`x`'; i.e., it yields the row vector $\mathbf{x}_t'$.

We now consider some tests for the null hypothesis of no autocorrelation against some unspecified alternative forms of autocorrelation.

Breusch-Godfrey's Test for Autocorrelation: Given R^2 of the special regression of e_t on $\mathbf{x}_t$, e_{t-1}, ..., e_{t-m} for some integer m, Breusch-Godfrey's test statistic is

$$S_{\text{bg}} = n \cdot \text{R}^2.$$

The test statistic has a χ^2 distribution with m degree of freedom under the null hypothesis of no autocorrelation.[2]

---

[1] To ensure the consistency of Newey-West's estimator we have to make certain assumptions about those covariances c_{st}. One particularlly important assumption is as follows:

$$c_{st} \longrightarrow 0, \qquad \text{as } |t-s| \longrightarrow \infty,$$

which means the covariance between ε_s and ε_t becomes increasingly smaller as the difference between the two time periods s and t gets larger.

[2] The alternative hypothesis Breusch-Godfrey's test considers is in fact autocorrelated disturbances of either AR(m) or MA(m) form.

Given the sample size ‘n’, the $n \times k$ data matrix ‘x’ of the explanatory variables, and the n-vector of OLS residuals ‘e’, the GAUSS program for calculating Breusch-Godfrey’s test statistic is

```
x0 = x~shiftr(ones(m,1)*e',seqa(1,m),0)';
y0 = e;

e0 = y0 - x0*invpd(x0'x0)*x0'y0;
tss0 = (y0 - meanc(y0))'(y0 - meanc(y0));
rss0 = e0'e0;
r2 = (tss0 - rss0)/tss0;

s_w = n*r2;
```

The most interesting part of the program is in the first line where the $n \times (k + m)$ matrix of explanatory variables is constructed. There we use a new GAUSS command ‘shiftr’ which takes three inputs: the first input is an $m \times n$ matrix, say, ‘a’, the second input is an $m \times 1$ column vector, say, ‘b’, and the third input is either an $m \times 1$ column vector or a scalar, say, ‘c’. What the command ‘shiftr’ does is to shift the rows of the matrix ‘a’ horizontally. The number of steps shifted in each row is determined by the value in the corresponding element in the vector ‘b’. If the number is positive, then the shift is to the right. Otherwise, the shift is to the left. For example, if the third element in ‘b’ is 5, then the third row of ‘a’ is shifted five steps to the right so that the last five numbers in that row will be lost while the first five numbers are all the number in the third element of the third input vector ‘c’. Going back to the above program, we see that the first input for the command ‘shiftr’ is an $m \times n$ matrix with the identical row ‘e’ and the second input is a column vector of sequential integers from 1 to m. The output of that command is an $m \times n$ matrix as follows:

$$\begin{bmatrix} 0 & e_1 & e_2 & e_2 & \cdots & e_m & e_{m+1} & e_{m+2} & \cdots & e_{n-1} \\ 0 & 0 & e_1 & e_2 & \cdots & e_{m-1} & e_m & e_{m+1} & \cdots & e_{n-2} \\ 0 & 0 & 0 & e_1 & \cdots & e_{m-2} & e_{m-1} & e_m & \cdots & e_{n-3} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & 0 & \cdots & e_1 & e_2 & e_2 & \cdots & e_{n-m} \end{bmatrix}.$$

The first row is shifted to the right by one step (so that the last element e_n is pushed off) while the mth row is shifted to the right by m steps. All the spaces left are filled with zero (which is specfied in the third input of the ‘shiftr’ command.

Finally, we note that the output of the command ‘shiftr’ is transposed to an $n \times m$ matrix before concatenated horizontally to the $n \times k$ matrix ‘x’. It should be emphasized that the operation of transpose has higher priority than the operation of concatenation so that it is not necessary to add parentheses around ‘shiftr(ones(m,1)*e',seqa(1,m),0)'’.

Q Tests for Autocorrelation: Given the jth order sample autocorrelations from the OLS residuals:

$$\hat{\rho}_j = \frac{\sum_{t=j+1}^{n} e_t e_{t-j}}{\sum_{t=1}^{n} e_t^2},$$

we have the following two test statistics for a given integer m:

1. Box-Pierce Q Statistic:
$$Q_1 = n \sum_{j=1}^{m} \hat{\rho}_j^2.$$

2. Ljung-Box Q Statistic:
$$Q_2 = n(n+2) \sum_{j=1}^{m} \frac{\hat{\rho}_j^2}{n-j}.$$

Both statistics have a χ^2 distribution with m degree of freedom under the null hypothesis of no autocorrelation.

Given the sample size 'n', an integer 'm', and the n-vector of OLS residuals 'e', the GAUSS program for calculating Q test statistics is

```
rho = (shiftr(ones(m,1)*e',seqa(1,m),0)*e)./(e'e);
q1 = n*(rho'rho);
q2 = n*(n + 2)*(rho'(rho./(n - seqa(1,m))));
```

7.2.3 Structural Stability

When we use Chow tests for structural changes, one basic assumption is that we know exactly when the structural changes occur so that we can divide the sample accordingly. But in many applications using *time-series data* we may not know whether there are structural changes and, if there are, when they occur. In other words we are not certain about the structural stability, or the constancy of the regression coefficients $\boldsymbol{\beta}$, over time. To test structural stability of a linear regression model $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$, we use the CUSUM test which is based on the *one-step ahead prediction errors*:

$$e_t^\circ \equiv y_t - \mathbf{x}_t' \mathbf{b}_{t-1}, \qquad t = k+1, k+2, \ldots, n,$$

where y_t and $\mathbf{x}_t$ are the t-th observation on the dependent variable and the k regressors, respectively. Here $\mathbf{x}_t' \mathbf{b}_{t-1}$ can be considered as a *predictor* of $\mathbf{y}_t$ based on the previous $t-1$ observations with $\mathbf{b}_{t-1}$ being defined by

$$\mathbf{b}_{t-1} = (\mathbf{X}_{t-1}' \mathbf{X}_{t-1})^{-1} \mathbf{X}_{t-1}' \mathbf{y}_{t-1},$$

where $\mathbf{y}_{t-1}$ is a $(t-1) \times 1$ vector containing the first $t-1$ observations on the dependent variable while $\mathbf{X}_{t-1}$ is a $(t-1) \times k$ matrix containing the first $t-1$ observations on the explanatory variables: Obviously, $\mathbf{b}_{t-1}$ is feasible only when the rank of $\mathbf{X}_{t-1}$ is equal to the number of explanatory variables k, i.e., when its

row number is no less than k. So we can compute $\mathbf{b}_{t-1}$, and therefore the recursive residuals e_t°, only for $t = k+1, k+2, \ldots, n$.

Let's also define the *rescaled* one-step ahead prediction errors:

$$w_t \equiv \frac{e_t^\circ}{\sqrt{1 + \mathbf{x}_t'(\mathbf{X}_{t-1}'\mathbf{X}_{t-1})^{-1}\mathbf{x}_t}}, \qquad t = k+1, k+2, \ldots, n,$$

and their sample variance:

$$s_w^2 \equiv \frac{1}{n-k-1}\sum_{t=k+1}^{n}(w_t - \overline{w})^2,$$

where $\overline{w} \equiv (n-k)^{-1}\sum_{t=k+1}^{n} w_t$. The CUSUM test is based on the fact that, under the null hypothesis that the regression coefficients $\boldsymbol{\beta}$ are stable, the distribution of the $n-k$ statistics

$$C_j \equiv \sum_{t=k+1}^{j}\frac{w_t}{s_w}, \qquad j = k+1, k+2, \ldots, n,$$

can be derived. In particular, we know the 90 %, 95 %, and 99 % critical values (denoted as $\psi_{0.05}$, $\psi_{0.025}$, and $\psi_{0.005}$, respectively) of this distribution are 0.850, 0.948, and 1.143, respectively. To conduct the CUSUM test at the $(1-\alpha)$% significance level, we check whether each point $(j,\ C_j)$ lies inside the corresponding confidence bound which is delimited by two straight lines that are symmetrical with respect to the horizontal axis: one is the line connecting the two points: $\left(k,\ \psi_{\alpha/2}\sqrt{n-k}\right)$ and $\left(n,\ 3\psi_{\alpha/2}\sqrt{n-k}\right)$. The other is the line connecting the two points: $\left(k,\ -\psi_{\alpha/2}\sqrt{n-k}\right)$ and $\left(n,\ -3\psi_{\alpha/2}\sqrt{n-k}\right)$. The null hypothesis is rejected if any of C_j lies outside the corresponding confidence interval.

Before writing a GAUSS program for calculating the CUSUM test statistics, we note that the computation repeatedly involves increasing matrices $\mathbf{y}_{t-1}$ and $\mathbf{X}_{t-1}$ and is hard to do in one stroke based on matrix algebra technique only. So using do-loops appears the only way for the task. Now, given that the n observations on the dependent variable and the k regressors have been loaded into 'y' and 'x', respectively, then the $n-k$ CUSUM test statistics C_j can be computed by the following simple GAUSS program:

```
w = zeros(n-k,1);
t = k+1;
do until t > n;
   e = y[t] - x[t,.]*invpd(x[1:t-1,.]'x[1:t-1,.])*x[1:t-1,.]'y[1:t-1];
   w[t-k] = e/sqrt(1 + x[t,.]*invpd(x[1:t-1,.]'x[1:t-1,.])*x[t,.]');
   t = t + 1;
endo;
s2 = (w - meanc(w))'(w - meanc(w))/(n-k-1);
c = cumsumc(w./sqrt(s2));
```

'c' is an $(n-k)\times 1$ vector containing the CUSUM test statistics C_j, for $j = k+1, k+2, \ldots, n$.

In the above program we have used a new GAUSS command 'cumsumc'. It takes one input which is an $(n-k)\times 1$ vector (of rescaled one-step ahead prediction errors w_t) and produced another vector of the same size whose elements are the cumulative sums of the elements of the input vector. A similar GAUSS command is 'cumprodc' which does cumulative products.

Finally, we note that to complete the testing procedure, we have to draw a graph for the confidence bound, which we will not do here.

Chapter 8

GAUSS Procedures: Basics

Recall in section 5.5, the three parts of the GAUSS program for computing the three Chow test statistics are almost identical. Each begins with the computation of two RSS (one for the restricted model and the other for the unrestricted model) and their respective degrees of freedom, then the test statistic and the corresponding p-value. The only difference in these computations is the data that are used. We may wonder whether there is a way in GAUSS to exploit the similarity in computations to save our programming chore. The answer is yes and the tool used for this purpose is called a procedure.

A procedure can be described in short as a group of GAUSS statements that is a self-contained unit residing in the program that accepts inputs and produces outputs. Once a procedure is defined inside a program, it can be used or, in GAUSS terminology, *called* repeatedly with different inputs to produce different outputs. For example, the part of the program for each of the three Chow test statistics can be made as a procedure. The inputs of such a procedure are three matrices: the vector of dependent variable, denoted as '`y`', the matrix of explanatory variables from the restricted model, denoted as '`xr`', and the one from the unrestricted model, denoted as '`xu`'. The output is the p-value of the Chow test statistic. The procedure can be formally written as follows:

```
proc (1) = chow(y,xr,xu);

local rss_r, rss_u, df_r, df_u, f, p_value;

rss_r = y'y - y'xr*invpd(xr'xr)*xr'y;
df_r = rows(xr) - cols(xr);

rss_u = y'y - y'xu*invpd(xu'xu)*xu'y;
df_u = rows(xu) - cols(xu);

f = ((rss_r - rss_u)/(df_r - df_u))/
    (rss_u/df_u);

p_value = cdffc(f,df_r-df_u,df_u);

retp(p_value);
endp;
```

Let's take a look at the structure of this procedure, which can be broken into five components:

1. The first component starts with the GAUSS command '`proc`' which initiates the definition of a procedure. The number in the first pair of parentheses indicates how many output matrices will be produced by this procedure. In our example there is one output – the p-value which is a scalar. The letters after

the equality sign, such as 'chow', give the name of the procedure which follows the same naming rule for variables. Whenever the procedure is needed, it will be called by this name. The arguments inside the second pair of parentheses are inputs. In our example, there are three inputs 'y', 'xr' and 'xu'.

2. The second component starts with the GAUSS command 'local' followed by a list of variable names. All the variables used inside the procedure, except those input variables specified in the 'proc' statement (e.g., the three matrices 'y', 'xr', and 'xu' in our example), are referred to as local variables and should be listed in this 'local' statement. In general, *all the variables that appear on the left-hand side of the GAUSS statements inside a procedure are local variables*. In our example, six local variables have been defined. The local statement may be omitted if no local variables are needed.

   In contrast to the local variables inside a procedure, all variables defined and used outside procedures are said to be global. The conceptual difference between local variables and global variables is quite important and it will be elaborated further in section 8.2.

3. The third component is the main body of the procedure, which may contain any number of GAUSS expressions that define the procedure. There are six expressions in our example.

4. The fourth component of the procedure is the 'retp' command that "returns" the output variables created by the procedure. Multiple output variables are possible and require multiple arguments in the parentheses. If there is no output variable to be returned, the 'retp' command may be omitted.

5. The last component is simply the 'endp' command that formally ends the definition of the procedures.

To use, or to call, the above procedure, we type the following command in the main program:

```
1  pv = chow(y,x1,x2);
```

where 'x1' and 'x2' are the matrices of explanatory variables from the restricted model and the unrestricted model, respectively. Both of them and the vector of dependent variable 'y' should have been defined earlier in the main program.

The output is stored in the new matrix (here, it is a scalar) 'pv', which is the p-value for the Chow test statistic. Note that the names of the input and output variables here are different from those used inside the procedure. As a matter of fact, all local variables used inside a procedure are completely independent of the global variables used outside the procedure. Furthermore, the local names used inside a procedure can be the same as some global names outside the procedure. GAUSS will understand they represent different variables. Also, all the global variables defined in a GAUSS program will remain in the memory *even after the execution of the program is finished*. But the local variables used inside a procedure will be erased from the memory *as soon as the execution of this procedure is completed*.

It is possible for a procedure to have no output variable at all. In such a case the number inside the first pair of parentheses in the 'proc' command is '0' and there is no need for the parentheses in the 'retp' command. What this kind of procedures usually do is to make some computation and/or to print something on the screen (or into an ASCII output file). For example, the goal of the previous procedure 'chow' is to compute the p-value of the Chow test statistic. The reason for deriving such a p-value is to eventually print it so that we can judge whether it is smaller than the designated size of the test, say, 5%, in which case the null

hypothesis of no structural change will be rejected. But we note GAUSS can also make such a judgment. What we really want is actually not the p-value per se but a decision of whether to reject the null. So we can rewrite the procedure in a way that it does not produce any matrix output but prints the testing decision.

```
proc (0) = chow_a(y,xr,xu);

local rss_r, rss_u, df_r, df_u, f, p_value;

rss_r = y'y - y'xr*invpd(xr'xr)*xr'y;
df_r = rows(xr) - cols(xr);

rss_u = y'y - y'xu*invpd(xu'xu)*xu'y;
df_u = rows(xu) - cols(xu);

f = ((rss_r - rss_u)/(df_r - df_u))/
    (rss_u/df_u);

p_value = cdffc(f,df_r-df_u,df_u);

if p_value < 0.05;
   "The Null Hypothesis of No Structural Change Is Rejected!";
else;
   "The Null Hypothesis of No Structural Change cannot be Rejected!";
endif;

endp;
```

Here, an '`if...else...endif`' command is used to determine which testing result to be printed. Since there is no output variables, the number of output specified in the '`proc`' command is set at 0, while the '`retp`' command is omitted. When such a procedure is needed, it is called by

```
call chow_a(y,x1,x2);
```

Let's consider the case where the number of output variables is more than one. In such a case the calling statement is a little different. Suppose in the original '`chow`' procedure, besides the p-value, we also want to output the test statistic '`f`' and the two corresponding degrees of freedom, then we need the following version of the procedure that has four output variables:

```
proc (4) = chow_b(y,xr,xu);

local rss_r, rss_u, df_r, df_u, f, p_value;

rss_r = y'y - y'xr*invpd(xr'xr)*xr'y;
df_r = rows(xr) - cols(xr);
```

```
7
8   rss_u = y'y - y'xu*invpd(xu'xu)*xu'y;
9   df_u = rows(xu) - cols(xu);
10
11  f = ((rss_r - rss_u)/(df_r - df_u))/
12      (rss_u/df_u);
13  p_value = cdffc(f,df_r-df_u,df_u);
14
15  retp(f,df_r-df_u,df_u,p_value);
16  endp;
```

The number of output specified in the 'proc' command is 4 and, correspondingly, there are four arguments in the 'retp' command, among which we note the second argument itself is an expression. That is, the arguments of the 'retp' command can take the form of algebraic expressions. This extra flexibility helps cut the number of local variables needed in the procedure. To call the above procedure, we type

```
1   {t,df1,df2,pv} = chow_b(y,x1,x2);
```

Note that the four output variables 't', 'df1', 'df2', and 'pv' are enclosed by a pair of braces. The structure of this statement is quite unique in GAUSS. It has more than one variable *on the left-hand side* of the equality sign.

Let's now present a full example. Suppose in the estimation of the Engel curve model we are interested in the difference between the developed and the developing countries. Since the data are not grouped into developed and developing countries as the framework of the Chow test requires, we must rearrange the data first. The original data needed are in the three ASCII files: 'share', 'totalexp', and 'country'. In particular, in the file 'country' there are two columns of characters: the first column contains the countries' names and the second one the continent codes: 'AF' (for Africa), 'AS' (for Asia), 'NA' (for North America), 'SA' (for South America), and 'EU' (for Europe). The continent codes are used to separate the developed countries from the developing ones.

```
1   new;
2
3   load s[60,10] = share;
4   load m[60,1] = totalexp;
5   load c[60,2] = country;
6
7   y = s[.,1];
8   x = ones(60,1)~ln(m);
9
10  /*  Constructing an indicator for the developed countries.              */
11  ind = (c[.,2] .== "NA" .or c[.,2] .== "EU");
12
13  /*  Reorganizing the data matrices 'y' and 'x' with the data for the
14      developed countries first, followed by the data for the developing
```

```
    countries.                                                    */
y = selif(y,ind)|delif(y,ind);
x = selif(x,ind)|delif(x,ind);

/*  The number of the developed countries (the number of 1's in the
    indicator.)                                                   */
n1 = sumc(ind);

/*  The expanded matrix of explanatory variables for the case 1 model.   */
x1 = (ones(n1,1)~zeros(n1,1)~x[1:n1,2])|
     (zeros(n-n1,1)~ones(n-n1,1)~x[(n1+1):n,2]);
/*  The expanded matrix of explanatory variables for the case 2 model.   */
x2 = (x[1:n1,.]~zeros(n1,2))|(zeros(n-n1,2)~x[(n1+1):n,.]);

format /rd 6,3;

{test,df1,df2,pv} = chow_b(y,x,x1);
" The Chow test statistic for testing the null hypothesis of";
" no structural change against the case 1 model is " test;
" The degrees of freedom are " df1;; " and " df2;
" The corresponding p-value is " pv;

{test,df1,df2,pv} = chow_b(y,x,x2);
" The Chow test statistic for testing the null hypothesis of";
" no structural change against the case 2 model is " test;
" The degrees of freedom are " df1;; " and " df2;
" The corresponding p-value is " pv;

{test,df1,df2,pv} = chow_b(y,x1,x2);
" The Chow test statistic for testing the null hypothesis of";
" the case 1 model against the case 2 model is " test;
" The degrees of freedom are " df1;; " and " df2;
" The corresponding p-value is " pv;

/*****************************************************************
 *          The Definition of the Procedure 'chow_b'.            *
 *****************************************************************/
proc (4) = chow_b(y,xr,xu);

local rss_r, rss_u, df_r, df_u, f, p_value;

rss_r = y'y - y'xr*invpd(xr'xr)*xr'y;
df_r = rows(xr) - cols(xr);

rss_u = y'y - y'xu*invpd(xu'xu)*xu'y;
```

```
60    df_u = rows(xu) - cols(xu);
61    f = ((rss_r - rss_u)/(df_r - df_u))/
62        (rss_u/df_u);
63    p_value = cdffc(f,df_r-df_u,df_u);
64
65    retp(f,df_r-df_u,df_u,p_value);
66    endp;
67    /*************************************************************************/
68
69    end;
```

Note that the definition of the procedure can be placed anywhere in the main program. Moreover, in chapter 9 we will also see how to store the procedure definitions in files different from the main program (it is just a matter of letting GAUSS know where to find the procedure definitions). The possibility of separating the main program from procedure definitions is quite useful. Just imagine the situation where there is a procedure containing some commonly used expressions which are needed by many different programs. If we must store the procedure definition in the same file with the main program, then each program that needs this procedure will have to include the procedure definition. This repetition bloats the program file and can be rather annoying.

8.1 Structural Programming

By using procedures we are able to shorten a program considerably. But reducing the size of a program is only part of the reason for using procedures. A more important purpose procedures serve is to make a program more "structural". That is, procedures help to break a program into several relatively independent units so that we can have a cleaner and broader vision of how the entire program works. For example, from the structure of the previous Chow test procedure we clearly see the fact that the routine of conducting various Chow tests is independent of the data '`y`', '`xr`', and '`xu`' it process. By including the computation in a self-contained procedure and specifying the data matrices outside the procedure, we provide the program with a structure that is easy to debug and more readily for future revision.

As a matter of a fact, the GAUSS language itself is built on several hundred of procedures. All the GAUSS commands that take input matrices (such as '`zeros`', '`ones`', '`eye`', '`seqa`', '`sumc`', '`meanc`', '`maxc`', '`minc`', '`ols`', etc.) are procedures. Whenever these commands are called, what GAUSS does is to find the corresponding procedure definitions and then execute them. (Although the definitions of these commands/procedures are not included in the program, GAUSS knows where to find them. We will explain how GAUSS does it in chapter 9.

There is another simple GAUSS command that also helps to make a long program more structural and less formidable. A program can be physically divided into more than one piece. Suppose a long program is divided into two parts and placed in two different files, say, '`part.1`' and '`part.2`'. To run such a program, we can submit the '`part.1`' file with the following command added at the end of the file:

```
1    #include part.2;
```

or, alternatively, submit the '`part.2`' file with the following command added at the beginning of the file:

```
#include part.1;
```

When a file is 'included', it is literally plugged into the place where it is 'included'. With this technique, it becomes possible to divide a long program into a few logically separable components and then 'include' them back into one program.

8.2 Accessing Global Variables Directly

Earlier when we discussed the 'local' statement, we mentioned that variables used inside the procedure definition were either inputs specified in the 'proc' statement or local variables declared by the 'local' statement. However, this statement is not exactly accurate. There is an important exception: all the global variables defined in the main program, but before the procedure is called, can be referred to and used inside the procedure definitions, given that the names of global variables do not conflict with any of the local variables.

For example, when we define 'chow_b' procedure, we do not really need to list the vector of dependent variable 'y' in the 'proc' command. That is, the following expression in that example

```
proc (4) = chow_b(y,xr,xu);
```

can be simplified to

```
proc (4) = chow_b(xr,xu);
```

The program will still work as long as the global variable 'y' is properly defined before calling this new version of the procedure. That is, the global variable 'y' can be referenced inside this new version of 'chow_b' just like the input variables or local variables. However, it should be emphasized that if 'y' has not been suitably defined before the procedure is called, then an error message will ensue.

The possibility of accessing global variables directly from inside a procedure helps to reduce the number of input variables necessarily specified in the 'proc' statement. But we should also note that such a practice makes a procedure less self-contained and therefore causes potential problems. Unlike the inputs or the local variables, which are explicitly specified in the 'proc' and the 'local' statements, the use of global variables usually is not clearly indicated in the procedure definition. So it is very easy to forget defining those global variables properly before calling the procedure. It is not hard to imagine how much trouble may happen to a long program containing hundreds of global variables and dozens of procedures trying to access various global variables. Nevertheless, in section 9.7 we will see there are some advanced commands in GAUSS (the 'external' and 'declare' commands) that help alleviate this difficulty.

Earlier when we discussed the 'ols' command in section 5.3, we mentioned the concept of "switches" and presented three of them: '__con', '__altnam', and '_olsres'. We also indicated that the 'ols' command, like many other GAUSS commands, was in fact a procedure. What we have been saying in this section is that all the procedures, certainly including the 'ols' command, can access global variables. What we called "switches" earlier are nothing but global variables which are accessed from inside the definition

of the ‘ols’ procedure. The function of these global variables is, as mentioned before, to control some aspect of the OLS computation. These global variables have their default values, but can be redefined before calling the ‘ols’ procedure.

Here a technical question arises: as we indicated before, the global variables must be prepared properly and explicitly before calling the procedure which accesses those global variables. But we also said that we do not really need to include the explicit definitions of the global variables, such as ‘__con’, ‘__altnam’, and ‘_olsres’, before we call the ‘ols’ procedure. How can this omission be allowed? Moreover, how are the default values of those global variables are defined? These questions will be answered in section 9.7 .

The above analysis of the global variables for the ‘ols’ command should also demonstrate why the ability of procedures to access the global variables is useful. If all those global variables, which allow us to conveniently control the execution of the ‘ols’ procedure, need to be defined as the input variables and listed in the ‘proc’ statement, then every time we call the ‘ols’ command, we have to specify all of them explicitly even though most of the time their values do not need to be changed. Repeatedly doing so can be quite annoying and prone to mistakes.

8.3 Calling Other Procedures in a Procedure

It is quite conceivable that the definition of a procedure may get too long to be comprehensible. So there may be a need to make part of a long procedure another self-contained procedure. Dealing with this kind of multi-level procedures actually is quite easy in GAUSS. All we have to do is in the first procedure to (1) include the name of the second procedure in the ‘proc’ statement; and (2) declare this name as a procedure in the ‘local’ statement. With these minor modification in the first procedure and no special treatment at all in the second procedure, we can call the second procedure from inside the first procedure just like calling the first procedure from the main program.

Let’s go back to the program of the three Chow tests. We will try to rewrite that program to demonstrate how the multi-level procedures can be used (although this change does not help simplify the program and appears unnecessary). Specifically, we will make the second part of that program a self-contained procedure, with the name ‘testing’, which calls the original procedure ‘chow_b’. That is, the main program, which becomes quite short, will call the procedure ‘testing’, while the procedure ‘testing’ will call the second procedure ‘chow_b’.

```
1  new;
2
3  load s[60,10] = share;
4  load m[60,1] = totalexp;
5  load c[60,2] = country;
6
7  y = s[.,1];
8  x = ones(60,1)~ln(m);
9
10 ind = (c[.,2] .== "NA" .or c[.,2] .== "EU");
11 n1 = sumc(ind);
12
13 y = selif(y,ind)|delif(y,ind);
```

```
14  x = selif(x,ind)|delif(x,ind);
15
16  x1 = (ones(n1,1)~zeros(n1,1)~x[1:n1,2])|
17       (zeros(n-n1,1)~ones(n-n1,1)~x[(n1+1):n,2]);
18  x2 = (x[1:n1,.]~zeros(n1,2))|(zeros(n-n1,2)~x[(n1+1):n,.]);
19
20  call testing(y,x,x1,x2,chow_b);
21
22  /*************************************************************************
23   *             The Definition of the Procedure 'testing'.                *
24   *************************************************************************/
25  proc (0) = testing(y,x,x1,x2,chowprg);
26
27  local test, df1, df2, pv, chowprg:proc;
28
29  format /rd 6,3;
30  {test,df1,df2,pv} = chowprg(y,x,x1);
31  " The Chow test statistic for testing the null hypothesis of";
32  " no structural change against the case 1 model is " test;
33  " The degrees of freedom are " df1;; " and " df2;
34  " The corresponding p-value is " pv;
35
36  {test,df1,df2,pv} = chowprg(y,x,x2);
37  " The Chow test statistic for testing the null hypothesis of";
38  " no structural change against the case 2 model is " test;
39  " The degrees of freedom are " df1;; " and " df2;
40  " The corresponding p-value is " pv;
41
42  {test,df1,df2,pv} = chowprg(y,x1,x2);
43  " The Chow test statistic for testing the null hypothesis of";
44  " the case 1 model against the case 2 model is " test;
45  " The degrees of freedom are " df1;; " and " df2;
46  " The corresponding p-value is " pv;
47
48  endp;
49
50  /*************************************************************************
51   *          The Definition of the Procedure 'chow_b'.                    *
52   *************************************************************************/
53  proc (4) = chow_b(y,xr,xu);
54
55  local rss_r, rss_u, df_r, df_u, f, p_value;
56
57  rss_r = y'y - y'xr*invpd(xr'xr)*xr'y;
58  df_r = rows(xr) - cols(xr);
```

```
59
60      rss_u = y'y - y'xu*invpd(xu'xu)*xu'y;
61      df_u = rows(xu) - cols(xu);
62      f = ((rss_r - rss_u)/(df_r - df_u))/
63          (rss_u/df_u);
64      p_value = cdffc(f,df_r-df_u,df_u);
65
66      retp(f,df_r-df_u,df_u,p_value);
67      endp;
68
69      end;
```

Obviously, what the procedure ‘testing’ does is simply calling the second procedure ‘chow_b’ (using the local name ‘chowprg’) three times and printing the testing results. What the main program does is to prepare the four data matrices ‘y’, ‘x’, ‘x1’, and ‘x2’.

Note that the definition of the procedure ‘chow_b’ is not changed at all. Also, the way the second procedure ‘chow_b’ is referred to in the procedure ‘testing’ is somewhat strange. Three points need to be discussed:

1. We have substituted another name ‘chowprg’ for the procedure name ‘chow_b’ everywhere inside the procedure ‘testing’. This name change is not necessary. But it underlines the important fact that the procedure name itself is nothing but a local reference inside the definition of the procedure ‘testing’. What is important is when the procedure ‘testing’ is called for execution, then the calling command has to correctly provide the procedure name, which is ‘chow_b’, for the local name ‘chowprg’, as is done by the fifth input of the calling command ‘call testing(y,x,x1,x2,chow_b);’ in the main program.

2. From point 1, we then understand why in the procedure ‘testing’ the name of the second procedure, designated by ‘chowprg’, must be listed in the ‘proc’ command just like any other input matrices. It is because the procedure ‘testing’ will have to input this information from the main program.

3. Since ‘chowprg’ listed in the ‘proc’ command is not an ordinary matrix input but a procedure name, we need to explicitly indicate this fact in the ‘local’ statement with the subcommand ‘:prg’ following the procedure name. (This rule is a bit unusual because we may be inclined to think the ‘local’ statement is for declaring the local variables only and should have nothing to do with the inputs listed in the ‘proc’ statement.)

8.4 String Inputs

In the previous section we have seen that, other than the matrices (both numeric and character ones), procedure names can also be the inputs listed in the ‘proc’ command. In this section we will consider another type of inputs, the string inputs, and discuss why we need string inputs in a procedure.

Suppose in the previous program for the Chow tests we want to print the testing results in an ASCII output file and we intend to do it inside the procedure ‘testing’. We could add the following command immediately after the ‘local’ command in the procedure ‘testing’:

```
output file = test.out on;
```

But after a couple of trials we will notice the problem that no matter which data set we use for testing, all the results will be printed into the same ASCII file ‘test.out’ and there is no control over the file name in the main program. To solve this problem, we have to consider a new device in procedure definitions. Let’s rewrite the first three expressions of the previous ‘testing’ procedure as follows:

```
proc (0) = testing(y,x,x1,x2,chowprg,outname);

local test, df1, df2, pv, chowprg:proc;

output file = ^outname on;

format /rd 6,3;

...
```

The input list in the ‘proc’ command is increased by an additional item ‘outname’, which also appears as the name of the ASCII output file, following a caret sign ‘^’.

It is important to know that ‘^outname’ in the ‘output file’ command is not the name of the output file. Instead, ‘outname’ is the name of a string variable that stores a string in it. It is the *string content* of the string variable ‘outname’ that defines the name of the output file. The exact content of the string variable ‘outname’ is provided by the main program so that the right to name the ASCII output file lies in the main program.

The way the main program calls the ‘testing’ procedure is as follows:

```
filename = "food_s.out";
call testing(y,x,x1,x2,chow_b,filename);
```

Here, ‘filename’ is a string variable. It contains a string ‘food_s.out’ which will be adopted as the name of the ASCII output file when the procedure ‘testing’ is executed. The above expressions can be simplified further as

```
call testing(y,x,x1,x2,chow_b,"food_s.out");
```

The key to the use of the string variables in procedure definitions is the caret sign ‘^’ before the string variable. We may interpret the caret sign as a reminder which says “Here Is a String Variable and Don’t Confuse the Variable Name with Its String Contents.”

Other common uses of the caret sign technique in procedure definitions include storing the string of path used with the ‘load’ and ‘save’ commands in string variables. For example, suppose in the above procedure of Chow tests we want to save the testing result ‘test’ in a matrix file (instead of in an ASCII file), then we

need to use a 'save' command followed by the name of the matrix file. Again, the name, as well as the path, of the matrix file can be specified by a string variable, say 'outfile'. In such a procedure we may have the following command:

```
save ^outfile = test;
```

while 'outfile', like 'outname' in the previous example, should also be listed in the 'proc' command such as

```
proc (0) = testing(y,x,x1,x2,chowprg,outfile);
```

To call this procedure in the main program, we need the commands

```
filespec = "c:\\mydir\\exp\\chow";
call testing(y,x,x1,x2,chow_b,filespec);
```

After the program is executed, the computed Chow test statistic will be save as the matrix file chow.fmt in the subdirectory c:\mydir\exp. Note that in the specification of the string contents each backslash '\' must be written as a double-backslash. The use of the string variable with the 'load' command is similar.

From the above discussions of the caret sign technique, we may arrive a conclusion that it is mainly used to specify the source and the destination of data input/output inside a procedure. We will see other applications of this technique later.

8.5 Functions: Simplified Procedures

"Functions" in GAUSS can be viewed as a special kind of procedures in which the definition can be written in a single expression and there is only one output variable. Unlike a procedure definition which starts with the somewhat complicated 'proc () =' command, the function definition starts with the simple command 'fn'. Also, the function definition does not need 'retp' and 'endp' commands to conclude the definition simply because the definition involves only one expression.

For example, in the previous sections the RSS has been computed many times while the computation involves a single expression only. So it is possible to make the RSS computation as a function:

```
fn rss(y,x) = y'y - y'x*invpd(x'x)*x'y;
```

This definition should be placed anywhere before the 'rss' function is called. The way of calling a function is exactly the same as calling a procedure. In the following use of the 'rss' function, we call the function inside a procedure definition. It is another possible version of the 'chow' procedure:

```
fn rss(y,x) = y'y - y'x*invpd(x'x)*x'y;

proc (1) = chow_c(y,xr,xu);
```

```
4
5   local rss_r, rss_u, df_r, df_u, f, p_value;
6
7   df_r = rows(xr) - cols(xr);
8   df_u = rows(xu) - cols(xu);
9
10  f = ((rss(y,xr) - rss(y,xu))/(df_r - df_u))/
11      (rss(y,xu)/df_u);
12
13  p_value = cdffc(f,df_r-df_u,df_u);
14
15  retp(pv);
16  endp;
```

8.6 Keywords: Specialized Procedures*

In section 1.1.1 we mentioned that all the DOS commands can be accessed even under the GAUSS command mode by using the 'dos' command. The example we used there was to copy a file 'prg.1' to another file 'prg.2', which was achieved by the command

```
1   dos copy prg.1 prg.2;
```

Now we have a challenging question: How do we write a procedure that can save us from typing the somewhat annoying command 'dos'. A possible procedure may look like the following one where the technique of string variables is employed:

```
1   proc (0) = copy(string);
2   string = "copy " $+ string;
3   dos ^string;
4   endp;
```

Here, we also use the '^' sign with the string variable 'string' after the 'dos' command. What this expression does is to execute a DOS command which is specified by the *content* of the string variable 'string'.

To call the above procedure we need to type

```
1   call copy("prg.1 prg.2");
```

So the use of the procedure makes the matter worse. The conclusion seems to be that it is impossible to use procedures to simplify some tasks that are already quite simple but still not simple enough for frequent uses. Keywords come to the rescue in such situations.

Keywords are special procedures that accept only one input, which must be a string variable, and return no output. The keyword definition corresponding to the above example is as follows:

```
keyword copy(string);
string = "copy " $+ string;
dos ^string;
endp;
```

Except that the first statement starts with ‘keyword’, the whole definition is identical to the corresponding procedure. The power of the keywords lies in the way how they are called. To call the above keyword, we type

```
copy prg.1 prg.2;
```

where we do not need to type ‘call’, the parentheses, and the quotation marks as in the procedure case. This is because all characters *after the keyword (i.e., ‘copy’) and before the semicolon at the end of the statement* are considered the string input for the corresponding keyword.

GAUSS itself provides keyword definitions for some frequently used DOS commands like ‘copy’, ‘xcopy’, and ‘dir’. These keywords help us avoid typing the word ‘dos’ and make the execution of these three commands exactly the same as under DOS. Since these DOS commands are used so often that being able to avoid typing the word ‘dos’ is indeed a blessing. The same technique can be used to simplify every DOS command.

Let’s consider another example. The DOS command ‘cd’ for changing directory is also used quite often though it is not made by GAUSS as a keyword yet. We can make ‘cd’ a keyword by the following definition:

```
keyword cd(string);
string = "cd " $+ string;
dos ^string;
endp;
```

Chapter 9

GAUSS Procedures: The Library System and Compiling

As mentioned before, GAUSS commands are nothing but ready-made procedures. To further analyze this feature of GAUSS commands, we divide GAUSS commands into two categories: intrinsic and extrinsic GAUSS command.

The intrinsic GAUSS commands are those which are written in *machine codes*. They are not recognizable to us but can be executed much faster than the ordinary procedures written in GAUSS. Intrinsic GAUSS commands are intrinsically bundled with the GAUSS language itself.

As to the extrinsic GAUSS commands, they are simply procedures that are written in GAUSS and can be understood by anyone who knows GAUSS. All extrinsic GAUSS commands are defined and contained in about 81 ASCII files, each of these files consists of the definitions of several procedures that perform similar functions. GAUSS gives each of these files a name with the '`.src`' extension and stores it in the '`c:\gauss\src`' subdirectory.

9.1 Autoloading and the Library Files

To execute extrinsic GAUSS commands or any procedures whose definitions are not defined in the main program, GAUSS will follow a specific guideline to search for files in which the procedures are expected to reside. This feature of GAUSS is called "autoloading". Whenever GAUSS spots a new name that has not yet been defined earlier in the program, GAUSS will immediately consider it as a procedure name and start the autoloading process. To fully understand how GAUSS searches for the appropriate files for procedure definitions, we need to learn another feature of GAUSS: the library system.

A library in GAUSS means an ASCII file that contains a list of file names and each file name is followed by a list of indented procedure names as follows:

```
c:\dir1\file1.src
    pcname11 : proc
    pcname12 : proc
    ...
c:\dir2\file2.src
    pcname21 : proc
    ...
```

where '`pcname11`', '`pcname12`', '`pcname21`', . . ., are the names of some procedures. In the above listing these procedure names are all indented and followed by the description '`: proc`', which indicates the listed names are procedures. (Information about function and keyword definitions, like that of procedures, can also be recorded in such library files. The descriptions will then be "`:fn`' and '`: keyword`', respectively.)

The above listing implies that the procedures 'pcname11', 'pcname12', ..., are contained in the file 'file1.src' while procedures 'pcname21', 'pcname22', ..., are contained in the file 'file2.src', etc. Obviously, one file can contain more than one procedure definition. A library file in GAUSS works just like a library: it provides information about the locations of procedures.

9.2 The 'GAUSS.LCG' Library File for Extrinsic GAUSS Commands

All library files must have the extension '.lcg' and must be stored in the 'c:\gauss\itemb' subdirectory.

There is a special library file 'gauss.lcg' that contains the file names of all 81 '.src' files where the definitions of all the extrinsic GAUSS commands are located. So this special library file has all the information for GAUSS to find the definitions of extrinsic GAUSS commands. In other words, when we run a program that contains extrinsic GAUSS commands , the "autoloader" (which executes the autoloading process) will search through the library file 'gauss.lcg' to get the location information about the extrinsic GAUSS commands.

9.3 The 'USER.LCG' Library File for User-Defined Procedures

Once we understand how autoloading works for extrinsic GAUSS commands, it seems natural to raise the question that whether we can write our own procedures, store them in some ASCII files, and make them work just like extrinsic GAUSS commands. The answer is yes. To do this, we just write procedures in a file and then catalog the file name in the special library file called 'user.lcg'. That is, if our program calls procedures which are not defined in the same program but whose locations are specified in the 'user.lcg' library file, then autoloader will find the procedure definitions.

To apply this new method to the Chow test example in section 8.3, we can include the procedure definitions of the two procedures 'testing' and 'chow_b' in a file, say, 'chowfile.1', put this file in a subdirectory 'c:\gauss\proc', and then create the ASCII file 'user.lcg' in the 'c:\gauss\itemb' subdirectory to include the following two lines:

```
1  c:\gauss\proc\chowfile.1
2      testing : proc
3      chow_b  : proc
```

These steps may be conveniently referred to as the catalog process. Once such a catalog process is completed, the definitions of the 'testing' and 'chow_b' procedures can be deleted from the main program and we have a much shorter and cleaner program. During the execution of this program, when GAUSS spots the name 'testing' which is not defined anywhere inside the program, then GAUSS will interpret 'testing' as a procedure name and start the autoloading process. Eventually, the autoloader will get the location information from the library file 'user.lcg' and find the definition of the 'testing' procedure in the file 'chowfile.1'.

9.4 Other Library Files

It is possible to give a library file a name other than 'user.lcg', so long as the extension is '.lcg' and the library file is in the 'c:\gauss\itemb' subdirectory. If we want to use different library file, then it is

necessary to indicate the name of the new library file in our program or, in GAUSS terminology, to activate the library file.

For example, if the name of the library file is, say, '`my.lcg`', then our main program should include the following line before the corresponding procedures are called:

```
library my;
```

The purpose of the '`library`' command is to activate the library files for autoloader. Note that the extension '`.lcg`' of the library file '`my.lcg`' is not needed in the '`library`' command.[1]

9.5 On-Line Help: Seeking Information as the Autoloader

Earlier in section 1.1 we mentioned it is possible to access on-line help by pressing `Alt-H`. When a help screen is displayed, pressing `H` again will give us the prompt '`Help On:`' at the bottom of the screen. It is from this prompt we are able to get all On-Line Help information by typing the topic or the command name. In particular, On-Line Help also allows us to view the definitions of all extrinsic commands and some accompanying documentation. So in principle we ought to be able to learn all GAUSS commands through On-Line Help, especially after we have already learned all the basics about GAUSS. Moreover, all those procedure definitions we create (the so-called user-defined procedures) can also be accessed through On-Line Help just like the way we access the help on GAUSS commands, given that the autoloader knows where to find them. The searching path that On-Line Help follows is exactly the same as the path autoloader uses to execute them in a program.

9.6 Compiling*

Recall that one reason for using procedures is to make programs more structural. Each procedure performs a particular function that is more or less independent of the rest of the program. When a program consists of many procedures, then our life will be easier if we can compile each procedure separately for syntax errors before running the whole program.

To compile procedures individually, we have to place these procedures in separate files. Let's consider the earlier example of the '`chow_b`' procedure:

```
proc (4) = chow_b(y,xr,xu);

...

retp(f,df_r-df_u,df_u,p_value);
endp;
```

Suppose this procedure definition alone is placed in the file '`chowfile.2`' and we execute the command

[1] A '`library`' command is used in the graph-drawing program presented in the appendix A. There the name of the library file is '`pgraph.lcg`' which contains the location information about all graph-drawing procedures.

```
compile chowfile.2 cpd_chow
```

then the linear regression procedure in the ASCII file 'chowfile.2' will be compiled into machine codes and saved in the file 'cpd_chow.gcg'. The second file name, i.e., 'cpd_chow', in the 'compile' command should not have extension because the extension '.gcg' will be automatically added. Also, the second file name can be omitted, in which case the compiled file will have the same name 'chowfile' as the source file but with the extension '.2' replaced by '.gcg'.

The compiled procedures stored in '.gcg' files can be executed much faster when they are called. Also, compiling procedures offers a way to hide the source codes from users to protect the copyright.

When a file is compiled, everything in the memory will be saved along with the compiled file, irrespective whether those things are relevant to the compiled file or not. To prevent these extraneous things being saved, add the 'new' command at the beginning of the source file. Also, if line numbers for debugging are not needed, add the '#linesoff' command at the beginning of the source file. Without line numbers the size of the compiled file can be reduced, sometimes substantially.

To call a compiled procedure, which is saved in the file, say, 'cpd_chow.gcg', we add the 'use' command at the top of the calling program (the main program who calls these procedures). For example,

```
use cpd_chow;
...
call chow_b(y,x1,x2);
...
```

Upon a compiled procedure is 'used', all previous variables and procedures existing in the memory will be erased. So it is generally necessary to place the 'use' command at the very top of the program.

9.7 The External and Declare Commands

As has been mentioned earlier, it is possible for a procedure to directly access global variables from the main program. But if a procedure used global variables, then it cannot be compiled independently from the main program without causing the "undefined symbol" syntax error. This is because global variables are defined by the main program and exist only when the main program is also compiled. To solve this problem, we need to "externalize" those global variables from the procedure definition by the 'external' command and then provide them with some temporary values (just for the compiling purpose) through the 'declare' command.

Let's go back to the 'chow_b' procedure again. Suppose the original input variable 'y' is now deleted from the input list in the 'proc' statement so that it is considered a global variable in the procedure. If we try to compile the file 'chowfile.2' that contains such a procedure, the error message of "undefined symbol" will result because the variable 'y' is used but has not been properly defined first. To avoid such a problem, we should add two more statements before the procedure definition:

```
external matrix y;
declare matrix y != 0;
```

```
3
4   proc (4) = chow_b(xr,xu);
5
6   ...
7
8   retp(f,df_r-df_u,df_u,p_value);
9   endp;
```

The ‘external’ command is used to remind the GAUSS compiler of the variables that have been used but are neither the input variables (specified in the ‘proc’ statement) nor the local variables (specified by the ‘local’ statement). The ‘external’ command is certainly a useful reminder to ourselves, too.

The ‘declare’ command is used to provide a global variable with an initial value for the compiling purpose. This initial value is in fact temporary because it will usually be modified by the main program. Every variable that appears in the ‘external’ command should be initialized by a corresponding ‘declare’ command.

The special notation ‘!=’ in the ‘declare’ statement means the initial value of the global variable is allowed to be modified by the main program. (The exclamation mark ‘!’ may be omitted there.) There are two alternatives which are less frequently used: ‘:=’ will result in an error message of “redefinition” whenever the main program tries to reset the value of the corresponding global variable. ‘?=’ does not allow the initial value to be reset at all by the main program. Finally, we note all three forms of the equality sign have to be followed by a constant, either a number or a string in quotation marks. No expressions or variable names are permitted after these equality signs.

Other than global matrices, the ‘external’ and ‘declare’ commands can also be used for global strings. Let’s consider an alternative version of the ‘testing’ procedure where an ASCII output file is opened inside the procedure with the file name defined by a *global* string variable ‘outname’:

```
1   proc (0) = testing(y,x,x1,x2,chowprg);
2
3   ...
4   output file = ^outname reset;
5   ...
6
7   endp;
```

Then before compiling the above procedure, the following ‘external’ and ‘declare’ commands must be added before the procedure definition:

```
1   external string outname;
2   declare string outname != "test.out";
3
4   proc (0) = testing(y,x,x1,x2,chowprg);
5
6   ...
```

```
7
8   endp;
```

where the string variable ‘outname’ for the name of the output file becomes global.

If there a large numbe of global matrices and strings used in procedure definitions, it is better to place all the ‘external’ commands in a separate file with a file name having the ‘.ext’ extension and all the ‘declare’ commands in another file with a file name having the ‘.dec’ extension. When the corresponding procedure is to be compiled, we can ‘include’ the ‘.ext’ file before the procedure. The ‘.dec’ file needs not be ‘included’. It can be recorded in a library file so that they can be accessed just like the procedure definitions are.

Let’s consider the ‘chow_b’ procedure again. Suppose the following ‘external’ commands are in the file ‘chowfile.ext’:

```
1   external matrix y;
```

and the following ‘declare’ commands are in the file ‘chowfile.dec’:

```
1   declare matrix y != 0;
```

then we add the ‘#include’ command at the beginning of the ‘chowfile.2’ file where the linear regression procedure ‘chow_b’ is defined:

```
1   #include chowfile.ext;
2
3   proc (4) = chow_b(xu.xr);
4
5   ...
6
7   retp(f,df_r-df_u,df_u,p_value);
8   endp;
```

while the library file ‘user.lcg’ should include the following information about the location of the file ‘chowfile.dec’:

```
1   chowfile.dec
2       y: matrix
```

Note that the second statement should be indented. Also, the description ‘matrix’ for ‘y’ is new. The other possible descriptions were ‘proc’, ‘fn’, ‘keyword’, and ‘string’.

Let’s summarize the main points in this subsection: To compile an individual procedure, all the global variables used in the procedure definitions should be listed in the ‘external’ command and initialized by the ‘declare’ command. The ‘external’ command must go with, or should at least be ‘included’ with, the procedure definition. But the ‘declare’ command can be placed in other file, as long as the autoloader can find it.

Chapter 10

Nonlinear Optimization

One of the most important tasks that computers can do well is nonlinear optimization; that is, given a real-valued nonlinear objective function $f(\boldsymbol{\theta})$ of a k-vector of independent variables $\boldsymbol{\theta}$, computer can be used to numerically search for the value of $\boldsymbol{\theta}$ that optimizes, either maximizes or minimizes, the function $f(\boldsymbol{\theta})$. We can confine our discussions to the case of minimization without loss of generality since maximizing $f(\boldsymbol{\theta})$ is equivalent to minimizing its negative $-f(\boldsymbol{\theta})$. In this chapter we first discuss some common numerical algorithms for nonlinear minimization and then introduce a ready-made GAUSS program that implements these algorithms.

10.1 Newton's Method

Most algorithms for nonlinear minimization are based on iterations; that is, we start with an initial value $\boldsymbol{\theta}_0$ and try to reach the optimal value through a step-by-step iterative procedure. In each step we generate a new value $\boldsymbol{\theta}_{j+1}$ by adding a term to the old value $\boldsymbol{\theta}_j$ from the previous step:

$$\boldsymbol{\theta}_{j+1} = \boldsymbol{\theta}_j + s_j \cdot \mathbf{d}_j, \qquad j = 0, 1, 2, \ldots \tag{10.1}$$

where the modification term consists of two parts: the vector $\mathbf{d}_j$ gives the *step direction* and the scalar s_j is the *step length*.

To facilitate the analysis of (10.1), let us assume the objective function is twice-differentiable and consider the first-order Taylor expansion of the objective function $f(\boldsymbol{\theta}_{j+1})$ around $\boldsymbol{\theta}_j$:

$$f(\boldsymbol{\theta}_{j+1}) \approx f(\boldsymbol{\theta}_j) + \mathbf{g}(\boldsymbol{\theta}_j)'(\boldsymbol{\theta}_{j+1} - \boldsymbol{\theta}_j) = f(\boldsymbol{\theta}_j) + s_j \cdot \mathbf{g}(\boldsymbol{\theta}_j)'\mathbf{d}_j,$$

where

$$\mathbf{g}(\boldsymbol{\theta}) \equiv \frac{\partial f(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

is the k-vector of the first-order derivatives of $f(\boldsymbol{\theta})$ with respect to the k-vector $\boldsymbol{\theta}$, which is frequently called the gradient of $f(\boldsymbol{\theta})$. If we rewrite the above approximation as follows:

$$f(\boldsymbol{\theta}_{j+1}) - f(\boldsymbol{\theta}_j) \approx s_j \cdot \mathbf{g}(\boldsymbol{\theta}_j)'\mathbf{d}_j,$$

then it is readily seen that an easy way to make the right-hand side term negative is to define $\mathbf{d}_j$ as $\mathbf{A}\mathbf{g}(\boldsymbol{\theta}_j)$ for some $k \times k$ negative-definite matrix $\mathbf{A}$. This is because in such a case the right-hand side term becomes $\mathbf{g}(\boldsymbol{\theta}_j)'\mathbf{A} \cdot \mathbf{g}(\boldsymbol{\theta}_j)$ and it is a negative scalar. In other words, the iteration (10.1) should be specified as

$$\boldsymbol{\theta}_{j+1} = \boldsymbol{\theta}_j + s_j \cdot \mathbf{A} \cdot \mathbf{g}(\boldsymbol{\theta}_j), \qquad j = 0, 1, 2, \ldots \tag{10.2}$$

The problem now is to find a good $k \times k$ negative-definite matrix $\mathbf{A}$. To this aim, let us examine the first-order condition $\mathbf{g}(\boldsymbol{\theta}) = \mathbf{0}$ for the minimization of $f(\boldsymbol{\theta})$. If the new value $\boldsymbol{\theta}_{j+1}$ is close to the optimum

so that it satisfies the first-order condition, then we have the following first-order Taylor expansion of the gradient $\mathbf{g}(\boldsymbol{\theta}_{j+1})$ around $\boldsymbol{\theta}_j$:

$$\mathbf{g}(\boldsymbol{\theta}_{j+1}) \approx \mathbf{g}(\boldsymbol{\theta}_j) + \mathbf{H}(\boldsymbol{\theta}_j)(\boldsymbol{\theta}_{j+1} - \boldsymbol{\theta}_j) = \mathbf{0}, \tag{10.3}$$

where

$$\mathbf{H}(\boldsymbol{\theta}) \equiv \frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{\partial^2 f(\boldsymbol{\theta})}{\partial \boldsymbol{\theta} \partial \boldsymbol{\theta}'}$$

is the $k \times k$ matrix of the second-order derivatives of $f(\boldsymbol{\theta})$ with respect to the k-vector $\boldsymbol{\theta}$, which is frequently called Hessian of $f(\boldsymbol{\theta})$. We note that $\mathbf{H}(\boldsymbol{\theta}_j)$ is necessarily positive definite when $\boldsymbol{\theta}_j$ is sufficiently close to the optimum. From (10.3) we solve for $\boldsymbol{\theta}_{j+1} = \boldsymbol{\theta}_j - \mathbf{H}(\boldsymbol{\theta}_j)^{-1}\mathbf{g}(\boldsymbol{\theta}_j)$, which suggests a possible specification for (10.2) with $s_j = 1$ and $\mathbf{A} = -\mathbf{H}(\boldsymbol{\theta}_j)^{-1}$. If in (10.2) we allow a general step length s_j and set the step direction $\mathbf{d}_j$ as $-\mathbf{H}(\boldsymbol{\theta}_j)^{-1}\mathbf{g}(\boldsymbol{\theta}_j)$, then the iteration formula becomes

$$\boldsymbol{\theta}_{j+1} = \boldsymbol{\theta}_j - s_j \cdot \mathbf{H}(\boldsymbol{\theta}_j)^{-1}\mathbf{g}(\boldsymbol{\theta}_j), \qquad j = 0, 1, 2, \ldots, \tag{10.4}$$

which is the basic formula for *Newton's Method*. Obviously, to implement Newton's method, we have to repeatedly compute the gradient $\mathbf{g}(\boldsymbol{\theta}_j)$, the inverted Hessian $\mathbf{H}(\boldsymbol{\theta}_j)^{-1}$, as well as the step length s_j. We shall discuss these computation problems in details shortly.

When applying Newton's method to solve a nonlinear minimization problem, we iterate the formula (10.4) until convergence is reached. Convergence is usually based on one or more of the following three criteria:

1. When the absolute value of the difference between successive results $\|\boldsymbol{\theta}_{j+1} - \boldsymbol{\theta}_j\|$ is smaller than a desired level (such as 10^{-7});

2. When the absolute value of the difference between successive functional values $|f(\boldsymbol{\theta}_{j+1}) - f(\boldsymbol{\theta}_j)|$ is smaller than a desired level;

3. When the absolute value of the gradients $\|\mathbf{g}(\boldsymbol{\theta}_{j+1})\|$ is smaller than a desired level.

10.1.1 The Computation of Gradients

It is quite often that deriving analytic gradients, i.e., the explicit mathematical expression for $\mathbf{g}(\boldsymbol{\theta})$, is fairly difficult and prone to errors, in which cases we need computers to numerically approximate the gradients. Such *numerical gradient* can be computed based on two operational formulas for derivatives:

1. *The forward-difference approximation*: the ith element of the gradient vector $\mathbf{g}(\boldsymbol{\theta}_j)$ is approximated by

$$g_i(\boldsymbol{\theta}_j) \approx \frac{f(\boldsymbol{\theta}_j + \epsilon \cdot \boldsymbol{\iota}_i) - f(\boldsymbol{\theta}_j)}{\epsilon}, \qquad i = 1, 2, \ldots, k,$$

   where $\boldsymbol{\iota}_i$ is a k-dimensional vector of zeros except that the ith element is one, while ϵ is a very small scalar, such as 10^{-8}. It should be pointed out that, given the value of $f(\boldsymbol{\theta}_j)$, the calculation of the entire k-dimensional gradient vector $\mathbf{g}(\boldsymbol{\theta}_j)$ requires k evaluations of the objective function $f(\boldsymbol{\theta}_j + \epsilon\boldsymbol{\iota}_i)$, for $i = 1, 2, \ldots, k$.

2. *The central-difference approximation*: the ith element of the gradient vector $\mathbf{g}(\boldsymbol{\theta}_j)$ is approximated by

$$g_i(\boldsymbol{\theta}_j) \approx \frac{f(\boldsymbol{\theta}_j + \epsilon \cdot \boldsymbol{\iota}_i) - f(\boldsymbol{\theta}_j - \epsilon \cdot \boldsymbol{\iota}_i)}{2\epsilon}, \qquad i = 1, 2, \ldots, k.$$

Given the value of $f(\boldsymbol{\theta}_j)$, using this formula to calculate the gradient vector $\mathbf{g}(\boldsymbol{\theta}_j)$ requires $2k$ evaluations of the objective function $f(\boldsymbol{\theta}_j + \epsilon \cdot \boldsymbol{\iota}_i)$ and $f(\boldsymbol{\theta}_j - \epsilon \cdot \boldsymbol{\iota}_i)$, for $i = 1, 2, \ldots, k$. Hence, the central-difference approximation requires two times computation time than the forward-difference approximation does. However, the central-difference approximation can only achieve slightly better accuracy.

Numerical gradients generally take much longer computation time than evaluating analytic gradients. Thus, it is usually advisable to provide the computer program with the analytic gradients when they are available.

10.1.2 The Computation of Hessian

Other than analytically deriving Hessian, we can also apply the same idea for numerical gradients to approximate Hessian. For example, based on the forward-difference approximation we can use the following formula for the (p, q)th element of the Hessian matrix:

$$h_{pq}(\boldsymbol{\theta}_j) \approx \frac{f(\boldsymbol{\theta}_j + \epsilon \cdot \boldsymbol{\iota}_p + \epsilon \cdot \boldsymbol{\iota}_q) - f(\boldsymbol{\theta}_j + \epsilon \cdot \boldsymbol{\iota}_p) - f(\boldsymbol{\theta}_j + \epsilon \cdot \boldsymbol{\iota}_q) + f(\boldsymbol{\theta}_j)}{\epsilon^2}, \tag{10.5}$$

for $p \geq q = 1, 2, \ldots, k$. Given the value of $f(\boldsymbol{\theta}_j)$, the calculation of numerical Hessian $\mathbf{H}(\boldsymbol{\theta}_j)$, which is a symmetric matrix, requires at least $k(k+1)/2$ evaluations of the objective function $f(\boldsymbol{\theta}_j + \epsilon \cdot \boldsymbol{\iota}_p + \epsilon \cdot \boldsymbol{\iota}_q)$ as well as k evaluations of $f(\boldsymbol{\theta}_j + \epsilon \cdot \boldsymbol{\iota}_p)$, for $p, q = 1, 2, \ldots, k$. Obviously, such computation, as well as the inversion of the resulting Hessian which is needed in Newton's formula (10.4), take considerable time. Moreover, the desired level of accuracy is usually hard to maintained with (10.5). If analytic gradients $\mathbf{g}(\boldsymbol{\theta}_j)$ are available, then we can use a more accurate approximation as follows:

$$h_{pq}(\boldsymbol{\theta}_j) \approx \frac{g_p(\boldsymbol{\theta}_j + \epsilon \cdot \boldsymbol{\iota}_q) - g_p(\boldsymbol{\theta}_j)}{\epsilon}, \tag{10.6}$$

for $p \geq q = 1, 2, \ldots, k$. Such calculation requires $k(k+1)/2$ evaluations of the gradient functions and can produce more accurate results than (10.5).

Since deriving analytic Hessian is generally quite cumbersome and evaluating numerical Hessian is extremely time-consuming, other approaches have been suggested. For example, it is possible to consider an abridged numerical Hessian in which all off-diagonal elements are set to zero: $h_{pq}(\boldsymbol{\theta}_j) = 0$, for $p \neq q$. This special case of Newton's method, which requires only k evaluations of the gradient functions, is referred to as the *steepest descent algorithm*. In the next subsection we consider another class of algorithms, the so-called *Quasi-Newton Method*, that circumvent almost all Hessian computation.

10.1.3 Quasi-Newton Method

The main idea of Quasi-Newton method is to avoid the tedious evaluations of Hessian and directly modify (or update) the successive inverted Hessian using gradients only. Specifically, we construct the new value of inverted Hessian at the $(j+1)$th step, denoted as $\mathbf{H}_1^{-1}$, by modifying the previous value $\mathbf{H}_\circ^{-1}$ at the jth

step using gradients $\mathbf{g}(\boldsymbol{\theta}_j)$ and $\mathbf{g}(\boldsymbol{\theta}_{j+1})$. To see this, let us first define two vectors $\mathbf{u} \equiv -\mathbf{H}_\circ^{-1}\mathbf{g}(\boldsymbol{\theta}_j)$ and $\mathbf{v} \equiv \mathbf{g}(\boldsymbol{\theta}_{j+1}) - \mathbf{g}(\boldsymbol{\theta}_j)$. Two versions of updating formulas have been proposed.[1]

1. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) update is:

$$\mathbf{H}_1^{-1} = \mathbf{H}_\circ^{-1} - \frac{1}{\mathbf{v}'\mathbf{u}}\left(\mathbf{H}_\circ^{-1}\mathbf{v}\mathbf{u}' + \mathbf{u}\mathbf{v}'\mathbf{H}_\circ^{-1}\right) + \frac{\mathbf{v}'\mathbf{H}_\circ^{-1}\mathbf{v}}{(\mathbf{v}'\mathbf{u})^2}\cdot\mathbf{u}\mathbf{u}' + \frac{1}{\mathbf{v}'\mathbf{u}}\cdot\mathbf{u}\mathbf{u}'.$$

2. The Davidon-Fletcher-Powell (DFP) update is

$$\mathbf{H}_1^{-1} = \mathbf{H}_\circ^{-1} - \frac{1}{\mathbf{v}'\mathbf{H}_\circ^{-1}\mathbf{v}}\cdot\mathbf{H}_\circ^{-1}\mathbf{v}\mathbf{v}'\mathbf{H}_\circ^{-1} + \frac{1}{\mathbf{v}'\mathbf{u}}\cdot\mathbf{u}\mathbf{u}'.$$

These two formulas can be further modified by multiplying the first three right-hand side terms of the BFGS update and the first two right-hand side terms of the DFP update, respectively, by $\mathbf{v}'\mathbf{u}/\mathbf{v}'\mathbf{H}_\circ^{-1}\mathbf{v}$. The resulting formulas are referred to as the *scaled BFGS update* and the *scaled DFP update*, respectively.

10.1.4 Newton's Method for Maximum Likelihood Estimation

In econometrics applications the need for nonlinear optimization is perhaps most apparent in deriving the maximum likelihood estimator (MLE). Given the sample x_1, x_2, ..., x_n which are assumed to be independently distributed with respective density functions $f_i(x_i|\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is an unknown k-dimensional parameter and the density functions f_i may *not* be identical, the MLE of $\boldsymbol{\theta}$ is derived by maximizing the log-likelihood function with respect to $\boldsymbol{\theta}$:

$$\ln L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \ln\, f_i(x_i|\boldsymbol{\theta}).$$

This maximization is equivalent to minimizing the negative log-likelihood function $-\ln L(\boldsymbol{\theta})$.

Since the second-order derivative of the log-density function $\partial^2 \ln f_i(x_i|\boldsymbol{\theta})/\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'$, as a function of the random variable x_i, is a random variable, the law of large numbers implies that, under certain regularity conditions, we have

$$\frac{1}{n}\frac{\partial^2 \ln L(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'} - \frac{1}{n}\mathrm{E}\left[\frac{\partial^2 \ln L(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'}\right] = \frac{1}{n}\sum_{i=1}^{n}\left\{\frac{\partial^2 \ln\, f_i(x_i|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'} - \mathrm{E}\left[\frac{\partial^2 \ln\, f_i(x_i|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'}\right]\right\} \quad \xrightarrow{\text{p}} \quad 0,$$

as the sample size n goes to infinity. Consequently, the Hessian matrix in (10.4), which in the present case is $-\partial^2 \ln L(\boldsymbol{\theta})/\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'$, can be approximated by its expectation (which in the MLE theory is called the information matrix):

$$\mathbf{H}(\boldsymbol{\theta}_j) \;\approx\; -\mathrm{E}\left[\frac{\partial^2 \ln L(\boldsymbol{\theta}_j)}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'}\right]. \tag{10.7}$$

Newton's method with such an approximate Hessian is called the *method of scoring*. Its usefulness lies on the fact that taking expectation usually helps simplify the expression of Hessian.

---

[1] Both of these formulas for $\mathbf{H}_1^{-1}$ satisfy the equality $\mathbf{H}_1^{-1}\mathbf{v} = \mathbf{u}$ or, equivalently, $\mathbf{g}(\boldsymbol{\theta}_j) - \mathbf{g}(\boldsymbol{\theta}_{j+1}) = \mathbf{H}_1\mathbf{H}_\circ^{-1}\mathbf{g}(\boldsymbol{\theta}_j)$. This condition can be motivated by the following first-order Taylor expansion of gradient $\mathbf{g}(\boldsymbol{\theta}_j)$ around $\boldsymbol{\theta}_{j+1}$: $\mathbf{g}(\boldsymbol{\theta}_j) = \mathbf{g}(\boldsymbol{\theta}_{j+1}) + \mathbf{H}_1(\boldsymbol{\theta}_j - \boldsymbol{\theta}_{j+1})$, while from (10.4) we have $\boldsymbol{\theta}_j - \boldsymbol{\theta}_{j+1} = \mathbf{H}_\circ^{-1}\mathbf{g}(\boldsymbol{\theta}_j)$ (where the step length s_j is set to one).

There is one more approximation to the Hessian matrix of the log-likelihood function that is based on the equality

$$-\mathrm{E}\left[\frac{\partial^2 \ln f_i(x_i|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'}\right] = \mathrm{E}\left[\frac{\partial \ln f_i(x_i|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \cdot \frac{\partial \ln f_i(x_i|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}'}\right].$$

Again, the law of large numbers implies that, under certain regularity conditions, we have

$$-\frac{1}{n}\frac{\partial^2 \ln L(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'} - \frac{1}{n}\sum_{i=1}^{n} \frac{\partial \ln f_i(x_i|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}} \cdot \frac{\partial \ln f_i(x_i|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}'} \quad \xrightarrow{\text{p}} \quad 0,$$

as the sample size n goes to infinity. As a result, we have another approximation to the Hessian matrix:

$$\mathbf{H}(\boldsymbol{\theta}_j) \approx \sum_{i=1}^{n} \frac{\partial \ln f_i(x_i|\boldsymbol{\theta}_j)}{\partial\boldsymbol{\theta}} \cdot \frac{\partial \ln f_i(x_i|\boldsymbol{\theta}_j)}{\partial\boldsymbol{\theta}'}. \tag{10.8}$$

Newton's method with such an approximate Hessian is referred to as the Berndt-Hall-Hall-Hausman (BHHH) method.

Asymptotic Standard Errors for MLE: The asymptotic theory for the MLE indicates that for the MLE $\hat{\boldsymbol{\theta}}$, we have

$$\hat{\boldsymbol{\theta}} \overset{\text{A}}{\sim} \mathcal{N}\left(\boldsymbol{\theta}, \quad \left\{-\mathrm{E}\left[\frac{\partial^2 \ln L(\boldsymbol{\theta})}{\partial\boldsymbol{\theta}\partial\boldsymbol{\theta}'}\right]\right\}^{-1}\right).$$

Hence, if we apply the method of scoring or the BHHH method, then the inverse of the Hessian matrix (10.7) or (10.8) after convergence is reached can be readily used as an approximate variance-covariance matrix for the MLE (so that the square roots of its diagonal elements are standard errors). In other words, no further computation is needed to obtain the standard errors of the MLE.

Finally, we should note that when quasi-Newton method is used to derive the MLE, the inverse of the approximate Hessian, constructed either by the BFGS formula or by the DFP formula, *cannot* be used as an approximate variance-covariance matrix for the MLE. Hence, after convergence is arrived, it is still necessary to go one step further to numerically evaluate Hessian based on (10.5), (10.6), (10.7), or (10.8).

10.1.5 The Computation of the Step Length

Once the gradient $\mathbf{g}(\boldsymbol{\theta}_j)$ and Hessian $\mathbf{H}(\boldsymbol{\theta}_j)$ (or its approximation) for the new step direction in (10.4) are obtained, the value of the objective function can be viewed as a function of the step length s_j only. The determination of s_j then becomes a one-dimensional minimization problem (which is sometimes called a line search problem). There are two popular line search methods: the *backtrack algorithm* and the *golden search algorithm*. Each method involves a particular trial-and-error search scheme. Here, without going into details, we only point out that the backtrack algorithm requires less computation while the golden search algorithm is more effective in finding a better step length. Finally, we note that setting $s_j = 1$ indiscriminately usually worsens instead improves the value of the objective function.

10.2 A GAUSS Program for Nonlinear Minimization: NLOPT

NLOPT is a self-contained GAUSS procedure for nonlinear minimization. Once the procedure is properly installed, all a user needs to do is to write the (differentiable) objective function as a function of a vector of independent variables in the format of GAUSS procedure.

The source codes of the procedure **NLOPT** are stored in the file 'NLOPT.GCG'. To use it, simply add the command 'use nlopt;' at the beginning of the file where you type your program and place the 'NLOPT.GCG' file in the same subdirectory as your program file.

Consider a simple example of minimizing the following function of a single variable:

$$f(\theta) = -3\theta^2 e^{-\theta^3}.$$

To solve this problem, we write a short program to call the procedure **NLOPT**:

```
new;
use nlopt;

proc ofn(t);
  local f;
  f = -3*(t^2)*exp(-t^3);
  retp(f);
endp;

start = 1;

{parout,vof,gout,retcode} = nlopt(&ofn,start);

end;
```

The first part of the program after the 'use nlopt;' command is the definition of procedure 'ofn' which takes one input 't' and yields one output 'f'. It is the place where we write the definition of the objective function $f(\boldsymbol{\theta})$. The input is of course the vector of independent variables (i.e., $\boldsymbol{\theta}$) and the output is the value of the objective function (i.e., f). In the above example there is only one independent variable and the value of the objective function is also a scalar.

It should be pointed out that in cases where there are more than one independent variable in the vector $\boldsymbol{\theta}$, an important consideration in setting up the definition of the 'ofn' procedure is that the values of independent variables should be made about the same magnitude. For example, suppose we are considering a function of two independent variables $f(\theta_1, \theta_2)$. If the optimal value of θ_1 is 12345.67890 while that of θ_2 is 0.9876, then we should redefine the function as $f(10000\,\bar{\theta}_1, \theta_2)$ so that the function will be minimized with the solution 1.234567890 and 0.9876 for $\bar{\theta}_1$ and θ_2, respectively. Doing this helps smoothing the convergence of the iterations and increasing the accuracy of the solution.

Immediately after the procedure definition is the specification of the column vector 'start' (which is a scalar in the above example). This vector contains the initial values of the independent variable $\boldsymbol{\theta}$, which are needed to start Newton's iteration formula (10.4). Initial values are usually set by the user subjectively.

The procedure **NLOPT** is called by the 'nlopt' command which takes two inputs and produces four outputs. The two inputs are respectively the procedure 'ofn' and the vector 'start' which have just been defined. Among the four outputs, the column vector 'parout' contains the solution; the scalar 'vof' gives the minimized function value evaluated at 'parout'; the column vector 'gout' is the gradient evaluated at 'parout'; and 'retcode' is a returned code indicating the program execution status. Returned codes are explained in Table 1.

Table 1

Code	Explanations
0	Convergence is achieved;
1	Computation is terminated manually;
2	The preset number of iterations 100,000 is exceeded without convergence;*
3	Function evaluation fails;
4	Gradient calculation fails;
5	Hessian calculation fails;
6	Step length calculation fails;
7	Function cannot be evaluated at initial values;
8	The dimension of the initial vector '`start`' differs from the dimension of the gradient vector;
9	Gradient returns a column vector rather than the required row vector;
10	Quasi-Newton method fails;
20	Hessian is singular.

* The preset number of iterations is set by the global variable '`_cfiter`'.

During the execution of the **NLOPT** procedure, intermediate results on $\boldsymbol{\theta}_j$ and the corresponding *relative gradient* $\mathbf{g}(\boldsymbol{\theta}_j)\cdot\boldsymbol{\theta}_j / f(\boldsymbol{\theta}_j)$ from each iteration will be shown on the computer screen.[2] Moreover, the **NLOPT** procedure will check the percentage changes in $\boldsymbol{\theta}_j$ after each iteration to see if they are all smaller than 10^{-5} (which is set by the global variable '`_cfxchk`'). If they are, then the **NLOPT** procedure will further check whether the absolute values of the gradients are all smaller than 10^{-15} or whether the absolute values of the relative gradients are all smaller than 10^{-6} (which is set by the global variable '`_cfgchk`'). If either condition is met, then convergence is deemed achieved and the **NLOPT** procedure will terminate after printing the final results, together with the intermediate results from the first five steps, into the ASCII file '`output.out`'.

In Table 2 we present the contents of the '`output.out`' file from the previous example. The intermediate results from each of the first five steps include the computation time, the method for Hessian evaluation (`HESSIAN: BFGS`), the method for step length calculation (`STEP: STEPBT` which means the backtrack algorithm) in the first line. The resulting function value, step length, and the number of trials when implementing the backtrack algorithm are shown in the second line. The final results are as follows: the solution is 0.873580, five iterations have been performed, the entire computation takes 0.22 seconds (this figure depends on the speed of the computer), the minimized function value is -1.175432, and the sum of the absolute values of all relative gradients is less than 10^{-7}.

We make two final points: First, the four outputs of the **NLOPT** procedure can be further processed in the same program. That is, we can add additional GAUSS statements that utilize these outputs, the vector

[2] If on the computer screen a lot of strange symbols are mingled with intermediate results, then you need to insert the following line into the `CONFIG.SYS` file in the root subdirectory:

```
DEVICE = C:\WINDOWS\COMMAND\ANSI.SYS
```

'parout' in particular, after the 'nlopt' command. Secondly, we should understand there is no guarantee that the result we obtain from the **NLOPT** procedure is always the *global* minimum of the objective function. To guard against the possibility of getting a local minimum, all we can do is to try different starting values and see whether we always reach the same solution. If there are more than one solution, we wish to get the global one among those local solutions.

10.2.1 Changing Options

In the previous example, the **NLOPT** procedure is executed in its standard form. The computation methods used there consist of the forward-difference approximation for numerical gradients, the BFGS version of quasi-Newton method for Hessian updates (where the identity matrix is used as the initial value for Hessian), and the backtrack algorithm for the step length. Each of these specifications can be changed and many alternative options are available. To make changes, we simply assign one or more of the following global variables (switches) with different values before calling the **NLOPT** procedure:[3]

1. The global variable '_cfgdmd' determines the algorithm for calculating numerical gradients. Possible values are:

   0: The central-difference approximation;

   1: The forward-difference approximation (the default);

   2: The Richardson extrapolation method.

2. The global variable '_cfhsmd' determines Hessian updating algorithm. Possible values are:

   1: Quasi-Newton method – the BFGS algorithm (the default);

   2: Quasi-Newton method – the scaled BFGS algorithm;

   3: Quasi-Newton method – the DFP algorithm;

   4: Quasi-Newton method – the scaled DFP algorithm;

   5: Newton's method with numerical/analytic Hessian (Hessian is evaluated either numerically by the forward-difference approximation or analytically by user's definition);

   6: The steepest descent algorithm;

3. The global variable '_cfstep' determines the algorithm for calculating the step length. Possible values are:

   1: The step length is fixed at one;

   2: The backtrack method (the default);

   3: The golden search method;

   4: The Brent method.

[3]Besides the various computation methods for numerical gradients, Hessian, and step length that have been described in the previous section, we note that there is one more method for numerical gradients: the Richardson extrapolation method, and one more for the step length: the Brent method. However, both methods are rarely used and will note be discussed here.

Table 2: The Contents of the `output.out` File

```
1  ******************************************************************************
2   ITER:1    TIME:0.00 SEC.  HESSIAN: BFGS   STEP: STEPBT
3   FUNCTION: -1.103638    STEP LENGTH: 0.000   BACKSTEPS: 0
4  ******************************************************************************
5      1   1.00000   1.000000
6  ******************************************************************************
7   ITER:2    TIME:0.00 SEC.  HESSIAN: BFGS   STEP: STEPBT
8   FUNCTION: -1.106689    STEP LENGTH: 0.250   BACKSTEPS: 1
9  ******************************************************************************
10     1   0.75000   0.734375
11 ******************************************************************************
12  ITER:3    TIME:0.00 SEC.  HESSIAN: BFGS   STEP: STEPBT
13  FUNCTION: -1.175431    STEP LENGTH: 1.000   BACKSTEPS: 0
14 ******************************************************************************
15     1   0.87386   0.001898
16 ******************************************************************************
17  ITER:4    TIME:0.00 SEC.  HESSIAN: BFGS   STEP: STEPBT
18  FUNCTION: -1.175432    STEP LENGTH: 1.000   BACKSTEPS: 0
19 ******************************************************************************
20     1   0.87357   0.000102
21 ******************************************************************************
22  ITER:5    TIME:0.00 SEC.  HESSIAN: BFGS   STEP: STEPBT
23  FUNCTION: -1.175432    STEP LENGTH: 1.000   BACKSTEPS: 0
24 ******************************************************************************
25     1   0.87358   0.000000
26 ******************************************************************************
27
28 ******************************************************************************
29           PARAMETER       ESTIMATE          PARAMETER       ESTIMATE
30 ******************************************************************************
31            PAR. 01        0.873580
32 ******************************************************************************
33
34                      NORMAL CONVERGENCE IS ACHIEVED
35
36                              FINAL RESULTS
37                 ************************************************
38                         ITERATION: 5
39                              TIME: 0.22 SECONDS
40                          FUNCTION: -1.175432
41                        REL. GRAD.: 0.0000000
42                 ************************************************
```

4. The global variable '_cfhess0' determines the initial Hessian. Possible values are:

   0: Using the identity matrix as the initial Hessian (the default);

   1: Using the numerical Hessian or the analytic Hessian, evaluated at the starting value given by the vector 'start', as the initial Hessian;

   There is one more possibility for the specification of '_cfhess0': assigning it with a user-defined inverted Hessian matrix.

The default forward-difference approximation is the most efficient algorithm for numerical gradients and it can generally produce quite accurate results. So it is seldom necessary to change it. As to the step length calculation, the default backtrack method is also quite effective so that again it is rarely necessary to change it. Besides, if the backtrack method cannot find an acceptable step length in 15 trials (which is set by the global variable '_cfbktks'), the golden search method will be automatically launched.

Here we offer some suggestions regarding the choice of the Hessian updating algorithms. Choosing numerical/analytic Hessian (i.e., '_cfhsmd = 5') requires the objective function to be smooth (twice differentiable), while quasi-Newton methods generally demand less stringent functional requirements. Moreover, comparing with quasi-Newton methodS, using numerical Hessian (i.e., '_cfhsmd = 5') help reducING the number of iterations while take much longer time for each iteration.

As to the various versions of quasi-Newton method, the default BFGS algorithm appears to be the best. The DFP algorithm could be more stable but requires more iterations and takes longer time to reach convergence. The steepest descent algorithm is best used as the starting method when initial values may be inadequately set. It will begin to perform poorly when approaching the optimum. Also, in using quasi-Newton methods the relative gradients may sometimes bog down without much improvement across iterations. When this happens, a few iterations with numerical/analytic Hessian can usually get the relative gradients going. These discussions all point to a need that we choose one algorithm (e.g., the steepest descent method) before calling the **NLOPT** procedure but change it (to, say, the BFGS algorithm or the numerical Hessian method) in the middle of the procedure execution. This process is the so-called run-time option switching and it is possible when using the **NLOPT** procedure. We will come back to this issue shortly.

10.2.2 Analytic Gradients and Analytic Hessian

Although the **NLOPT** procedure can compute gradients and Hessian numerically, a user should always consider including the analytic gradients and/or analytic Hessian in the program whenever possible. This is because using analytic gradients and/or analytic Hessian can substantially reduce the computation time and in many cases increase computation accuracy.

Similar to the definition of the objective function, the definitions of the analytic gradient and analytic Hessian, if included, are placed in two separate procedures 'gradofn' and 'hessofn'. Furthermore, we should change two global variables accordingly: if the analytic gradient is included, then set '_cfgdofn = &gradofn'; if analytic Hessian is included, then set '_cfhsofn = &hessofn'.

For the example of the objective function $f(\theta) = -3\theta^2 e^{-\theta^3}$, we have the following analytic gradient and analytic Hessian, respectively,

$$g(\theta) = -3\theta(2 - 3\theta^3)e^{-\theta^3} \qquad \text{and} \qquad h(\theta) = -3(2 - 18\theta^3 + 9\theta^6)e^{-\theta^3}.$$

If the analytic gradient is included, then the program is expanded as follows:

```
new;
use nlopt;

proc ofn(t);
  local f;
  f = -3*(t^2)*exp(-t^3);
  retp(f);
endp;

start = 1;

proc gradofn(t);
  local g;
  g = -3*t*(2 - 3*t^3)*exp(-t^3);
  retp(g);
endp;

_cfgdofn = &gradofn;

{parout,vof,gout,retcode} = nlopt(&ofn,start);

end;
```

Note that the global variable '_cfgdofn' is presented before the **NLOPT** procedure is called.

There is an important requirement in the definition of the 'gradofn' procedure which is unfortunately not clearly illustrated in the above example. Recall that the gradient of a function with respect to the vector of independent variables is usually expressed as a column vector in hand-written form. But the gradient vector ('g' in the above example) returned by 'retp' command must nevertheless be a row vector. The above example does not show this requirement in the specification of the 'gradofn' procedure since there is only one independent variable. We will come back to this issue in the next chapter where more examples are presented.

The computation results from this program is almost identical to those in Table 2 (even the intermediate results are all very similar) so that they are omitted here.

To further include the analytic Hessian, we write

```
new;
use nlopt;

proc ofn(t);
  local f;
  f = -3*(t^2)*exp(-t^3);
  retp(f);
endp;
```

```
9
10  start = 1;
11
12  proc gradofn(t);
13    local g;
14    g = -3*t*(2 - 3*t^3)*exp(-t^3);
15    retp(g);
16  endp;
17
18  _cfgdofn = &gradofn;
19
20  proc hessofn(t);
21    local g;
22    g = -3*(2- 18*t^3 + 9*t^6)*exp(-t^3);
23    retp(g);
24  endp;
25
26  _cfhsofn = &hessofn;
27  _cfhsmd = 5;
28
29  {parout,vof,gout,retcode} = nlopt(&ofn,start);
30
31  end;
```

In addition to the procedure '_gradofn' and the global variable '_cfgdofn', we present the procedure '_hessofn' and the global variable '_cfhsofn' before calling the **NLOPT** procedure. We also change the value of the global variable '_cfhsmd' in order to pick Newton's method so that the analytic Hessian defined in the '_hessofn' procedure can be fully utilized. The output file are shown in Table 3. The final results are the same as in Table 2 but the intermediate results are not. In particular, the function values, the variable values, and the corresponding relative gradients in steps 3 and 4 are quite different.

Deriving analytic gradients and including them as a part of the program are strongly encouraged. But in many cases trying to do the same for analytic Hessian can be much more difficult, if not completely intractable. So the best strategy in most applications appears to be including analytic gradients in the program and then applying one of the quasi-Newton methods that use gradients to approximate the inverted Hessian.

This strategy is also applicable to most MLE applications. As mentioned in subsection 10.1.4, in addition to the MLE themselves we often need to compute the corresponding asymptotic variance-covariance matrix, which can be approximated by the inverted Hessian. But the approximated inverted Hessian generated by the quasi-Newton method cannot be used for this purpose. So it is necessary to go one step further to evaluate Hessian numerically using the formula (10.6). To do this, we simply change the value of the global variable '_cfmlese' to 1 by the following statement before calling the **NLOPT** procedure:

```
1  _cfmlese = 1;
```

Table 3: The Contents of the `output.out` File

```
1  ******************************************************************************
2   ITER:1    TIME:0.00 SEC.  HESSIAN: N-R    STEP: STEPBT
3   FUNCTION: -1.103638    STEP LENGTH: 0.000   BACKSTEPS: 0
4  ******************************************************************************
5      1   1.00000   1.000000
6  ******************************************************************************
7   ITER:2    TIME:0.00 SEC.  HESSIAN: N-R    STEP: STEPBT
8   FUNCTION: -1.106690    STEP LENGTH: 0.250   BACKSTEPS: 1
9  ******************************************************************************
10     1   0.75000   0.734375
11 ******************************************************************************
12  ITER:3    TIME:0.00 SEC.  HESSIAN: N-R    STEP: STEPBT
13  FUNCTION: -1.174475    STEP LENGTH: 1.000   BACKSTEPS: 0
14 ******************************************************************************
15     1   0.88797   0.100491
16 ******************************************************************************
17  ITER:4    TIME:0.06 SEC.  HESSIAN: N-R    STEP: STEPBT
18  FUNCTION: -1.175432    STEP LENGTH: 1.000   BACKSTEPS: 0
19 ******************************************************************************
20     1   0.87356   0.000144
21 ******************************************************************************
22  ITER:5    TIME:0.05 SEC.  HESSIAN: N-R    STEP: STEPBT
23  FUNCTION: -1.175432    STEP LENGTH: 1.000   BACKSTEPS: 0
24 ******************************************************************************
25     1   0.87358   0.000000
26 ******************************************************************************
27
28 ******************************************************************************
29           PARAMETER       ESTIMATE          PARAMETER       ESTIMATE
30 ******************************************************************************
31            PAR. 01        0.873580
32 ******************************************************************************
33
34                     NORMAL CONVERGENCE IS ACHIEVED
35
36                              FINAL RESULTS
37                 ************************************************
38                         ITERATION: 5
39                              TIME: 0.22 SECONDS
40                          FUNCTION: -1.175432
41                        REL. GRAD.: 0.0000000
42                 ************************************************
```

The resulting inverse of the numerical Hessian[4] will go to the the global variable '_cfhess1' which can then be further processed. For example, after setting the global variable '_cfmlese' to 1 and calling the **NLOPT** procedure to calculate the MLE, the standard errors of the resulting MLE can be computed (and assigned to variable 'se') by the following statement:

```
1  se = sqrt(diag(_cfhess1));
```

10.2.3 Imposing Restrictions

In this subsection we explain how to impose some common restrictions on the value of an independent variable, say, θ_1 when using the **NLOPT** procedure to minimize the objective function $f(\theta_1, \boldsymbol{\theta}_2)$, where $\boldsymbol{\theta}_2$ is the subvector of other variables. We consider three types of restrictions:[5]

1. $\theta_1 \in (0,\ \infty)$;
2. $\theta_1 \in (0,\ 1)$;
3. $\theta_1 \in (-1,\ 1)$.

The approach we adopt here is based on a transformation of the original variable under restriction to some unrestricted new variable. The idea is to change the restricted minimization problem to an unrestricted one so that we can apply the **NLOPT** procedure as usual.

In the first case where the value of the independent variable θ_1 is restricted to be positive, we consider the following one-to-one transformation from the original variable θ_1 to the new one δ:

$$\theta_1 = k(\delta) \equiv e^{\delta} \in (0,\ \infty). \tag{10.9}$$

We note that the new variable $\delta = \ln\theta_1$ is completely unrestricted. In the second case where the value of the independent variable θ_1 is restricted to be between 0 and 1, we consider the following one-to-one transformation:

$$\theta_1 = k'(\delta) \equiv \frac{1}{e^{-\delta}+1} \in (0,\ 1), \tag{10.10}$$

where $\delta = \ln[\theta_1/(1-\theta_1)]$ can assume any value and is therefore unrestricted. In the third case where the value of the independent variable θ_1 is restricted to be between -1 and 1, we consider the following one-to-one transformation:

$$\theta_1 = k''(\delta) \equiv \frac{e^{\delta}-1}{e^{\delta}+1} \in (-1,\ 1), \tag{10.11}$$

where $\delta = \ln[(1+\theta_1)/(1-\theta_1)]$ is necessarily unrestricted. We note that in each of these three cases, we can rewrite the objective function $f(\theta_1, \boldsymbol{\theta}_1)$ of the original variable as a function of the unrestricted new variable. For example, for the first case we have $\tilde{f}(\delta, \boldsymbol{\theta}_2) \equiv f\left[k(\delta), \boldsymbol{\theta}_2\right]$.

---

[4] If the analytic gradient is not included in the program, then the formula (10.5) will be used. In such a case the computation will take substantially more time and generate much less accurate results.

[5] The upper bound of the second type of restriction as well as the absolute value of the two bounds of the third type of restriction can be extended from 1 to any positive number.

Let us now consider an example of the restricted minimization of the following function

$$f(\theta) = -\theta^7(1-\theta)^4, \qquad \text{for } \theta \in (0,\ 1).$$

Using the transformation (10.10), we rewrite the above function as

$$\tilde{f}(\delta) = f\big[k'(\delta)\big] = -\left(\frac{1}{e^{-\delta}+1}\right)^7\left(\frac{e^{-\delta}}{e^{-\delta}+1}\right)^4,$$

which becomes a function of the unrestricted variable δ. The GAUSS program for unrestricted minimization of this transformed function is

```
new;
use nlopt;

proc ofn(d);
  local t, f;
  t = 1/(exp(-d)+1);
  f = -t^7*(1-t)^4;
  retp(f);
endp;

start = 0;

{parout,vof,gout,retcode} = nlopt(&ofn,start);

theta = 1/(exp(-parout)+1);

format /rd 10,6;
theta;

end;
```

The objective function in the '`ofn`' procedure is defined as a function of '`d`', which represents the unrestricted variable δ. The solution to θ is 0.636364. It is printed in the default output file.

If we intend to include analytic gradients in the program, then we must be careful about the relationship between the gradient with respect to the original variable and the gradient with respect to the new variable. Specifically, we have

$$\tilde{g}_1(\delta, \boldsymbol{\theta}_2) \equiv \frac{\partial \tilde{f}(\delta, \boldsymbol{\theta}_2)}{\partial \delta} = \frac{\partial f(\theta_1, \boldsymbol{\theta}_2)}{\partial \theta_1}\cdot\frac{\partial \theta_1}{\partial \delta} = g_1(\theta_1, \boldsymbol{\theta}_2)\cdot\frac{\partial \theta_1}{\partial \delta} = g_1\big[k(\delta), \boldsymbol{\theta}_2\big]\cdot\frac{\partial k(\delta)}{\partial \delta},$$

where $g_1(\theta_1, \boldsymbol{\theta}_2)$ is the first element of the gradient $\mathbf{g}(\boldsymbol{\theta})$ with respect to the original variables. For the three transformations $\theta_1 = k(\delta)$ in (10.9), $\theta_1 = k'(\delta)$ in (10.10), and $\theta_1 = k''(\delta)$ in (10.11), the corresponding derivatives $\partial\theta_1/\partial\delta$ are, respectively,

$$\frac{\partial k(\delta)}{\partial \delta} = e^{\delta} = k(\delta) = \theta_1,$$

$$\frac{\partial k'(\delta)}{\partial \delta} = \frac{1}{e^{-\delta}+1}\left(1-\frac{1}{e^{-\delta}+1}\right) = k'(\delta)\big[1-k'(\delta)\big] = \theta_1(1-\theta_1),$$

and

$$\frac{\partial k''(\delta)}{\partial \delta} = \frac{1}{2}\left(1+\frac{e^{\delta}-1}{e^{\delta}+1}\right)\left(1-\frac{e^{\delta}-1}{e^{\delta}+1}\right) = \frac{1}{2}\big[1+k''(\delta)\big]\big[1-k''(\delta)\big] = \frac{1}{2}(1+\theta_1)(1-\theta_1).$$

In the above example, the gradient with respect to the original variable is

$$g(\theta) = -\theta^6(1-\theta)^3(7-11\theta),$$

while the gradient with respect to the new variable is

$$\tilde{g}(\delta) = g\big[k'(\delta)\big]\cdot\frac{\partial k'(\delta)}{\partial \delta} = -\theta^6(1-\theta)^3(7-11\theta)\cdot\theta(1-\theta) = -\theta^7(1-\theta)^4(7-11\theta)$$

$$= -\left(\frac{1}{e^{-\delta}+1}\right)^7\left(1-\frac{1}{e^{-\delta}+1}\right)^4\left(7-11\cdot\frac{1}{e^{-\delta}+1}\right)$$

If we include the analytic gradient in the program, then we have

```
new;
use nlopt;

proc ofn(d);
  local t, f;
  t = 1/(exp(-d)+1);
  f = -t^7*(1-t)^4;
  retp(f);
endp;

start = 0;

proc gradofn(d);
  local t, g;
  t = 1/(exp(-d)+1);
  g = -t^7*((1-t)^4)*(7-11*t);
  retp(g);
endp;

_cfgdofn = &gradofn;

{parout,vof,gout,retcode} = nlopt(&ofn,start);

theta = 1/(exp(-parout)+1);

format /rd 10,6;
```

```
27  theta;
28
29  end;
```

If we also want to include analytic Hessian in the program, then we must first examine the relationship between the Hessian with respect to the original variable and the Hessian with respect to the new variable. These analyses, which are somewhat cumbersome, will be omitted here.

10.2.4 Additional Options

The file name of the output file can be changed from the default 'output.out' to, say, 'trial.1st' by the following standard GAUSS command for ASCII output:

```
1  output file = trial.1st reset;
```

The printing of the computation results can be controlled by the global variable '__output', whose possible values are

- 0: Only the final results will go to screen and the output file;
- 1: Iteration number, time, function and relative gradient values, step length, step methods, Hessian algorithm, and the final results will go to both the screen and the output file;
- 2: Besides those of the choice 1, all intermediate variable values and the corresponding relative gradients will go to the screen; those from the first five iterations will also go to the output file (the default).

If we want to specify the variable names in the final printouts (the default are 'PAR. 01', 'PAR. 02', 'PAR. 03', ...), we can assign the global variable '_cfvarnm' with a character vector of the desired variable names.

If for some reasons the program is terminated before reaching convergence, then it may be useful to retain those intermediate variable values and the corresponding inverted Hessian right before the program stops. Inspecting these intermediate results might help determine why the program is abnormally terminated. They can also be used as the initial values (i.e., by assigning them to the global variables 'start' and '_cfhess0', respectively) to restart the program. The **NLOPT** procedure always keeps the latest intermediate variable values and the corresponding inverted Hessian in the global variables '_cfintvv' and '_cfhess1', respectively.

10.2.5 Run-Time Option Switching

Many options can be changed while the **NLOPT** procedure is running. In Table 4 we list the key commands for all run-time option switching. The **NLOPT** procedure will respond to these key commands at the end of each iteration. For example, no matter which algorithm for step length calculation is set in the program,

Table 4. Run-Time Option Switching

	Key	Effect	Explanation
1.	ALT 1	Set _cfhsmd = 1	Quasi-Newton method – the BFGS update
2.	ALT 2	Set _cfhsmd = 2	Quasi-Newton method – the scaled BFGS update
3.	ALT 3	Set _cfhsmd = 3	Quasi-Newton method – the DFP update
4.	ALT 4	Set _cfhsmd = 4	Quasi-Newton method – the scaled DFP update
5.	ALT 5	Set _cfhsmd = 5	Numerical/analytic Hessian
6.	ALT 6	Set _cfhsmd = 6	The steepest descent method
7.	SFT 1	Set _cfstep = 1	Step length fixed at 1
8.	SFT 2	Set _cfstep = 2	The backtrack method
9.	SFT 3	Set _cfstep = 3	The golden search method
10.	SFT 4	Set _cfstep = 4	The Brent method
11.	0	Set __output = 0	Output control: option 0
12.	1	Set __output = 1	Output control: option 1
13.	2	Set __output = 2	Output control: option 2
14.	ALT C	Force Convergence	Exit program immediately
15.	ALT G	Change _cfgdmd	Change gradient method
16.	ALT N	Change _cfmlese	Whether to evaluate final inverted Hessian
17.	ALT V	Change _cfgchk	Change gradient convergence criterion
18.	ALT M	Set _cfbktks	Change the number of trials allowed in step length search
19.	ALT I		Compute Hessian immediately
20.	ALT E		Edit variable values*
21.	<PgUp>		Print the previous 54 parameters**
22.	<PgDn>		Print the next 54 parameters**

* Pressing 'ALT E' brings the program into an interactive mode under which we can alter variable values.
** When the number of variables is greater than 54 so that not all intermediate results can be shown on one screen, then using '<PgUp>' and '<PgDn>' keys can control which 54 variables to show.

pressing 'SFT 3' (i.e., pressing 'Shift' and '3' simultaneously) while the program is running switches the algorithm to the golden search method.[6] [7]

Running NLOPT Recursively: The procedures 'OFN', 'GRADOFN', 'HESSOFN' themselves can call the **NLOPT** procedure. The number of nested levels is limited only by the amount of computer memory. Each

[6] If the program is taking too many iterations using the default backtrack method, trying golden search for a few iterations this way (and then pressing 'SFT 2' to switch back to the backtrack method) may help speed up convergence.

[7] When we press 'ALT E' and enter into an interactive modeare, then the top two lines on the screen will list current variable values one at a time that is ready for changes. We can then do one of the following: (1) press 'CURSOR UP' and 'CURSOR DOWN' to move along the variable list back and forth; (2) Press 'ENTER' to select the variable for editing, type the new value, and then press 'ENTER'; (3) Press 'Q' or 'q' to quit. As soon as a variable value is changed, Hessian will be reset to the identity matrix if any of the quasi-Newton methods is being used.

level contains its own set of global variables. However, the run-time option switches can be used only at one level of the **NLOPT** procedure, where the value of the corresponding global variable '_cfrtos' should be 1 (the default) while those for other levels should all be 0.

10.2.6 Global Variable List

We summarize the global variables used by the **NLOPT** procedure in Tables 5 and 6. Each of these global variables controls one aspect of the procedure and can be changed before calling the **NLOPT** procedure. Those in Table 5 are the ones that have been discussed in the previous subsections.

There are still a few control variables that are less important and have not yet been discussed. They are presented in Table 6.

The global variable '_cfusrch' needs some explanations. It allows us to set the step length manually during the run-time. Setting the value of the global variable '_cfusrch' to 1 before calling the **NLOPT** procedure causes the program to enter into an interactive mode if all methods for calculating step length fail. Under such an interactive mode, lines 6 to lines 25 on the screen show the initial function value as well as a new function value with a step length of 1. A number 0.1 listed as 'Stepsize Change' will also appear. This is a value to be used as an increment for changing the value of current step length. We can do one of the following:

- Press 'CURSOR UP' to raise the 'Stepsize Change' 10 times;
- Press 'CURSOR DOWN' to reduce the 'Stepsize Change' 10 times;
- Press '+' to increase the step length by the amount of 'Stepsize Change' and to recompute the function value;
- Press '-' to decrease the step length by the amount of 'Stepsize Change' and to recompute the function value.

After pressing '+' or '-', a new function value will be computed based on the new step length. The difference between this new function value and the initial one, together with the corresponding new step length, will be listed in light white color if the new function value is lower than the initial one, and in dark gray color otherwise. Step lengths can be repeatedly tried and 20 trial results will be shown on the screen simultaneously for comparisons. When a step length with a lower function value occurs (i.e. a light white color appears on the screen), press 'Q' to exit the interactive mode. The last step length used will be accepted by the **NLOPT** procedure as a new step length and the iteration will continue.

Table 5. The List of Global Variables

	Variable	Default	Explanation
1.	_cfgdmd	1	Switch for numerical gradient evaluation algorithms. It can be changed during the run time by pressing 'ALT G'.
2.	_cfhsmd	1	Switch for Hessian evaluation algorithms. It can be changed during the run time by pressing 'ALT 1 -- 5'.
3.	_cfstep	2	Switch for step length evaluation algorithms. It can be changed during the run time by pressing 'SFT 1 -- 4'.
4.	_cfgdofn	0	Switch for analytic gradients.
5.	_cfhsofn	0	Switch for analytic Hessian.
6.	_cfhess0	0	Switch for initial Hessian.
7.	_cfhess1	0	Storage for the final inverted Hessian.
8.	_cfintvv	0	Storage for the final variable values.
9.	_cfmlese	0	Switch for the inverted Hessian after reaching convergence. It can be changed during the run time by pressing 'ALT N'.
10.	_cfvarnm	0	Storage for variable names.
11.	__output	2	Switch for output printing. It can be changed during the run time by pressing '0', '1', or '2'.
12.	_cfxchk	10^{-5}	Convergence criterion for the changes in successive variable values.
13.	_cfgchk	10^{-6}	Convergence criterion for the relative gradients. It can be changed during the run time by pressing 'ALT V'.
14.	_cfbktks	15	Maximum number of trials allowed in searching for step length using the backtrack, golden search, and Brent methods. It can be changed during the run time by pressing 'ALT M'.
15.	_cfrtos	1	Switch for the run-time option switching. The value 0 turns this feature off while 1 turns it on. If the **NLOPT** procedure is being run recursively, (i.e., the **NLOPT** procedure is being called inside of another **NLOPT** procedure), then the run-time switch feature should be turned off for the inner version of the **NLOPT** procedure so that the outer version can retain the control.

Table 6. The List of Additional Global Variable

	Variable	Default	Explanation
1.	_cfiter	10^5	Maximum number of iterations allowed for computation.
2.	_cftime	10^5	Maximum time in minutes allowed for computation.
3.	_cfsltry	100	Maximum number of trials allowed in computing step length based on the golden search and Brent methods.
4.	_grdh	0	Increment used in computing numeric gradients. If it is set to zero, then the increment will be computed automatically.
5.	_cfeiglb	0.1	Lower bound for Hessian eigenvalues when Hessian is evaluated. The eigenvalues of the Hessian matrix will be forced to be greater than this value. If it is set to zero, then such a constraint will be disabled.
6.	_cfradus	0	Radius of random direction. When it is set to a nonzero value (10^{-2}, say) and all other step length search methods fail, then a random direction with radius determined by _cfradus will be tried without further search for step length based on the previous direction. If it is set to the default value 0, then this random direction generating mechanism will not start on its own.
7.	_cfusrch	1	Switch for user-controlled step length search. If it is set to a nonzero value and if all other step length search methods fail, then the **NLOPT** procedure will enter into an interactive mode under which the user can select step length directly.

Appendix A

Drawing Graphs for the Simple Linear Regression Model

In this appendix we discuss a GAUSS program that contains the most commonly used commands for drawing graphs. We demonstrate, in the case of the simple regression model, how to put sample points into a graph and draw the corresponding regression line. The usage of each command in the program is briefly explained in the comment that follows.

The basic structure of this program is to specify a number of "feature commands" (commands started with the two letters '_p') before the main command

```
1  xy(x,y);
```

which appears as the very last command in the program. Each of those feature commands defines one aspect, such as the size, color, line type, legends, messages, etc. of the graph.

The main body of the graph is either a bunch points or a curve that connects these points. The (x, y) coordinates of these points are specified by the two input 'x' and 'y' in the 'xy(x,y)' command. The common row number of 'x' and 'y' tells us the number of points on the graph, while their column number indicates the number of different sets of points or the number of different curves. If one of 'x' and 'y' is a single column vector while the other is a matrix, then the single column will be expanded automatically to match the other matrix (the usual element-by-element operation).

The order of the feature commands in the program is not important. If a particular feature is not needed, then the corresponding feature command can be omitted and its default value will be adopted by GAUSS. If n sets of points or n curves are drawn, many of the feature commands will then contain n *rows* of specifications: each *row* specifies the feature of the corresponding set of points or curve.

There are six relatively independent components of the graphs:

1. The general specifications about graph title, axis system, and some miscellaneous details.
2. The specification of the main curves.
3. The specification of the legends which help identify different curves.
4. The specification of auxiliary lines or curves besides the main curves.
5. The specification of messages.
6. The specifications of auxiliary arrows and symbols.

Auxiliary lines, messages, arrows, and symbols are used to highlight certain part of the graph to make it easier to understand.

The kind of graphs we create with the 'xy(x,y)' command in the following program is called the 'XY' graph. GAUSS can make other graphs such as bar charts, histograms, contours, and 3-dimensional graphs. Once we are familiar with the following program for the 'XY' graph, exploring other types of graphs will be fairly straightforward.

If the version of GAUSS used is 2.2 or lower, then before this graph drawing program can be executed we need to configure GAUSS as follows: the program 'equip' in the subdirectory '\GAUSS' has to be run once to inform GAUSS the specifications of the computer and the printer. This program will create a file with the name 'config.gpc', which is generally to be stored in the '\GAUSS' subdirectory. We also need to add the following line to the file 'autoexec.bat' which is in the root subdirectory.

```
set gpc=c:\gauss
```

The 'autoexec.bat' file is an ASCII file. If the version of GAUSS used is 3.0 or higher, then we only need to configure GAUSS by modifying the ASCII file 'pqgrun.cfg' in the '\GAUSS' subdirectory to specify the computer and the printer we use.

Before using the following program, we must run the linear regression program for the Engel Curve model in chapter 5 with the following command added at the end of that program (before the 'end' command):

```
x = x[.,2];                          /*  We do not need the constant term. */

save x, y, b, e, seb, s2, r2;
```

Similarly, if the 'ols' command is used, then the following commands should be part of the program:

```
_olsres = 1;

{vnam,mmt,b,stb,vb,seb,s2,cor,r2,e,d} = ols(0,y,x);

save x, y, b, e, seb, s2, r2;
```

These commands save seven matrices from the estimation in the matrix file format. These matrix data will then be loaded in the following graph program.

```
new;
/*****************************************************************
 *           Drawing Graph for the Simple Regression Model       *
 *****************************************************************/
library pgraph;      /*  Calling the graph program.               */
graphset;            /*  Resetting graphics globals to default values.   */
                     /*  These commands are discussed in chapter 9.      */

load x, y, b, seb, s2, r2; /*  These six matrices must have been
```

```
                                produced and saved earlier.                */

_pframe = 0|1;
/*************************************************************************
 *  Row 1 -- 0: frame off,          Row 2 -- 0: tick marks off,          *
 *           1: frame on,                    1: tick marks on.           *
 *************************************************************************/

_pbox = 0;
/*************************************************************************
 *  0: box off,                                                          *
 *  n: box on, with desired color.                                       *
 *************************************************************************/

_pgrid = 0|2;
/*************************************************************************
 *  Row 1 -- 0: no grid,         2: fine dotted grid,                    *
 *           1: dotted grid,     3: solid grid,                          *
 *  Row 2 -- 0: no subdivision,  2: tick marks only at subdivisions.     *
 *           1: dotted line at subdivisions,                             *
 *************************************************************************/

fonts("simplex complex");
/*************************************************************************
 *  The input of the 'fonts' commands is a string between the quotation  *
 *  marks. The contents of this string may include up to four font       *
 *  types which control the style of the letters in the graph. The four  *
 *  fonts are                                                            *
 *                                                                       *
 *      Simplex, (standard san serif font),                              *
 *      Simgrma (math notation and Greek letters),                       *
 *      Microb (bold face font),                                         *
 *      Complex (standard serif font).                                   *
 *                                                                       *
 *  A simple way to find out how these character printed by your         *
 *  printer is to try out all the keyboard keys (94 keys in total,       *
 *  except the arrows keys and the function keys) in all four fonts. Any *
 *  number of these four fonts may be included in the FONT STRING of the *
 *  above 'fonts' command. The order of font names in the FONT STRING    *
 *  defines the FONT INDICATORS that will be referenced later whenever   *
 *  fonts is needed. The FONT INDICATORS in order are '\201', '\202',    *
 *  '\203', and '\204'. The first font specified will be used for the    *
 *  axes numbers. If the 'fonts' command is not used, the SIMPLEX font   *
 *  is used by default.                                                  *
 *************************************************************************/
```

```
55
56  if b[1,1] >= 0;                                /* This section of codes  */
57     st1 = ftos(b[1,1],"Y = %*.*lf ",1,3);       /* constructs a self-     */
58     st4 = ftos(seb[1,1],"   (%*.*lf)  ",1,3);   /* explanatory three-line */
59  else;                                          /* title for the graph    */
60     st1 = ftos(-b[1,1],"Y = - %*.*lf ",1,3);    /* that summarizes the    */
61     st4 = ftos(seb[1,1],"     (%*.*lf)  ",1,3); /* estimation results of  */
62  endif;                                         /* a simple linear        */
63                                                 /* regression model.      */
64  if b[2,1] >= 0;
65     st2 = ftos(b[2,1],"+ %*.*lf X",1,3);
66  else;
67     st2 = ftos(-b[2,1],"- %*.*lf X",1,3);
68  endif;
69
70  st3 = ftos(r2,"   R[2] = %*.*lf",1,3);
71  st5 = ftos(seb[2,1],"(%*.*lf) ",1,3);
72  st6 = ftos(s2,"   s[2] = %*.*lf",1,3);
73
74  title("\202A Simple Regression Model\l"        /*  The title string is    */
75        $+ st1 $+ st2 $+ st3 $+ "\l "            /*  concatenation of 8     */
76        $+ st4 $+ st5 $+ st6);                   /*  strings defined above. */
77  /*************************************************************************
78   *  The input of TITLE command is a string which may contain up to 3     *
79   *  lines of titles up to 180 characters:                                *
80   *                                                                       *
81   *  1. Title string starts with one of the 4 font indicators \201,       *
82   *     \202, \203 or \204, which are defined by the 'fonts' command.     *
83   *     Fonts can be altered in the middle of the string by changing      *
84   *     font indicators.                                                  *
85   *                                                                       *
86   *  2. The multi-line title is separated by \l. For example:            *
87   *         TITLE("\201LINE 1\l LINE 2\l LINE 3");                        *
88   *                                                                       *
89   *  3. Some basic formats:                                               *
90   *         ]a[     subscript a,                                          *
91   *         [a]     superscript a,                                        *
92   *         ]a[[b]  subscript a and superscript b.                        *
93   *                                                                       *
94   *  4. Embedding numbers in the string: We first use the 'ftos' command  *
95   *     to transform numbers to a string, and then concatenate the        *
96   *     result to the title string: "\20?...." $+ string $+ "....";       *
97   *************************************************************************/
98
99  _ptitlht = 0.15;
```

```
/*************************************************************************
 *  The size of the title in inches (0: default, 0.13).                  *
 *************************************************************************/

_paxes = 1;
/*************************************************************************
 *  0: axes off,   1: axes on.                                           *
 *************************************************************************/

_pcross = 0;
/*************************************************************************
 *  0: axes intersect at corner,                                         *
 *  1: axes intersect at (0,0).                                          *
 *************************************************************************/

xlabel("\202Log Income");
ylabel("\202Budget Share for Food");
/*************************************************************************
 *  The inputs for 'xlabel' and 'ylabel' commands are strings whose      *
 *  formats follows the same rule as the title string does. These two    *
 *  commands determine the labels for the X axis and the Y axis.         *
 *************************************************************************/

_paxht = 0;
/*************************************************************************
 *  The size of axes labels in inches (0: default, 0.13).                *
 *************************************************************************/

_pnum = 2;
/*************************************************************************
 *  0: no ticks marks and numbers on axes,                               *
 *  1: vertically-oriented numbers on Y axis,                            *
 *  2: horizontally-oriented numbers on Y axis.                          *
 *************************************************************************/

_pnumht = 0;
/*************************************************************************
 *  The height of axis numbers in inches (0: default, 0.13).             *
 *************************************************************************/

_pxpmax = 3;  _pypmax = 3;
/*************************************************************************
 *  The numbers of decimal points of the Y and Y axes numbers.           *
 *************************************************************************/

```

```
/*   xtics(min,max,step,div);
     ytics(min,max,step,div); */
/*************************************************************************
 *  To define scaling, axes numbering and tick marks for X and Y axes:   *
 *    min:  the minimum value,                                           *
 *    max:  the maximum value,                                           *
 *    step: the value between major tics,                                *
 *    div:  the number of subdivisions.                                  *
 *************************************************************************/

asclabel(0,0);
/*************************************************************************
 *  To set up character labels for the X and Y axes. It requires two     *
 *  character vectors as inputs. If any input is 0, then character       *
 *  labels will not be used for that axis.                               *
 *    input 1: labels for the tick marks on the X axis,                  *
 *    input 2: labels for the tick marks on the Y axis.                  *
 *************************************************************************/

/*************************  -- MAIN CURVES -- ***************************/

_plctrl = -1;
/*************************************************************************
 *  Line control (may contain multiple rows and one row for each curve)  *
 *     0: line only (default),                                           *
 *     n: ( > 0) line and symbols at every n points,                     *
 *     n: ( < 0) symbols only at every n points,                         *
 *    -1: symbols only at every point.                                   *
 *************************************************************************/

_pltype = 0;
/*************************************************************************
 *  Line type (one row for each line)                                    *
 *    0: default,                                                        *
 *    1: dashed,     3: short dashed,      5: dots and dashes,           *
 *    2: dotted,     4: densed dots,       6: solid.                     *
 *************************************************************************/

_plwidth = 0;
/*************************************************************************
 *  Line thickness: 0 (normal) or greater, one row for each line        *
 *************************************************************************/

_psymsiz = 3;
/*************************************************************************
```

```
 *  Symbol size: 1-9 (may have decimals) (if 0, default 5 is used)       *
 ************************************************************************/

_pstype = 1;
/************************************************************************
 *  Symbol type (one row for each line)                                 *
 *    1: circle,     6: reverse triangle,  11: solid plus,              *
 *    2: square,     7: star,              12: solid diamond,           *
 *    3: triangle,   8: solid circle,      13: solid reverse triangle,  *
 *    4: plus,       9: solid square,      14: solid star,              *
 *    5: diamond,   10: solid triangle,     0: default.                 *
 ************************************************************************/

_pcolor = 4;
/************************************************************************
 *  Line color (one row for each line)                                  *
 *    0: black,  4: red,      8: grey,        12: light red,            *
 *    1: blue,   5: Magenta,  9: light blue,  13: light magenta,        *
 *    2: green,  6: brown,   10: light green, 14: yellow,               *
 *    3: cyan,   7: white,   11: light cyan,  15: bright white.         *
 ************************************************************************/

_pmcolor = 15;
/************************************************************************
 *  A row of nine numbers to define color for:                          *
 *      row 1: axes,           row 4: y axis label,  row 7: box,        *
 *      row 2: axes numbers,   row 5: z axis label,  row 8: date,       *
 *      row 3: x axis label,   row 6: title,         row 9: background. *
 *  If scalar, then it will be expanded to a 9x1 vector.                *
 ************************************************************************/

/*************************  -- LEGEND --  ******************************/

_plegstr = "";
/************************************************************************
 *  Multiple legend strings are separated by \000                       *
 *       _plegstr = "\20?legend1\000legend2\000legend3";                *
 *  See title string for other formatting possibilities.                *
 ************************************************************************/

_plegctl = 0;
/************************************************************************
 *  Legend control can a scalar, or 4 element vector.                   *
 *    1. Scalar: 0 -- no legend (default),                              *
 *               1 -- a legend will be created based on _plegstr. The   *
```

```
235     *                    legend box will be placed in the lower right-hand  *
236     *                    corner just inside the axes area.                  *
237     *    2. 4 element vector:                                               *
238     *       row 1: 1 -- in plot coordinates,                                *
239     *              2 -- in inches (9.0 x 6.855),                            *
240     *              3 -- in pixels (3120 x 4096),                            *
241     *       row 2: font size -- 1-9, can have decimals (5 default),         *
242     *       row 3: x location of the lower left corner of legend box,       *
243     *       row 4: y location of the lower left corner of legend box.       *
244     ************************************************************************/
245
246   /*************************  -- MESSAGES --  *************************/
247
248   _pmsgstr = "";
249   /************************************************************************
250    *  Multiple massage strings are separated by \000                      *
251    *       _msgstr = "\20?MASSAGE1\000MASSAGE2\000MASSAGE3";              *
252    *  See title string for other format considerations.                   *
253    ************************************************************************/
254
255   _pmsgctl = 0;
256   /************************************************************************
257    *  Massage control: one row for each message                           *
258    *    column 1: x location of lower left corner,                        *
259    *    column 2: y location of lower left corner,                        *
260    *    column 3: massage height in inches,                               *
261    *    column 4: rotation in degrees,                                    *
262    *    column 5: 1 -- plot coordinates; 2 -- inches,                     *
263    *    column 6: color,                                                  *
264    *    column 7: font thickness, 0 (normal) or greater.                  *
265    ************************************************************************/
266
267   /***********************  -- AUXILIARY LINES --  ***********************/
268
269   _pline = 1~6~                                  /* The fitted regression */
270          minc(x[.,2])~(b[1]+minc(x[.,2])*b[2])~  /* line is drawn on top  */
271          maxc(x[.,2])~(b[1]+maxc(x[.,2])*b[2])~  /* of the sampled points.*/
272          1~14~0;
273   /************************************************************************
274    *  Line definitions: one row for each line                             *
275    *    column 1: 1 -- line in plot coordinates,                          *
276    *              2 -- line in inches (9.0x6.855),                        *
277    *              3 -- line in pixel(3120x4096),                          *
278    *              4 -- circle in plot coordinates,                        *
279    *              5 -- circle in inches (9.0x6.855),                      *
```

```
 *             6 -- radius in plot coordinates,                          *
 *             7 -- radius in inches (9.0x6.855),                        *
 *    column 2: line type (see above),                                   *
 *    column 3: x location of the starting point,                        *
 *    column 4: y location of the starting point,                        *
 *    column 5: x location of the end point,                             *
 *    column 6: y location of the end point,                             *
 *    column 7: 0 -- continuation; 1 -- new,                             *
 *    column 8: color,                                                   *
 *    column 9: line thickness, 0 (normal) or greater.                   *
 *  If 0, no line will be drawn.                                         *
 ************************************************************************/

/***********************  -- AUXILIARY SYMBOLS --  ***********************/

_psym = 0;
/************************************************************************
 *  Extra symbol definitions: one row for each symbol                   *
 *    column 1: x location of the center of the symbol,                 *
 *    column 2: y location of the center of the symbol,                 *
 *    column 3: symbol type,                                            *
 *    column 4: symbol size -- 1-9, can have decimals (5 default),      *
 *    column 5: symbol color,                                           *
 *    column 6: 1 -- inches; 2 -- plot coordinates,                     *
 *    column 7: line thickness, 0 (normal) or greater.                  *
 ************************************************************************/


/***********************  -- AUXILIARY ARROWS --  ***********************/

_parrow = 0;
/************************************************************************
 *  Arrow definitions: one row for each arrow                           *
 *    column 1: x location of the starting point,                       *
 *    column 2: y location of the starting point,                       *
 *    column 3: x location of the end point,                            *
 *    column 4: y location of the end point,                            *
 *    column 5: ratio of the arrow head length to its half width,       *
 *    column 6: size of arrow head in inches,                           *
 *    column 7: fn -- type of arrow head,                               *
 *              f is for form:                                          *
 *                   0 -- solid,                                        *
 *                   1 -- empty,                                        *
 *                   2 -- open,                                         *
 *                   3 -- closed,                                       *
```

```
 *             n is for number:                                          *
 *                  0 -- none,                                           *
 *                  1 -- at the final end,                               *
 *                  2 -- at both ends,                                   *
 *    column 8: color,                                                   *
 *    column 9: 1 -- in plot coordinates,                                *
 *              2 -- in inches (9.0x6.855),                              *
 *              3 -- in pixels (3120x4096),                              *
 *    column 10: line type (see above),                                  *
 *    column 11: line thickness, 0 (normal) or greater.                  *
 *  If 0, no arrow will be drawn.                                        *
 ************************************************************************/


/*************************  -- MAIN COMMAND -- **************************/

_pdate = "";
/************************************************************************
 *  A small label on the top left corner (default: \201GAUSS).          *
 *  If the string is "", then nothing will be printed.                  *
 ************************************************************************/

_ptek = "food_s.tkf";
/************************************************************************
 *  TKF output file's name with .TKF extension. The file can be         *
 *  transported and saved for later printing.  no graphics file will be  *
 *  produced if "" is set.  Default is GRAPHIC.TKF.                     *
 ************************************************************************/

xy(x,y);
/************************************************************************
 *  This command directs GAUSS to draw the two dimensional graph. The   *
 *  two inputs are data on X and Y variables. If data on X and Y        *
 *  variables are multiple columns, then multiple curves will be drawn.  *
 ************************************************************************/

end;
```

After succeeding in creating a graph on the screen, we can press the ‘space bar’ to get printing selections on the screen. These choices are self-explanatory.

Let’s consider another example where we show how to draw the scatter diagram for the residuals as opposed to the log income, which is the explanatory variable of the Engel curve model. Such a diagram is interesting because we often want to know whether there is a relationship between the residual and the explanatory variable in a linear regression model. If there is, then some modification of the OLS estimation may be needed. Here we only present seven commands that are required to be modified from the previous

program;

```
load x, e;
title("\202 The Scatter Diagram for Residuals");
_pcross = 1;
ylabel("\202Residuals");
_ptek = "residual.tkf";
_pline = 0;
xy(x[.,2],e);
```

Also, the '`if...else...endif`' statements before the '`title`' command can all be deleted.

The GAUSS graph drawing capability is powerful and flexible. But with many user-friendly and even more powerful graphic softwares available in the market, we must say that the quality of the GAUSS graphs is not particularly outstanding. Sometimes it seems to be a good idea to use GAUSS to draw a sketchy draft and then export the data to some other graphic softwares for the final version of the graph.

Appendix B

GAUSS Data Set (GDS) Files

We have introduced two types of data files in chapter 2, ASCII data files and matrix files. We have also learned that they are mainly for smaller data sets. To store large data sets in GAUSS, we may need the third type of data files – the GAUSS data set (GDS) files. In this section we explain how to use the GDS files. Here let's first briefly list the main features of the GDS files:

1. GDS files are formulated in matrices: rows for observations and columns for variables.
2. It is possible to assign a name, the variable name, to each column and store these "variable names" along with data. The column number of a GDS file is limited only by the size of computer memory, while the row number is limited only by the size of the hard disk.
3. Each GDS is stored in a pair of files with the same file name but different extensions '`.dat`' and '`dht`'. The file with the extension '`.dht`' is called the header file, in which the variable names are stored.
4. Data can be stored in three levels of precision: 2 (integer precision), 4 (single precision), or 8 (double precision) bytes per number. The precision level of 2 bytes per number is ideal for integers while the double precision is used for most real numbers. Note that matrix files are always stored in double precision, which can sometimes be wasteful in terms of disk storage. For example, a 5000×100 matrix of integers requires 4,000,000 bytes, or 4 MB, of disk space if it is stored as a matrix file. But it can be stored as a GDS file at the integer precision with only 1 MB of memory.

B.1 Writing Data to a New GAUSS Data Set

We can create a GDS file using the trio: the '`create...with`', the '`writer`', and the '`close`' commands. For example, suppose we have a set of data in a 100×5 matrix '`xx`' left in the memory. We want to create a GDS file with the file name '`out1`" to store those 100 observations on 5 variables at the single precision level, using the names '`id`', '`name`', '`age`', '`var1`', and '`region`', respectively. To do this, we first put the 5 variable names in a character vector '`varname`':

```
let varname = id name age var1 region;
```

and then type the following commands:

```
create fout = out1 with ^varname,5,4;
check1 = writer(fout,xx);
check2 = close(fout);
```

We now explain these three commands in detail:

1. The 'create...with' command:

   (1) Both 'fout' and 'out1' on the two sides of the equality sign are file names for the same output file to be created. The name 'out1' to the right of the equality sign can be regarded as an "external" file name because it is the file name recorded in the hard disk but is never referred to inside the GAUSS program. The name 'fout' to the left of the equality sign can be considered as the "internal" file name because it is used as a reference label exclusively inside the program. 'fout' is also called *file handle*.

   (2) We can use the technique of the caret '^' sign plus the string variable to specify the file name. For example, the following statement assigns a string to the variable 'filename'

   ```
   filename = "c:\\example\\data\\out1";
   ```

   The content of the variable 'filename' is a long string that specifies the external output file name with drive and subdirectory information. Note that, as mentioned before, the backslash '\' should be replaced by a double backslash '\\' everywhere in the string. The string variable 'filename' can be used in the 'create...with' command as follows:

   ```
   create fout = ^filename with ^varname,5,4;
   ```

   The caret sign means that 'filename' itself is not the external file name. Instead, it is the content of the variable 'filename' that specifies the file name.

   (3) The names of variables are in the character vector 'varname' which is included after the 'with' subcommand. We note 'varname' is preceded by the caret sign '^', which again informs GAUSS the name 'varname' itself is not the variable name but a character vector that contains the variable names.

   (4) The first number that follows 'varname' (and a comma) indicates the number of variables that are going to be created.

   We may wonder why we need to tell GAUSS explicitly the number of variables while it is already obvious from the 5×1 character vector 'varname' that there are five variables. The reason for such an extra effort is that the number of variables to be created may not be the same as the number of variable names in the character vector 'varname'. If the specified variable number, say 3, is smaller than the number of names in 'varname', then only the first three variable names will be used. If the specified variable number, say 10, is larger, then the last variable name in 'varname', i.e., 'region', will be duplicated and the last 6 of the 10 variables will have the sequential names 'region1', 'region2', ..., 'region6', respectively. So if we have the following 'create...with' command:

   ```
   create fout = out1 with Y,15,8;
   ```

   then the GDS file 'out1' will contain 11 variables, whose names are 'Y01', 'Y02', ..., 'Y15', respectively. Note that the label 'Y' is not preceded by the caret sign so that 'Y' itself is the variable name.

(5) The last number in the '`create...with`' command indicates the precision level, which can be 2 (integer), 4 (single precision), or 8 (double precision). Double precision must be adopted if data contain characters.

(6) Multiple '`create...with`' commands can be used in the same program so that data may be written into different GDS files in the same program.

2. The '`writer`' command:

(1) The '`writer`' command writes the data in the matrix '`xx`', row by row, to the GDS file, which is referred to by the name '`fout`' (instead '`out1`').

(2) After the values in the '`xx`' are completely written into '`fout`', a scalar '`check1`' is produced to indicate the total number of rows that have just been written.

(3) The column number of the matrix "`xx`' must be the same as the variable number specified in the '`create...with`' command.

(4) Multiple '`writer`' commands can be used after the same '`create...with`' command. In such a case, data will be written consecutively into the same output file.

(5) If the precision level is set at 2 and the source matrix '`xx`' contains missing values, which are usually denoted by a dot '`.`' in GAUSS, then these missing values will all change to the value -32768 automatically. At other levels of precision, missing values will be recorded as missing values.

3. The '`close`' command closes the file '`fout`' and creates a scalar '`check2`' to indicate whether the file is successfully closed: 0 for success and -1 for failure.

   The '`close`' command can be skipped since the '`end`' command, which is usually placed at the very end of the program, will close all the created files. The advantage of using '`close`' command immediately after the file creation is that we can close the file as early as possible. If a file is not closed and the program is terminated abnormally due to, say, power failure, then the data in the file will be lost.

   A related command is

```
closeall;
```

   which closes all files. A list of internal file names (file handles) can also follow the '`closeall`' command, in which case only the listed files will be closed. So

```
check2 = close(fout);
```

   and

```
closeall fout;
```

   have the same effect.

In the above discussions we find the 'create...with' command is the most complicated one that specifies a lot of information. Sometimes it is easier to first put all information in a separate (ASCII) file and then refer to it. To do this, we use the following GAUSS command:

```
create fout = out1 using varspec;
```

Here, instead of the 'with' subcommand, we have the 'using' subcommand and all the information specified after 'with' in the 'create...with' command is now contained in an ASCII file whose file name is 'varspec'.
There are usually two commands inside the ASCII file such as 'varspec':

```
numvar 12 vv;
outtype 4;
```

where the 'numvar' command specifies the number of the variables and their names. In the above example, twelve variable names will be created: 'vv01', 'vv02', ..., 'vv12'. So 'vv' in the 'numvar' command indicates the prefix of variable names which will be followed by consecutive numbers. The 'outtype' command specifies the precision level which can be 2, 4, or 8. If we want to define exact variable names, then we should use the 'outvar' command, instead of the 'numvar' command:

```
outvar id name age var1 region;
outtype 4;
```

Here, five different variable names are specified.

B.2 Reading the GAUSS Data Set

We can read a GDS file using the trio 'open', 'readr', and 'close' commands. For example, suppose we want to read a GDS file with the file name 'out1.dat" which contains 100 observations on 5 variables, whose names are 'id', 'name', 'age', 'var1', and 'region', respectively, we type

```
open fin = out1;
xx = readr(fin,100);
check1 = close(fin);
```

The 'open' command specifies the GDS file 'out1.dat' to be read. The specification of file name is similar to that of the 'create...with' command, the one on the right-hand side of the equality sign 'out1' is the external file name of 'out1.dat' (without the extension '.dat') which is used by DOS to store it in the disk, while the one on the left-hand side is the internal file name (or the file handle) 'fin' which is used by GAUSS within the program. Also, as explained by the item 1 (2) in the previous subsection, the external file name can be stored as a string in a variable and then be referred to using the "caret" technique.

The 'readr' command reads rows of data from the specified file. It requires two inputs: the internal name of the file and the number of rows in the file to be read. The output is a matrix whose row number should of course be the same as the row number as is specified in the 'readr' command. The column number is decided by the number of variables contained in the source GDS files. In the above example, the output matrix 'xx' will be a 100×5 matrix. Note that even though there are 100 rows in the original GDS files, we do not have to read them all. We may specify 50, for example, in the above 'readr' command. In such a case only the first 50 rows will be read into the 'xx' matrix.

The 'close' command closes the opened file and indicates whether the closing is successful. See item 3 in the previous subsection for more details.

B.2.1 Using Variable Names

Since the GDS file records variable names, it is possible to refer to each column of the output matrix by the corresponding variable name. In the above example, we know the five columns of the output matrix 'xx' are observations on the five variables 'id', 'name', 'age', 'var1', and 'region', respectively. We can then refer to the five columns of 'xx' by 'iid', 'iname', 'iage', 'ivar1', and 'iregion', respectively. Here, we note these variable names are all prefixed by 'i'. These variable names are useful when we want to select some columns from 'xx' for some manipulation. To use this feature of the GDS file, we should add the 'varindxi' subcommand at the end of the 'open' command. For example,

```
open fin = out1 varindxi;
xx = readr(fin,100);
check1 = close(fin);
x1 = xx[.,iname iregion iage];
x2 = xx[.,2 5 3];
```

With the 'varindxi' subcommand, we then have a better idea about the contents of the 100×3 submatrix 'x1'. It contains observations on the three variables 'name', 'region', and 'age' (in that order). We note the submatrix 'x2' is identical to 'x1'. The creation of 'x2' is based on the equivalent but less appealing method of indexing.

If we forget the names of the variables in a GDS file, we can use the 'getname' command to retrieve the variable names. For example, to get the variable names from the GDS file 'out1', we use

```
name = getname("out1");
$name;
```

Here, we note the external file name of the GDS is put inside the quotation marks. The output 'name' will be a character vector containing the variable names. They are printed for viewing.

B.3 Reading and Processing Data with Do-Loops

Since the GDS file is mostly used to store a large number of data, trying to read all data from a large GDS file into a matrix often causes the insufficient memory problem. We note each scalar requires 8 bytes of

memory. If a GDS file contains 10,000 observations on 50 variables, then we need 4 million bytes (4,000 KB or 4 MB) memory to create such a 10000×50 matrix. In a PC that is equipped with 4 MB memory, the chances are that there is only several hundred KB memory left to run GAUSS programs so that it is not possible to read all data into one big matrix. But this does not mean that the GDS files is of little use in GAUSS programming because in most GAUSS applications we do not really need to read all data in one huge matrix before we can process them. We are usually able to partition the data of the GDS file, read each portion of the data into a smaller matrix, and then work on one matrix at a time. The following example illustrates this point.

Suppose we have a GDS file '`source.dat`' that stores 30,000 observations on 25 variables. We want to compute the OLS estimate in a linear regression model with the last variable as the dependent variable and the first 24 variables as the non-constant explanatory variables. To accomplish this job with limited amount of memory, we need to reexamine the OLS formula $\mathbf{b} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$ where in the present example $\mathbf{X}$ is $30{,}000 \times 25$ matrix containing 30,000 observations on a constant term and 24 explanatory variables, while $\mathbf{y}$ is $30{,}000 \times 1$ vector of the dependent variable. Let's partition both $\mathbf{X}$ and $\mathbf{y}$ vertically into, say, 100 submatrices, each of which contains 300 rows:

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \\ \vdots \\ \mathbf{X}_{100} \end{bmatrix} \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_{100} \end{bmatrix}.$$

Then we note

$$\mathbf{b} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} = \left(\sum_{k=1}^{100} \mathbf{X}_k'\mathbf{X}_k\right)^{-1} \left(\sum_{k=1}^{100} \mathbf{X}_k'\mathbf{y}_k\right).$$

Here, $\mathbf{X}_k$ and $\mathbf{y}_k$ are 300×25 and 300×1 matrices, respectively. Their cross products $\mathbf{X}_k'\mathbf{X}_k$ and $\mathbf{X}_k'\mathbf{y}_k$ are 25×25 and 25×1 matrices, respectively. The sizes of these four matrices are moderate and they require 68 KB memory in total. Based on this formula, we can then compute $\mathbf{X}_k'\mathbf{X}_k$ and $\mathbf{X}_k'\mathbf{y}_k$ separately and repeatedly, and eventually obtain the OLS estimate $\mathbf{b}$ without demanding too much memory. This idea can be implemented as follows:

```
1  open ff = source;
2
3  k = 1;
4  xx = 0;
5  xy = 0;
6
7  do until k > 100;
8
9     dat = readr(ff,300);
10    x = ones(300,1)~dat[.,1:24];
11    y = dat[.,25];
12    xx = xx + x'x;
13    xy = xy + x'y;
```

```
14      k = k + 1;
15
16    endo;
17    check = close(ff);
18
19    b = invpd(xx)*xy;
```

The key idea here is the use of a do-loop.

Although the logic of such a do-loop is straightforward, we may have the question: how does the 'readr' command works inside the do-loop? To answer this question, we have to know an important concept in using the 'readr' command.

B.3.1 The 'readr' and the 'writer' Commands and Do-Loop

After each row of the GDS file is read, GAUSS always prepares to read the next row or, we might say, GAUSS places a reading pointer at the next row for the next reading. So after the first iteration of the above do-loop and one hundred rows of the GDS file 'ff' (or 'source.dat') are read into the output matrix 'dat', GAUSS places the reading pointer at the 101-th row of the GDS file even though that row is not immediately read. After the first cycle is completed and GAUSS is prepared to execute the 'readr' command for the second time, GAUSS will then follow the reading pointer and read the 101-th row of the GDS file.

In the above example the decision of partitioning the GDS file into 100 parts is somewhat arbitrary. We note the computation time of a do-loop is directly proportional to the number of cycles to be iterated. If the number of cycles is large, then the execution of the do-loop can be quite time consuming and inefficient. So it is sometimes necessary to experiment with different partitions of the GDS file and we should always push toward the limit of memory by minimizing the number of partitions. For example, if we have more than 136 KB of memory, then we can cut the computation time in half by reducing the partition of the GDS file 'ff' to 50 parts, in which case each $\mathbf{X}_k$ will be a 200×25 matrix.

Suppose after some further experiments we find it is possible to increase the row number of the submatrix $\mathbf{X}_k$ to 450. But in such a case the corresponding number of partition is 67 while the last submatrix contains 300 rows only which is different from that of other 66 submatrices. Nevertheless, the do-loop can still be used as usual. It should be noted that in the last iteration, the matrix 'dat', as well as 'x' and 'y', will contain 300 rows only. Here, we will be using a special feature of the 'readr' command: it automatically terminates when the reading pointer passes over the last row of the GDS file 'ff'.

The above program can be somewhat simplified by a new GAUSS command 'eof': the command line that starts the do-loop:

```
1   do until k > 100;
```

can be replaced by

```
1   do until eof(ff);
```

The command 'eof' indicates whether the reading pointer passes the end of GDS file 'ff' or not. When it does, then the 'eof(ff)' returns the value 1 and the do-loop terminates.

All the above discussions on how the 'readr' command interacts with do-loop can be extended to the 'writer' command.

B.3.2 The 'seekr' Command

There is a GAUSS command "seekr' that allows us to move the reading pointer to any row of the GDS file we want. This command is useful when we want to read a part of the GDS file more than once or when we want to skip some part of the GDS file. For example, given the above GDS file 'source.dat', we need the first 100 and the last 100 observations on the first variable, and the first 200 observations on the last variable, then the GAUSS program for retrieving these data is as follows:

```
open ff = source;

dat = readr(ff,100);
x1 = dat[.,1];

rn1 = seekr(ff,29900);
dat = readr(ff,100);
x1 = x1|dat[.,1];

rn2 = seekr(ff,1);
dat = readr(ff,200);
x3 = dat[.,25];
```

After the first 'readr' command, the reading pointer is placed at the 101-th row of the GDS file 'ff'. But the 'seekr' command moves the reading pointer to the 29,900-th row of the GDS file 'ff' so that the last 100 rows can then be read directly. There is an output from the 'seekr' command which indicates the location of the reading point after the 'seekr' command is executed. It is usually the same as the second input of the 'seekr' command, so the value of 'rn1' should be 29,900. Note that the second 'seekr' command moves the reading pointer back to the first row of the GDS file 'ff' and the value of 'rn2' is 1.

Note that if the 'seekr' command is specified as 'rn3 = seekr(ff,0)', then the reading pointer is moved *after* the last row of the GDS file 'ff' and the value of 'rn3' is 30,001.

B.4 GAUSS Commands That Are Related to GDS Files

Given that a GDS file is opened and is assigned an internal file name (file handle) 'ff', then we can use the following GAUSS commands:

```
rn = rowsf(ff);    /*  The number of rows of the file 'ff'.            */
cn = colsf(ff);    /*  The number of columns of the file 'ff'.         */

pl = typef(ff);    /*  The value of 'pl' is either 2, 4, or 8, which is
                       the precision level of the data in the file 'ff'.*/
```

If we want to get the variable names from a GDS file without first opening it, we can use

```
vname = getname("source");
```

The input of this command is a string containing the name of GDS file and the output is a character vector of the variable names.

B.4.1 Sorting the GDS File

In subsection 8.3 we mentioned the GAUSS command 'sortc' that sorts a matrix (rearrange the order of the rows) according the values of one of its column. If we want to sort a GDS file directly, then use

```
sortd("source","sortout","VAR11",-1);
```

Here, the first input indicates the GDS file that is to be sorted. The sorted file will be another GDS file whose name is given by the second input. The third input specifies the column number or the variable name whose values will be used as the "key" for ordering. Note that the first three inputs are all included in quotation marks so that they are all strings. If the third input is the column number, then it can be a scalar (a number not inside quotation marks). The last input can take four possible values: 1 and -1 mean the values of the key variable are numeric, while 1 (-1) causes the rows to be the sorted in ascending (descending) order; 2 and -2 mean the key is a character variable, while 2 (-2) causes the rows to be the sorted in ascending (descending) order. Note that the 'sortd' command does not have matrix output.

B.4.2 The 'ols' Command and the GDS File

When we first discussed the 'ols' command in subsection 16.8, we mentioned it took three inputs while there we set the first input to 0 without explanation. Now with the knowledge of GDS files, we are ready for more discussions on the 'ols' command. We can directly use the data in a GDS file for the OLS estimation. Suppose the GDS file we want to use is 'source.dat' again. Let's assume the 25 variables in this data set have the variable names 'VAR01', 'VAR02', ..., 'VAR25', respectively. If we want to run an OLS estimation with 'VAR07' as the dependent variable and 'VAR12', 'VAR06', 'VAR21', 'VAR17', and 'VAR01' as regressors, then the GAUSS commands are

```
rgssor = {"VAR12","VAR06","VAR21","VAR17","VAR01"};
{vnam,mmt,b,stb,vb,seb,s2,cor,r2,e,d} = ols("source","VAR07",rgssor);
```

or, equivalently,

```
rgssor = {12, 6, 21, 17, 1};
{vnam,mmt,b,stb,vb,seb,s2,cor,r2,e,d} = ols("source",7,rgssor);
```

That is, when the first input is a string that shows the name of the GDS file to be used, then the second input is either a string containing the name of the dependent variable or a scalar indicating the index of the dependent variable, while the third input is either a character vector containing the variable names of the regressors or a numeric vector indicating the indices of the regressors. Note that it is possible to set the second input and the third input to 0: if the second input is 0, then the last variable in the GDS file is the dependent variable. If the third input is 0, then all variables in the GDS file, except the one that has been designated as the dependent variable, will be used as the regressors.

There are two more issues when the 'ols' command is applied to a GDS file. Since the GDS file is usually very large, the 'ols' command will automatically use a do-loop to process the data during the OLS estimation and the way the data set being partitioned is determined by the size of memory. Occasionally, the insufficient memory problem may still occur. In such a situation, we may need to manually decide how many rows to be processed in each cycle of the do-loop execution. Such a row number will then be assigned to the global variable '__row' before the execution of the 'ols' command.

Finally, we note it is quite common for a large GDS file to have missing values. The 'ols' command can not process a GDS file with missing value unless we specifically tell GAUSS how to deal with missing values using the global variable '__miss' (whose default is 0). If we set '__miss' to 1, then all of those rows with missing values will be dropped from the estimation. But if we set '__miss' to 2, then all missing values will be replaced by 0 before the computation and no row will be dropped.

B.5 Revising GDS Files

If we want to modify an existing GDS file, then we have to specifically indicate that when we open this GDS file by adding a subcommand in the 'open' command. For example, the command

```
open ff = source for update;
```

will open the GDS file 'source.dat' and place the reading pointer at the first row. We can then use either the 'readr' command to read data from the file or the 'writer' command to write data into it. If we only want to add some data at the end of an existing GDS file, then we use

```
open ff = source for append;
```

then the GDS file 'source.dat' is opened but the reading pointer is placed after the last row of the file so that new data may be directly written to the end of the file with the 'writer' command. Note that in this case the GDS file cannot be read and the 'readr' command is not applicable.

B.6 Reading and Writing Small GDS Files

If the GDS file 'source.dat' is small enough to be included in one matrix without causing insufficient memory problem, we can read it into a matrix directly using

```
dat = loadd("source");
```

Since we do not need "open" the file first, this command is much simpler than the standard way of reading the GDS file.

Similarly, if we want to store an existing matrix, say 'dat' which a 500 × 5 matrix, into a GDS file 'ggout.dat' and assign some variable names to the five columns of the matrix, then we type

```
check = saved(dat,"ggout","VAR");
```

The first input is the name of the matrix to be stored. The second input is a string specifying the output GDS file name. The third input can be 0, a string (as in the above example), or a character vector. If it is 0, then the five variable names will be 'X1', 'X2', . . ., 'X5'. In the above example, the five variable names will be 'VAR1', 'VAR2', . . ., 'VAR5'. If a five specific variable names are to be adopted, then they should be included in character vector and plugged as the third input of the 'saved' command. (The way variable names are assigned is similar to that in the 'create...with' command.)

The output 'check' is a scalar. It is 1 if the saving process is successful and is 0 otherwise. Also, the precision level of the resulting GDS file is double precision.

B.7 The ATOG Program*

Most large-scale socio-economic and financial data downloaded from mainframe computers are stored in ASCII files. Observations are usually arranged in rows and variables are separated (delimited) either by space or by commas. It is also common to find, in some "packed" ASCII files, variables are not delimited and can be identified only based on their column positions. These ASCII data files are usually so large that they can be accessed by GAUSS only as the GDS files. Therefore, the first thing we need is a utility program to convert those ASCII data files to GDS files. The name of such a program is "ATOG.EXE" (ATOG means "**A**SCII files **t**o **G**auss data set files.")

To use the ATOG program, we have to first create a small ASCII file, which is called the command file, to describe the structure of the source ASCII data file as well as how we want the output GDS file arranged. The name of this command file usually has the extension '.cmd'.

Suppose the source ASCII file, 'demodata', contains data on 6 variables: Age, Sex, Income, Marital [Status], School[ing Years], and Region, among which Sex, Marital, and Region are character variables and the other three are numeric variables. Observations are stored in rows. That is, each row contains an observation on each of these six variables which are delimited by spaces. Note that in an ASCII file different rows are separated by hidden codes called carriage returns or CR. So we may say observations are delimited by CR. We want to create a GDS file that only contains data on Age, School, and Income at the double precision level of 8 bytes per number. To do these, we type the command file, say, 'demo.cmd', as follows:

```
input a:\demodata;
output a:\out1;
invar # age $ sex # income $ marital # school $ region;
outvar age school income;
outtyp d;
```

The ‘input’ command specifies the name and the location of the source ASCII file, which in the present example contains 6 columns of data. The ‘output’ command specifies the name of the output GDS files, which causes two files ‘out1.dat’ and ‘out1.dht’ to be created.

The ‘invar’ command specifies the names and the types (character or numeric) of the variables in the source ASCII file. Note that the variable names are not part of the source ASCII file but arbitrarily assigned by us here in the command file. Variable names are preceded by either ‘$’ or ‘#’ depending on whether the variable is character or numeric. The indicators ‘$’ and ‘#’ can be omitted, in which case the type of variable is the same as the previous one. If none of the indicator is specified at all, then all variables are considered numeric.

The ‘outvar’ command specifies the variables to be included in the output GDS files and the output variables can be in any designated order. The ‘outtyp’ command specifies the precision level of the output GDS. Three options are available: ‘d’ for double precision, ‘f’ for single precision (which is the default), and ‘i’ for integer precision.

If in the ‘invar’ and ‘outvar’ commands the variable names are in sequence, such as the seven variables ‘rec01’, ‘rec02’, ..., ‘rec07’, then these sequential variable names can be abbreviated as ‘rec[7]’. Also, three sequential variable names like ‘xx4’, ‘xx5’, and ‘xx6’ can be abbreviated as ‘xx[4:6]’.

To execute the ATOG program in the GAUSS command mode, we type

```
atog demo;
```

Note that the file extension ‘.cmd’ is not needed in the above command. But if the name of the command file has an extension different from ‘.cmd’, then the extension needs to be fully specified.

B.7.1 The Structure of the Source ASCII file

Because the data in the source ASCII file may be arranged in a variety of formats, we will have to provide all the necessary information to the ATOG program through the ‘invar’ command. More specifically, variables in the source ASCII file can be delimited in three different ways so that the corresponding ‘invar’ command has three basic formats:

1. If the variables in the source ASCII file are delimited by spaces, commas, or CR, then we only need a simple ‘invar’ command that specifies variable names together with their types (# for numeric values and & for characters). The above example illustrates such a case.

   (1) Strictly speaking, observations do not have to be listed in rows and delimited by CR. For example, in the above example each row may contain 18, instead of 6, values. With 6 variable names being specified in the ‘invar’ command, the ATOG program will treat the 18 values in each row as three sets of observations. That is, different observations do not have to be delimited by CR so that they appear in different rows. Spaces that separate values are treated just the same as the CR that separate rows and vice versa.

   (2) Multiple spaces or commas with no data in between will be ignored and considered as a single space or comma.

   (3) Missing values should be recorded as a period ‘.’.

2. If the variables in the source ASCII file are delimited by some characters (which are called the delimiter and can be any printable characters including commas), then we add the 'delimit' subcommand immediately after the 'invar' command to inform the ATOG program what character is used as the delimiter. When the 'delimit' subcommand is used, then any pair of delimiters with no values in between imply a missing value which is equivalent to a period '.' between the pair of delimiters.

   The 'delimit' subcommand allows two inputs. Let's consider the following examples.

   (1) If the delimiter is ';' and we expect one delimiter after every variable, then we type

   ```
   1  invar delimit(;n) zz[5];
   ```

   Here, the first input ';' of the 'delimit' subcommand specifies the delimiter. The second input 'n' of the 'delimit' subcommand indicates one delimiter after every variable. So there is a delimiter even after the last variable in each row. In the above example where five variables are specified in the 'invar' command, five ';' are expected in each row. If a row contains less than five ';', then that row is considered incomplete and will *not* be included in the output GDS file.

   (2) If the delimiter is '*' and we do not expect a delimiter after the last variables, then we type

   ```
   1  invar delimit(*) zz[5];
   ```

   Here, the first input '*' of the 'delimit' subcommand specifies the delimiter. There is no second input for the 'delimit' subcommand, which implies that there is a delimiter between two variables but no delimiter is expected after the last variable. So with five variables specified in the 'invar' command, only four delimiters are expected in each row.

   (3) If the delimiter is ',' and we do not expect a delimiter after the last variables, then we type

   ```
   1  invar delimit zz[5];
   ```

   Here, there is no input for the 'delimit' subcommand since the character ',' happens to be the default delimiter. Also, with five variables specified in the 'invar' command, only four commas are expected in each row.

   (4) If *each row only contains one value only* while blank rows are considered missing, then we give the 'delimit' subcommand the special input '\n':

   ```
   1  invar delimit(\n) zz[5];
   ```

   (5) There is another special input '\r' for the 'delimit' subcommand:

   ```
   1  invar delimit(\r) zz[5];
   ```

   which specifies the following: there are five values in each row, the delimiter is comma ',', and no comma after the last variable is expected.

3. We need add the ‘record’ subcommand immediately after the ‘invar’ command if the source ASCII file is packed; that is, all rows in the source ASCII file have the same number of characters and the variables in each row are not delimited by any delimiter and can only be specified by the column position in each row.

   The use of the ‘record’ subcommand can be best explained through an example. Suppose the length of each row is 45, i.e., the total number of characters in each row is 45. The typical row looks as follows:

```
          3656510970892837 92800ADEF GLKAMXZ02 84555790<CR>
          |   |    |    |    |    |    |    |    |    | |
position: 1   5   10   15   20   25   30   35   40   45 46
```

   Note that there is a carriage return (CR) after the 45-th character.

   A possible ‘invar’ command with the ‘record’ subcommand is

```
invar record = 46 #(35,2.0) var1 $(28,4) var4 $(23,4) var10 #(5,7.3)
               var2;
```

   The length of the row, including the carriage return (sometime the formatting code for “line feed”, if any, may also need to be counted as another character), is indicated after the subcommand ‘record=’, which is then followed by a list of variable names, preceded by their respective specifications. The value of the variable ‘var1’ is numeric and starts at the 35-th position, with 2 digits and 0 after the decimal point, i.e., its value is ‘2’. The value of the variable ‘var4’ is character and starts at the 28-th position, with the 4 characters ‘GLKA’. The value of the variable ‘var10’ is character and starts at the 23-rd position, with the 4 characters ‘ADEF’. The value of the variable ‘var2’ is numeric and starts at the 5-th position, with 7 digits and 3 after the decimal point, i.e., its value is ‘5109.708’. As the output variables are fully specified by the ‘record’ subcommand, the ‘outvar’ command is not needed in the present case.

   Consider another example

```
      FGLPWSDFG890345  5678 2890   9089475 ASEDPPGFHVVC 233490856<CR><LF>
      |   |    |    |    |    |    |    |    |    |    |    |    | |  |
Cols: 1   5   10   15   20   25   30   35   40   45   50   55   60 61 62
```

   and

```
invar record = 62 $(1,3) c[3] #(30,3.1) n[2] #(*,1,0) x $(38,4)
               vr[2] $(*,2) p1 p3 #(51,3.2) a b c;
```

   Here, the length of the row is 62, including ‘CR’ and the formatting code for line feed LF. The values of the variables ‘c1’, ‘c2’, and ‘c3’ are ‘FGL’, ‘PWS’, and ‘DFG’, respectively, since the same

specification '`$(1,3)`' will be applied to all '`c[3]`'. Similarly, the values of the variables '`n1`' and '`n2`' are '90.8' and '94.7', respectively. The asterisk in the next specification '`#(*,1,0)`' indicates the starting position is right next to the last position for the previous variable, which is the 36-th position in the present case. So the value of the variable '`x`' is 5. With the same rule, we can find the values of '`vr1`', '`vr2`', '`p1`', '`p3`', '`a`', '`b`', and '`c`' are '`ASED`', '`PPGF`', '`HV`', '`VC`', '23.3', '49.0', and '85.6', respectively.

There are two more commands in the command file that may be useful. The command

```
append;
```

instructs the ATOG program to append the data to an existing GDS file without erasing the data already there. The command

```
msym &;
```

defines the character '`&`' as the missing value character. Note that the dot '`.`' is the default.