A Hansl Primer

The scripting language of gretl in 55 minutes

Allin Cottrell
Department of Economics
Wake Forest University

Riccardo (Jack) Lucchetti
Dipartimento di Scienze Economiche e Sociali
Università Politecnica delle Marche

September, 2023

Contents

Chapter 1

Introduction

What hansl is and what it is not

Hansl is a recursive acronym: it stands for "Hansl's A Neat Scripting Language". You might therefore expect something very general in purpose. Not really. Hansl was born as the scripting language for the econometrics program gretl and its role is unlikely to change. As a consequence, hansl should not be viewed as a fully fledged programming language such as C, Fortran, Perl or Python. Not because it lacks anything to be considered as such,[1] but because its aim is different. Hansl should be considered as a special-purpose or domain-specific language, designed to make an econometrician's life easier. Hence it incorporates a series of conventions and choices that may irritate purists and have some marginal impact on raw performance, but that we, as professional econometricians, consider "nice to have". This makes hansl somewhat different from plain matrix-oriented interpreted languages, such as the Matlab/Octave family, Ox and so on.

On the other hand, hansl is by no means just a tool to automate rote tasks. It has several features which support advanced work: structured programming, recursion, complex data structures, and so on. As for style, the language which hansl most resembles is probably that of the bash shell.

The intent and structure of this document

The intended readers of this document are those who already know how to write code, and are familiar with the associated do-s and don't-s. Such people may wish to add hansl to their toolbox, alongside languages like C or FORTRAN, or programs such as R, Ox, Matlab, Stata or Gauss, some of which they are already confident with. Here, therefore, the focus is not so much on "How do I do this?", but rather on "How do I do this *in hansl*?".

As a consequence, this document aims at making the reader a reasonably proficient hansl user in a (relatively) short time; however, not all the features of hansl are illustrated; for those, interested readers should consult the *Gretl Command Reference* and the *Gretl User's Guide*.

This guide comprises two main parts. Part I ("Without a dataset") concentrates on hansl as a pure matrix-oriented programming language. Part II ("With a dataset") exploits the fact that hansl scripts are run through gretl, which has very nice facilities for handling statistical datasets (interactively if necessary). This provides hansl with a series of extra constructs and features which make it extremely easy to write hansl scripts to perform all sorts of statistical procedures.

In order to use hansl, you will need a working installation of gretl. We assume you have one. If you don't, please refer to chapter 1 of the *Gretl User's Guide*.

Other resources

If you are serious about learning hansl then after working through this primer—or in the process of doing so—you'll want to take a look at the following additional resources.

- The *Gretl Command Reference*. This contains a complete listing of the commands and built-in functions available in hansl, with a full account of their syntax and options. Examples of usage

[1] Hansl is in fact Turing-complete.

are provided in some instances. This is available in an "online" version for handy reference as well as in PDF, both accessible via the Help menu in the gretl GUI.

- The *Gretl User's Guide*. Chapters 10 to 16, in particular, go into more detail on some of the programming topics discussed here (data types, loops, the definition and use of functions). In addition Part II of the *Guide*, on Econometric Methods, gives many examples of hansl usage. The *Guide* is available via gretl's Help menu; the latest version can also be found online at http://sourceforge.net/projects/gretl/files/manual/.
- Sample scripts. The gretl package comes with a large number of sample or practice scripts, which can be found under the menu item /File/Script files/Example scripts. Many of these are simple replication exercises for textbook problems but you will find some more interesting examples under the Gretl tab.
- Function packages. Relatively ambitious examples of hansl coding can be found in the various contributed "function packages". You can download these packages via the gretl menu item /Tools/Function packages/On server. Once a package is downloaded it appears in the listing under /Tools/Function packages/On local machine; in that context you can right-click and select View code to examine the hansl functions.
- The gretl-users mailing list. Most well-considered questions get answered quite quickly and fully. See https://gretlml.univpm.it/postorius/lists/.

Chapter 2

For the impatient

OK, so you're impatient. Then perhaps you're thinking "Do I really need to go through the whole thing? After all, I've been coding econometric stuff for a while, and I'm pretty confident I can pick a new scripting language if it's not too obscure. I just need a few tips to get me started". If that's not what you're thinking at all, we suggest you move along to chapter 3; but if it is, well then, we'll give you a hansl script which exemplifies a hefty share of the topics discussed in the rest of this primer. We will use for our example a Vector AutoRegressive model, or VAR for short.

As you probably know, a finite-order VAR can be estimated via conditional maximum likelihood simply by applying OLS to each equation individually. That amounts to solving a least-squares problem and its solution can be easily written, in matrix notation, as $\hat{\Pi} = (X'X)^{-1}X'Y$, where Y contains your endogenous variables and X contains their lags plus other exogenous terms (typically, a constant term at least). But of course, you may choose to find the maximum of the concentrated likelihood $\mathcal{L} = -(T/2)\ln|\hat{\Sigma}|$ numerically if you so wish.

The following example contains a hansl script which performs these actions:

1. Reads data from a disk file.
2. Performs some data transformation and simple visualization.
3. Estimates the VAR via
   (a) the native hansl `var` command
   (b) sequential single-equation OLS
   (c) matrix algebra (in 3 different ways)
   (d) numerical maximization of the log-likelihood.
4. Prints out the results.

The script also contains some concise comments.

```
open AWM.gdt --quiet                    # load data from disk

/* data transformations and visualisation */

series y = 100 * hpfilt(ln(YER))        # the "series" concept: operate on
series u = 100 * URX                    # vectors on an element-by-element basis
series r = STN - 100*sdiff(ln(HICP))    # (but you also have special functions)

scatters y r u --output=display         # command example with an option: graph data

/* in-house VAR */

scalar p = 2                            # strong typing: a scalar is not a
                                        # matrix nor a series

var p y r u                             # estimation command
```

```
A = $coeff                              # and corresponding accessor

/* by iterated OLS */

list X = y r u                          # the list is yet another variable type

matrix B = {}                           # initialize an empty matrix

loop foreach i X                        # loop over the 3 var equations
    ols $i const X(-1 to -p) --quiet    # using native OLS command
    B ~= $coeff                         # and store the estimated coefficients
endloop                                 # as matrix columns

/* via matrices */

matrix mY = { y, r, u }                 # construct a matrix from series
matrix mX = 1 ~ mlag(mY, {1,2})         # or from matrix operators/functions
mY = mY[p+1:,]                          # and select the appropriate rows
mX = mX[p+1:,]                          # via "range" syntax

C1 = mX\mY                              # matlab-style matrix inversion
C2 = mols(mY, mX)                       # or native function
C3 = inv(mX'mX) * (mX'mY)               # or algebraic primitives

/* or the hard, needlessly complicated way --- just to show off */

function scalar loglik(matrix param, const matrix X, const matrix Y)

    # this function computes the concentrated log-likelihood
    # for an unrestricted multivariate regression model

    scalar n = cols(Y)
    scalar k = cols(X)
    scalar T = rows(Y)
    matrix C = mshape(param, k, n)
    matrix E = Y - X*C
    matrix Sigma = E'E

    return -T/2 * ln(det(Sigma))
end function

matrix c = zeros(21,1)                  # initialize the parameters
mle ll = loglik(c, mX, mY)              # and maximize the log-likelihood
    params c                            # via BFGS, printing out the
end mle                                 # results when done

D = mshape(c, 7, 3)                     # reshape the results for conformability

/* print out the results */

# note: row ordering between alternatives is different

print A B C1 C2 C3 D
```

If you were able to follow the script above in all its parts, congratulations. You probably don't need to read the rest of this document (though we don't mind if you do). But if you find the script too scary, never fear: we'll take things step by step. Read on.

Part I

Without a dataset

Chapter 3

Hello, world!

We begin with the time-honored "Hello, world" program, the obligatory first step in any programming language. It's actually very simple in hansl:

```
# First example
print "Hello, world!"
```

There are several ways to run the above example: you can put it in a text file `first_ex.inp` and have gretl execute it from the command line through the command

```
gretlcli -b first_ex.inp
```

or you could just copy its contents in the editor window of a GUI gretl session and click on the "gears" icon. It's up to you; use whatever you like best.

From a syntactical point of view, allow us to draw attention on the following points:

1. The line that begins with a hash mark (#) is a comment: if a hash mark is encountered, everything from that point to the end of the current line is treated as a comment, and ignored by the interpreter.
2. The next line contains a *command* (`print`) followed by an *argument*; this is fairly typical of hansl: many jobs are carried out by calling commands.
3. The quotation character is a straight double-quote.
4. Hansl does not have an explicit command terminator such as the "`;`" character in the C language family (C++, Java, C#, ...) or GAUSS; instead it uses the newline character as an implicit terminator. So at the end of a command, you *must* insert a newline. Conversely, you can't split a single command over more than one line unless (a) the line to be continued ends with a comma or (b) you insert a "\" (backslash) character, which causes gretl to ignore the following line break.

Note also that the `print` command automatically appends a line break, and does not recognize "escape" sequences such as "`\n`". Such sequences are just printed literally—with a single exception, namely that a backslash immediately followed by double-quote produces an embedded double-quote. The `printf` command can be used for greater control over output; see chapter 5.

Let's now examine a simple variant of the above:

```
/*
  Second example
*/
string foo = "Hello, world"
print foo
```

In this example, the comment is written using the convention adopted in the C programming language: everything between "`/*`" and "`*/`" is ignored.[1] Comments of this type cannot be nested.

Then we have the line

[1] Each type of comment can be masked by the other:

```
string foo = "Hello, world"
```

In this line, we assign the value "`Hello, world`" to the variable named `foo`. Note that

1. The assignment operator is the equals sign (=).
2. The name of the variable (its *identifier*) must follow the following convention: identifiers can be at most 31 characters long and must be plain ASCII. They must start with a letter, and can contain only letters, numbers and the underscore character.[2] Identifiers in hansl are case-sensitive, so `foo`, `Foo` and `F00` are three distinct names. Of course, some words are reserved and can't be used as identifiers (however, nearly all reserved words only contain lowercase characters).
3. The string delimiter is the double quote (").

In hansl, a variable has to be of one of these types: `scalar`, `series`, `matrix`, `list`, `string`, `bundle` or `array`. As we've just seen, string variables are used to hold sequences of alphanumeric characters. We'll introduce the other ones gradually; for example, the `matrix` type will be the object of the next chapter.

The reader may have noticed that the line

```
string foo = "Hello, world"
```

implicitly performs two tasks: it *declares* `foo` as a variable of type `string` and, at the same time, *assigns* a value to `foo`. The declaration component is not strictly required. In most cases gretl is able to figure out by itself what type a newly introduced variable should have, and the line `foo = "Hello, world"` (without a type specifier) would have worked just fine. However, it is more elegant (and leads to more legible and maintainable code) to use a type specifier at least the first time you introduce a variable.

In the next example, we will use a variable of the `scalar` type:

```
scalar x = 42
print x
```

A `scalar` is a double-precision floating point number, so `42` is the same as `42.0` or `4.20000E+01`. Note that hansl doesn't have a separate variable type for integers.

An important detail to note is that, contrary to most other matrix-oriented languages in use in the econometrics community, hansl is *strongly typed*. That is, you cannot assign a value of one type to a variable that has already been declared as having a different type. For example, this will return an error:

```
string a = "zoo"
a = 3.14 # no, no, no!
```

If you try running the example above, an error will be flagged. However, it is acceptable to destroy the original variable, via the `delete` command, and then re-declare it, as in

```
scalar X = 3.1415
delete X
string X = "apple pie"
```

- If `/*` follows `#` on a given line which does not already start in ignore mode, then there's nothing special about `/*`, it's just part of a `#`-style comment.
- If `#` occurs when we're already in comment mode, it is just part of a comment.

[2] Actually one exception to this rule is supported: identifiers taking the form of a single Greek letter. See chapter 15 for details.

There is no "type-casting" as in C, but some automatic type conversions are possible (more on this later).

Many commands can take more than one argument, as in

```
set verbose off

scalar x = 42
string foo = "not bad"
print x foo
```

In this example, one `print` is used to print the values of two variables; more generally, `print` can be followed by as many arguments as desired. The other difference with respect to the previous code examples is in the use of the `set` command. Describing this command in detail would lead us to an overly long diversion; suffice it to say that it is used to set the values of various "state variables" that influence the behavior of the program; here it is used as a way to silence unwanted output. See the *Gretl Command Reference* for more on `set`.

The `eval` command is useful when you want to look at the result of an expression without assigning it to a variable; for example

```
eval 2+3*4
```

will print the number 14. This is most useful when running gretl interactively, like a calculator, but it is usable in a hansl script for checking purposes, as in the following (rather silly) example:

```
scalar a = 1
scalar b = -1
# this ought to be 0
eval a+b
```

3.1 Manipulation of scalars

Algebraic operations work in the obvious way, with the classic algebraic operators having their traditional precedence rules. The caret (^) is used for exponentiation. For example,

```
scalar phi = exp(-0.5 * (x-m)^2 / s2) / sqrt(2 * $pi * s2)
```

in which we assume that `x`, `m` and `s2` are pre-existing scalars. The example above contains two noteworthy points:

- The usage of the `exp` (exponential) and `sqrt` (square root) functions; it goes without saying that hansl possesses a reasonably wide repertoire of such functions. See the *Gretl Command Reference* for the complete list.
- The usage of `$pi` for the constant π. While user-specified identifiers must begin with a letter, built-in identifiers for internal objects typically have a "dollar" prefix; these are known as *accessors* (basically, read-only variables). Most accessors are defined in the context of an open dataset (see part II), but some represent pre-defined constants, such as π. Again, see the *Gretl Command Reference* for a comprehensive list.

Hansl does not possess a specific Boolean type, but scalars can be used for holding true/false values. It follows that you can also use the logical operators *and* (&&), *or* (||), and *not* (!) with scalars, as in the following example:

```
a = 1
b = 0
c = !(a && b)
```

In the example above, c will equal 1 (true), since (a && b) is false, and the exclamation mark is the negation operator. Note that 0 evaluates to false, and anything else (not necessarily 1) evaluates to true.

A few constructs are taken from the C language family: one is the postfix increment operator:

```
a = 5
b = a++
print a b
```

the second line is equivalent to b = a, followed by a++, which in turn is shorthand for a = a+1, so running the code above will result in b containing 5 and a containing 6. Postfix subtraction is also supported; prefix operators, however, are not supported. Another C borrowing is inflected assignment, as in a += b, which is equivalent to a = a + b; several other similar operators are available, such as -=, *= and more. See the *Gretl Command Reference* for details.

The internal representation for a missing value is NaN ("not a number"), as defined by the IEEE 754 floating point standard. This is what you get if you try to compute quantities like the square root or the logarithm of a negative number. You can also set a value to "missing" directly using the keyword NA. The complementary functions missing and ok can be used to determine whether a scalar is NA. In the following example a value of zero is assigned to the variable named test:

```
scalar not_really = NA
scalar test = ok(not_really)
```

Note that you cannot test for equality to NA, as in

```
if x == NA ... # wrong!
```

because a missing value is taken as indeterminate and hence not equal to anything. This last example, despite being wrong, illustrates a point worth noting: the test-for-equality operator in hansl is the double equals sign, "==" (as opposed to plain "=" which indicates assignment).

3.2 Manipulation of strings

Most of the previous section applies, with obvious modifications, to strings: you may manipulate strings via operators and/or functions. Hansl's repertoire of functions for manipulating strings offers all the standard capabilities one would expect, such as toupper, tolower, strlen, etc., plus some more specialized ones. Again, see the *Gretl Command Reference* for a complete list.

In order to access part of a string, you may use the substr function,[3] as in

```
string s = "endogenous"
string pet = substr(s, 3, 5)
```

which would result to assigning the value dog to the variable pet.

The following are useful operators for strings:

- the ~ operator, to join two or more strings, as in[4]

```
string s1 = "sweet"
string s2 = "Home, " ~ s1 ~ " home."
```

[3] Actually, there is a cooler method, which uses the same syntax as matrix slicing (see chapter 4): substr(s, 3, 5) is functionally equivalent to s[3:5].

[4] On some national keyboards, you don't have the tilde (~) character. In gretl's script editor, this can be obtained via its Unicode representation: type Ctrl-Shift-U, followed by 7e.

- the closely related ~= operator, which acts as an inflected assignment operator (so `a ~= "_ij"` is equivalent to `a = a ~ "_ij"`);
- the offset operator +, which yields a substring of the preceding element, starting at the given character offset. An empty string is returned if the offset is greater than the length of the string in question.

A noteworthy point: strings may be (almost) arbitrarily long; moreover, they can contain special characters such as line breaks and tabs. It is therefore possible to use hansl for performing rather complex operations on text files by loading them into memory as a very long string and then operating on that; interested readers should take a look at the `readfile`, `getline`, `strsub` and `regsub` functions in the *Gretl Command Reference*.[5]

For *creating* complex strings, the most flexible tool is the `sprintf` function. Its usage is illustrated in Chapter 5.

[5] We are not claiming that hansl would be the tool of choice for text processing in general. Nonetheless the functions mentioned here can be very useful for tasks such as pre-processing plain text data files that do not meet the requirements for direct importation into gretl.

Chapter 4

Matrices

Matrices are one- or two-dimensional arrays of double-precision floating-point numbers. Matrices have rows and columns, and that's it (but see Section 6.2 for means of constructing higher-dimensional compound objects).

4.1 Matrix indexing

Individual matrix elements are accessed through the `[r,c]` syntax, where indexing starts at 1. For example, `X[3,4]` indicates the element of X on the third row, fourth column. For example,

```
matrix X = zeros(2,3)
X[2,1] = 4
print X
```

produces

```
X (2 x 3)

  0   0   0
  4   0   0
```

Here are some more advanced ways to access matrix elements:

1. In case the matrix has only one row (column), the column (row) specification can be omitted, as in `x[3]`.
2. Including the comma but omitting the row or column specification means "take them all", as in `x[4,]` (fourth row, all columns).
3. For square matrices, the special syntax `x[diag]` can be used to access the diagonal.
4. Consecutive rows or columns can be specified via the colon (`:`) character, as in `x[,2:4]` (columns 2 to 4). But note that, unlike some other matrix languages, the syntax `[m:n]` is illegal if $m > n$.
5. It is possible to use a vector to hold indices to a matrix. E.g. if $e = [2, 3, 6]$, then `X[,e]` contains the second, third and sixth columns of X.

Moreover, matrices can be empty (zero rows and columns).

In the example above, the matrix `X` was constructed using the function `zeros()`, whose meaning should be obvious, but matrix elements can also be specified directly, as in

```
scalar a = 2*3
matrix A = { 1, 2, 3 ; 4, 5, a }
```

The matrix is defined by rows; the elements on each row are separated by commas and rows are separated by semicolons. The whole expression must be wrapped in braces. Spaces within the braces are not significant. The above expression defines a 2×3 matrix.

Note that each element should be a numerical value, the name of a scalar variable, or an expression that evaluates to a scalar. In the example above the scalar a was first assigned a value and then used in matrix construction. (Also note, in passing, that a and A are two separate identifiers, due to case-sensitivity.)

4.2 Matrix operations

Matrix sum, difference and product are obtained via +, - and *, respectively. The prime operator (') can act as a unary operator, in which case it transposes the preceding matrix, or as a binary operator, in which case it acts as in ordinary matrix algebra, multiplying the transpose of the first matrix into the second one.[1] Errors are flagged if conformability is a problem. For example:

```
matrix a = {11, 22 ; 33, 44}  # a is square 2 x 2
matrix b = {1,2,3; 3,2,1}     # b is 2 x 3

matrix c = a'         # c is the transpose of a
matrix d = a*b        # d is a 2x3 matrix equal to a times b

matrix gina = b'd     # valid: gina is 3x3
matrix lina = d + b   # valid: lina is 2x3

/* -- these would generate errors if uncommented ----- */

# pina = a + b  # sum non-conformability
# rina = d * b  # product non-conformability
```

Other noteworthy matrix operators include ^ (matrix power), ** (Kronecker product), and the "concatenation" operators, ~ (horizontal) and | (vertical). Readers are invited to try them out by running the following code

```
matrix A = {2,1;0,1}
matrix B = {1,1;1,0}

matrix KP = A ** B
matrix PWR = A^3
matrix HC = A ~ B
matrix VC = A | B

print A B KP PWR HC VC
```

Note, in particular, that $A^3 = A \cdot A \cdot A$, which is different from what you get by computing the cubes of each element of A separately.

Hansl also supports matrix left- and right-"division", via the \ and / operators, respectively. The expression A\b solves $Ax = b$ for the unknown x. A is assumed to be an $m \times n$ matrix with full column rank. If A is square the method is LU decomposition. If $m > n$ the QR decomposition is used to find the least squares solution. In most cases, this is numerically more robust and more efficient than inverting A explicitly.

Element-by-element operations are supported by the so-called "dot" operators, which are obtained by putting a dot (".") before the corresponding operator. For example, the code

```
A = {1,2; 3,4}
B = {-1,0; 1,-1}
eval A * B
eval A .* B
```

[1] In fact, in this case an optimized algorithm is used; you should always use a'a instead of a'*a for maximal precision and performance.

produces

```
 1   -2
 1   -4

-1    0
 3   -4
```

It's easy to verify that the first operation performed is regular matrix multiplication $A \cdot B$, whereas the second one is the Hadamard (element-by-element) product $A \odot B$. In fact, dot operators are more general and powerful than shown in the example above; see the chapter on matrices in the *Gretl User's Guide* for details.

Dot and concatenation operators are less rigid than ordinary matrix operations in terms of conformability requirements: in most cases hansl will try to do "the obvious thing". For example, a common idiom in hansl is `Y = X ./ w`, where X is an $n \times k$ matrix and w is an $n \times 1$ vector. The result Y is an $n \times k$ matrix in which each row of X is divided by the corresponding element of w. In proper matrix notation, this operation should be written as

$$Y = \langle w \rangle^{-1} X,$$

where the $\langle \cdot \rangle$ indicates a diagonal matrix. Translating literally the above expression would imply creating a diagonal matrix out of w and then inverting it, which is computationally much more expensive than using the dot operation. A detailed discussion is provided in the *Gretl User's Guide*.

Hansl provides a reasonably comprehensive set of matrix functions, that is, functions that produce and/or operate on matrices. For a full list, see the *Gretl Command Reference*, but a basic "survival kit" is provided in Table 4.1. Moreover, most scalar functions, such as `abs()`, `log()` etc., will operate on a matrix element-by-element.

Function(s)	Purpose
`rows(X)`, `cols(X)`	return the number of rows and columns of X, respectively
`zeros(r,c)`, `ones(r,c)`	produce matrices with r rows and c columns, filled with zeros and ones, respectively
`mshape(X,r,c)`	rearrange the elements of X into a matrix with r rows and c columns
`I(n)`	identity matrix of size n
`seq(a,b)`	generate a row vector containing integers from a to b
`inv(A)`	invert, if possible, the matrix A
`maxc(A)`, `minc(A)`, `meanc(A)`	return a row vector with the max, min, means of each column of A, respectively
`maxr(A)`, `minr(A)`, `meanr(A)`	return a column vector with the max, min, means of each row of A, respectively
`mnormal(r,c)`, `muniform(r,c)`	generate $r \times c$ matrices filled with standard Gaussian and uniform pseudo-random numbers, respectively

Table 4.1: Essential set of hansl matrix functions

The following piece of code is meant to provide a concise example of all the features mentioned above.

```
# example: OLS using matrices

# fix the sample size
scalar T = 256

# construct vector of coefficients by direct imputation
```

```
matrix beta = {1.5, 2.5, -0.5} # note: row vector

# construct the matrix of independent variables
matrix Z = mnormal(T, cols(beta)) # built-in functions

# now construct the dependent variable: note the
# usage of the "dot" and transpose operators

matrix y = {1.2} .+ Z*beta' + mnormal(T, 1)

# now do estimation
matrix X = 1 ~ Z  # concatenation operator
matrix beta_hat1 = inv(X'X) * (X'y) # OLS by hand
matrix beta_hat2 = mols(y, X)       # via the built-in function
matrix beta_hat3 = X\y              # via matrix division

print beta_hat1 beta_hat2 beta_hat3
```

4.3 Matrix pointers

Hansl uses the "by value" convention for passing parameters to functions. That is, when a variable is passed to a function as an argument, what the function actually gets is a *copy* of the variable, which means that the value of the variable at the caller level is not modified by anything that goes on inside the function. But the use of pointers allows a function and its caller to cooperate such that an outer variable can be modified by the function.

This mechanism is used by some built-in matrix functions to provide more than one "return" value. The primary result is always provided by the return value proper but certain auxiliary values may be retrieved via "pointerized" arguments; this usage is flagged by prepending the ampersand symbol, "&", to the name of the argument variable.

The `eigensym` function, which performs the eigen-analysis of symmetric matrices, is a case in point. In the example below the first argument A represents the input data, that is, the matrix whose analysis is required. This variable will not be modified in any way by the function call. The primary result is the vector of eigenvalues of A, which is here assigned to the variable `ev`. The (optional) second argument, `&V` (which may be read as "the address of V"), is used to retrieve the right eigenvectors of A. A variable named in this way must be already declared, but it need not be of the right dimensions to receive the result; it will be resized as needed.

```
matrix A = {1,2 ; 2,5}
matrix V
matrix ev = eigensym(A, &V)
print A ev V
```

Chapter 5

Nice-looking output

5.1 Formatted output

A common occurrence when you're writing a script—particularly when you intend for the script to be used by others, and you'd like the output to be reasonably self-explanatory—is that you want to output something along the following lines:

```
The coefficient on X is Y, with standard error Z
```

where X, Y and Z are placeholders for values not known at the time of writing the script; they will be filled out as the values of variables or expressions when the script is run. Let's say that at run time the replacements in the sentence above should come from variables named vname (a string), b (a scalar value) and se (also a scalar value), respectively.

Across the spectrum of programming languages there are basically two ways of arranging for this. One way originates in the C language and goes under the name printf. In this approach we (a) replace the generic placeholders X, Y and Z with more informative *conversion specifiers*, and (b) append the variables (or expressions) that are to be stuck into the text, in order. Here's the hansl version:

```
printf "The coefficient on %s is %g, with standard error %g\n", vname, b, se
```

The value of vname replaces the conversion specifier "%s," and the values of b and se replace the two "%g" specifiers, left to right. In relation to hansl, here are the basic points you need to know: "%s" pairs with a string argument, and "%g" pairs with a numeric argument.

The C-derived printf (either in the form of a function, or in the form of a command as shown above) is present in most "serious" programming languages. It is extremely versatile, and in its advanced forms affords the programmer fine control over the output.

In some scripting languages, however, printf is reckoned "too difficult" for non-specialist users. In that case some sort of substitute is typically offered. We're skeptical: "simplified" alternatives to printf can be quite confusing, and if at some point you want fine control over the output, they either do not support it, or support it only via some convoluted mechanism. A typical alternative looks something like this (please note, display is *not* a hansl command, it's just illustrative):

```
display "The coefficient on ", vname, "is ", b, ", with standard error ", se, "\n"
```

That is, you break the string into pieces and intersperse the names of the variables to be printed. The requirement to provide conversion specifiers is replaced by a default automatic formatting of the variables based on their type. By the same token, the command line becomes peppered with multiple commas and quotation marks. If this looks preferable to you, you are welcome to join one of the gretl mailing lists and argue for its provision!

Anyway, to be a bit more precise about printf, its syntax goes like this:

printf *format*, *arguments*

The *format* is used to specify the precise way in which you want the *arguments* to be printed.

The format string

In the general case the `printf` format must be an expression that evaluates to a string, but in most cases will just be a *string literal* (an alphanumeric sequence surrounded by double quotes). However, some character sequences in the format have a special meaning. As illustrated above, those beginning with a percent sign (%) are interpreted as placeholders for the items contained in the argument list. In addition, special characters such as the newline character are represented via a combination beginning with a backslash (\).

For example,

```
printf "The square root of %d is (roughly) %6.4f.\n", 5, sqrt(5)
```

will print

```
The square root of 5 is (roughly) 2.2361.
```

Let's see how:

- The first special sequence is `%d`: this indicates that we want an integer at that place in the output; since it is the leftmost "percent" expression, it is matched to the first argument, that is 5.
- The second special sequence is `%6.4f`, which stands for a decimal value with 4 digits after the decimal separator[1] and at least 6 digits wide; this will be matched to the second argument. Note that arguments are separated by commas. Also note that the second argument is neither a scalar constant nor a scalar variable, but an expression that evaluates to a scalar.
- The format string ends with the sequence `\n`, which inserts a newline.

The conversion specifiers in the square-root example are relatively fancy, but as we noted earlier `%g` will work fine for almost all numerical values in hansl. So we could have used the simpler form:

```
printf "The square root of %g is (roughly) %g.\n", 5, sqrt(5)
```

The effect of `%g` is to print a number using up to 6 significant digits (but dropping trailing zeros); it automatically switches to scientific notation if the number is very large or very small. So the result here is

```
The square root of 5 is (roughly) 2.23607.
```

The escape sequences `\n` (newline), `\t` (tab), `\v` (vertical tab) and `\\` (literal backslash) are recognized. To print a literal percent sign, use `%%`.

Apart from those shown in the above example, recognized numeric formats are `%e`, `%E`, `%f`, `%g`, `%G` and `%x`, in each case with the various modifiers available in C. The format `%s` should be used for strings. As in C, numerical values that form part of the format (width and or precision) may be given directly as numbers, as in `%10.4f`, or they may be given as variables. In the latter case, one puts asterisks into the format string and supplies corresponding arguments in order. For example,

```
scalar width = 12
scalar precision = 6
printf "x = %*.*f\n", width, precision, x
```

If a matrix argument is given in association with a numeric format, the entire matrix is printed using the specified format for each element. A few more examples are given in table 5.1.

[1] The decimal separator is the dot in English, but may be different in other locales.

Command	effect
`printf "%12.3f", $pi`	3.142
`printf "%12.7f", $pi`	3.1415927
`printf "%6s%12.5f%12.5f %d\n", "alpha", 3.5, 9.1, 3`	alpha 3.50000 9.10000 3
`printf "%6s%12.5f%12.5f\t%d\n", "beta", 1.2345, 1123.432, %11`	beta 1.23450 1123.43200 11
`printf "%d, %10d, %04d\n", 1,2,3`	1, 2, 0003
`printf "%6.0f (%5.2f%%)\n", 32, 11.232`	32 (11.23%)

Table 5.1: Print format examples

Output to a string

A closely related effect can be achieved via the `sprintf` function: instead of being printed directly the result is stored in a named string variable, as in

```
string G = sprintf("x = %*.*f\n", width, precision, x)
```

after which the variable `G` can be the object of further processing.

Output to a file

Hansl does not have a file or "stream" type as such, but the `outfile` command can be used to divert output to a named text file. To start such redirection you must give the name of a file; by default a new file is created or an existing one overwritten but the `--append` can be used to append material to an existing file. Only one file can be opened in this way at any given time. The redirection of output continues until the command `end outfile` is given; then output reverts to the default stream.

Here's an example of usage:

```
printf "One!\n"
outfile "myfile.txt"
  printf "Two!\n"
end outfile
printf "Three!\n"
outfile "myfile.txt" --append
  printf "Four!\n"
end outfile
printf "Five!\n"
```

After execution of the above the file `myfile.txt` will contain the lines

```
Two!
Four!
```

Three special variants on the above are available. If you give the keyword `null` in place of a real filename along with the write option, the effect is to suppress all printed output until redirection is ended. If either of the keywords `stdout` or `stderr` are given in place of a regular filename the effect is to redirect output to standard output or standard error output, respectively.

This command also supports a `--quiet` option: its effect is to turn off the echoing of commands and the printing of auxiliary messages while output is redirected. It is equivalent to doing

```
set verbose off
```

before invoking `outfile`, except that when redirection is ended the prior value of the `verbose` state variable is restored.

5.2 Graphics

The primary graphing command in hansl is `gnuplot` which, as the name suggests, in fact provides an interface to the `gnuplot` program. It is used for plotting series in a dataset (see part II) or columns in a matrix. For an account of this command (and some other more specialized ones, such as `boxplot` and `qqplot`), see the *Gretl Command Reference*.

Chapter 6

Structured data types

Hansl possesses two kinds of "structured data type": associative arrays, called *bundles* and arrays in the proper sense of the word. Loosely speaking, the main difference between the two is that in a bundle you can pack together variables of different types, while arrays can hold one type of variable only.

6.1 Bundles

Bundles are *associative arrays*, that is, generic containers for any assortment of hansl types (including other bundles) in which each element is identified by a string. Python users call these *dictionaries*; in C++ and Java, they are referred to as *maps*; they are known as *hashes* in Perl. We call them *bundles*. Each item placed in the bundle is associated with a key which can used to retrieve it subsequently.

To use a bundle you first either "declare" it, as in

```
bundle foo
```

or define an empty bundle using the `defbundle` function without any arguments:

```
bundle foo = defbundle()
```

However, since defining a bundle is a common trope in hansl a handy abbreviation is supported: `_()` is short for `defbundle()`.

The formulations above are basically equivalent, in that they both create an empty bundle. The difference is that the second variant, with explicit assignment, may be reused—if a bundle named `foo` already exists the effect is to empty it—while the first may only be used once in a given gretl session; it is an error to declare a variable that already exists.

To add an object to a bundle you assign to a compound left-hand value: the name of the bundle followed by the key. The most common way to do this is to join the key to the bundle name with a dot, as in

```
foo.matrix1 = m
```

which adds an object called `m` (presumably a matrix) to bundle `foo` under the key `matrix1`. The key must satisfy the rules for a gretl variable name (31 characters maximum, starting with a letter and composed of just letters, numbers or underscore)

An alternative way to achieve the same effect is to give the key as a quoted string literal enclosed in square brackets, as in

```
foo["matrix1"] = m
```

When using the more elaborate syntax, keys do not have to be valid as variable names—for example, they can include spaces—but they are still limited to 31 characters.

To get an item out of a bundle, again use the name of the bundle followed by the key, as in

```
matrix bm = foo.matrix1
# or using the long-hand notation
matrix m = foo["matrix1"]
```

Note that the key identifying an object within a given bundle is necessarily unique. If you reuse an existing key in a new assignment, the effect is to replace the object which was previously stored under the given key. In this context it is not required that the type of the replacement object is the same as that of the original.

A quicker way is to use `defbundle` or `_()`, as in

```
bundle b = _(s="Sample string", m=I(3))
```

Note that in this style the key-strings are not quoted; this works only if they do not contain spaces. An alternative syntax can handle arbitrary keys:

```
bundle b = _("s", "Sample string", "m", I(3))
```

Here every odd-numbered argument must evaluate to a key, and every even-numbered argument to an object of a type that can be included in a bundle.

Note that when you add an object to a bundle, what in fact happens is that the bundle acquires a copy of the object. The external object retains its own identity and is unaffected if the bundled object is replaced by another. Consider the following script fragment:

```
bundle foo
matrix m = I(3)
foo.mykey = m
scalar x = 20
foo.mykey = x
```

After the above commands are completed bundle `foo` does not contain a matrix under `mykey`, but the original matrix `m` is still in good standing.

To delete an object from a bundle use the `delete` command, with the bundle/key combination, as in

```
delete foo.mykey
delete foo["quoted key"]
```

This destroys the object associated with the key and removes the key from the hash table.[1]

Besides adding, accessing, replacing and deleting individual items, the other operations that are supported for bundles are union and printing. As regards union, if bundles `b1` and `b2` are defined you can say

```
bundle b3 = b1 + b2
```

to create a new bundle that is the union of the two others. The algorithm is: create a new bundle that is a copy of `b1`, then add any items from `b2` whose keys are not already present in the new bundle. (This means that bundle union is not necessarily commutative if the bundles have one or more key strings in common.)

If `b` is a bundle and you say `print b`, you get a listing of the bundle's keys along with the types of the corresponding objects, as in

[1] Internally, gretl bundles in fact take the form of GLib hash table.

```
? print b
bundle b:
 x (scalar)
 mat (matrix)
 inside (bundle)
```

Bundle usage

To illustrate the way a bundle can hold information, we will use the Ordinary Least Squares (OLS) model as an example: the following code estimates an OLS regression and stores all the results in a bundle.

```
/* assume y and X are given T x 1 and T x k matrices */

bundle my_model = _()           # initialization
my_model.T = rows(X)            # sample size
my_model.k = cols(X)            # number of regressors
matrix e                        # will hold the residuals
b = mols(y, X, &e)              # perform OLS via native function
s2 = meanc(e.^2)                # compute variance estimator
matrix V = s2 .* invpd(X'X)     # compute covariance matrix

/* now store estimated quantities into the bundle */

my_model.betahat = b
my_model.s2 = s2
my_model.vcv = V
my_model.stderr = sqrt(diag(V))
```

The bundle so obtained is a container that can be used for all sort of purposes. For example, the next code snippet illustrates how to use a bundle with the same structure as the one created above to perform an out-of sample forecast. Imagine that $k = 4$ and the value of $\mathbf{x}$ for which we want to forecast y is

$$\mathbf{x}' = [10 \quad 1 \quad -3 \quad 0.5]$$

The formulae for the forecast would then be

$$\begin{aligned} \hat{y}_f &= \mathbf{x}'\hat{\beta} \\ s_f &= \sqrt{\hat{\sigma}^2 + \mathbf{x}' V(\hat{\beta}) \mathbf{x}} \\ CI &= \hat{y}_f \pm 1.96 s_f \end{aligned}$$

where CI is the (approximate) 95 percent confidence interval. The above formulae translate into

```
x = { 10, 1, -3, 0.5 }
scalar ypred    = x * my_model.betahat
scalar varpred  = my_model.s2 + qform(x, my_model.vcv)
scalar sepred   = sqrt(varpred)
matrix CI_95    = ypred + {-1, 1} .* (1.96*sepred)
print ypred CI_95
```

6.2 Arrays

A gretl array is a container which can hold zero or more objects of a certain type, indexed by consecutive integers starting at 1. It is one-dimensional. This type is implemented by a quite "generic" back-end. The types of object that can be put into arrays are strings, matrices, bundles, lists and arrays (that is, arrays can be nested). A given array can hold only one of these types.

Array operations

The following is, we believe, rather self-explanatory:

```
strings S1 = array(3)
matrices M = array(4)
strings S2 = defarray("fish", "chips")
S1[1] = ":)"
S1[3] = ":("
M[2] = mnormal(2,2)
print S1
eval inv(M[2])
S = S1 + S2
print S
```

The `array()` takes an integer argument for the array size; the `defarray()` function takes a variable number of arguments (one or more), each of which may be the name of a variable of the given type or an expression which evaluates to an object of that type. The corresponding output is

```
Array of strings, length 3
[1] ":)"
[2] null
[3] ":("

     0.52696      0.28883
    -0.15332     -0.68140

Array of strings, length 5
[1] ":)"
[2] null
[3] ":("
[4] "fish"
[5] "chips"
```

In order to find the number of elements in an array, you can use the `nelem()` function.

High-dimensional objects

Since one can construct an array of matrices, and arrays can be nested, it is possible to build numerical objects of higher dimensionality than matrices. For example a 3-tensor can be represented as an array of matrices, and a 4-tensor as an array of arrays of matrices.

But since such objects are compound you have be careful to get indexation right. For example, if you have an array `M` holding at least three matrices and you want to access element 2,2 of matrix 3 you'll need two sets of square brackets, as in

```
M[3][2,2]
```

Chapter 7

Numerical methods

7.1 Numerical optimization

Many, if not most, cases in which an econometrician wants to use a programming language such as hansl, rather than relying on pre-canned routines, involve some form of numerical optimization. This could take the form of maximization of a likelihood or similar methods of inferential statistics. Alternatively, optimization could be used in a more general and abstract way, for example to solve portfolio choice or analogous resource allocation problems.

Since hansl is Turing-complete, in principle any numerical optimization technique could be programmed in hansl itself. Some such techniques, however, are included in hansl's set of native functions, in the interest of both simplicity of use and efficiency. These are geared towards the most common kind of problem encountered in economics and econometrics, that is unconstrained optimization of differentiable functions.

In this chapter, we will briefly review what hansl offers to solve generic problems of the kind

$$\hat{\mathbf{x}} \equiv \underset{\mathbf{x}\in\Re^k}{\text{Argmax}}\, f(\mathbf{x};\mathbf{a}),$$

where $f(\mathbf{x};\mathbf{a})$ is a function of $\mathbf{x}$, whose shape depends on a vector of parameters $\mathbf{a}$. The objective function $f(\cdot)$ is assumed to return a scalar real value. In most cases, it will be assumed it is also continuous and differentiable, although this need not necessarily be the case. (Note that while hansl's built-in functions maximize the given objective function, minimization can be achieved simply by flipping the sign of $f(\cdot)$.)

A special case of the above occurs when $\mathbf{x}$ is a vector of parameters and $\mathbf{a}$ represents "data". In these cases, the objective function is usually a (log-)likelihood and the problem is one of estimation. For such cases hansl offers several special constructs, reviewed in section 12.2. Here we deal with more generic problems; nevertheless, the differences are only in the hansl syntax involved: the mathematical algorithms that gretl employs to solve the optimization problem are the same.

The reader is invited to read the "Numerical methods" chapter of the *Gretl User's Guide* for a comprehensive treatment. Here, we will only give a small example which should give an idea of how things are done.

```
function scalar Himmelblau(matrix x)
    /* extrema:
    f(3.0, 2.0) = 0.0,
    f(-2.805118, 3.131312) = 0.0,
    f(-3.779310, -3.283186) = 0.0
    f(3.584428, -1.848126) = 0.0
    */
    scalar ret = (x[1]^2 + x[2] - 11)^2
    return -(ret + (x[1] + x[2]^2 - 7)^2)
end function

# ------------------------------------------------------------------------

set max_verbose 1
```

```
matrix theta1 = { 0, 0 }
y1 = BFGSmax(theta1, "Himmelblau(theta1)")
matrix theta2 = { 0, -1 }
y2 = NRmax(theta2, "Himmelblau(theta2)")

print y1 y2 theta1 theta2
```

We use for illustration here a classic "nasty" function from the numerical optimization literature, namely the Himmelblau function, which has four different minima; $f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$. The example proceeds as follows.

1. First we define the function to optimize: it must return a scalar and have among its arguments the vector to optimize. In this particular case that is its only argument, but there could have been other ones if necessary. Since in this case we are solving for a minimum our definition returns the negative of the Himmelblau function proper.
2. We next set `max_verbose` to 1. This is another example of the usage of the `set` command; its meaning is "let me see how the iterations go" and it defaults to 0. By using the `set` command with appropriate parameters, you control several features of the optimization process, such as numerical tolerances, visualization of the iterations, and so forth.
3. Define $\theta_1 = [0, 0]$ as the starting point.
4. Invoke the `BFGSmax` function; this will seek the maximum via the BFGS technique. Its base syntax is `BFGSmax(arg1, arg2)`, where `arg1` is the vector contining the optimization variable and `arg2` is a string containing the invocation of the function to maximize. BFGS will try several values of θ_1 until the maximum is reached. On successful completion, the vector `theta1` will contain the final point. (Note: there's *much* more to this. For details, be sure to read the *Gretl User's Guide* and the *Gretl Command Reference*.)
5. Then we tackle the same problem but with a different starting point and a different optimization technique. We start from $\theta_2 = [0, -1]$ and use Newton–Raphson instead of BFGS, calling the `NRmax()` function instead if `BFGSmax()`. The syntax, however, is the same.
6. Finally we print the results.

Table 7.1 on page 27 contains a selected portion of the output. Note that the second run converges to a different local optimum than the first one. This is a consequence of having initialized the algorithm with a different starting point. In this example, numerical derivatives were used, but you can supply analytically computed derivatives to both methods if you have a hansl function for them; see the *Gretl User's Guide* for more detail.

The optimization methods hansl puts at your disposal are:

- BFGS, via the `BFGSmax()` function. This is in most cases the best compromise between performance and robustness. It assumes that the function to maximize is differentiable and will try to approximate its curvature by clever use of the change in the gradient between iterations. You can supply it with an analytically-computed gradient for speed and accuracy, but if you don't, the first derivatives will be computed numerically.
- Newton–Raphson, via the `NRmax()` function. Actually, the function is less specific than the name implies. This is a "curvature-based" method, relying on the iterations

  $$x_{i+1} = -\lambda_i C(x_i)^{-1} g(x_i)$$

  where $g(x)$ is the gradient and $C(x_i)$ is some measure of curvature of the function to optimize; if $C(x)$ is the Hessian matrix, you get Newton–Raphson proper. Again, you can code your own functions for $g(\cdot)$ and $C(\cdot)$, but if you don't then numerical approximations to

the gradient and the Hessian will be used, respectively. Other popular optimization methods (such as BHHH and the scoring algorithm) can be implemented by supplying to `NRmax()` the appropriate curvature matrix $C(\cdot)$. This method is very efficient when it works, but is less robust than BFGS; for example, if $C(x_i)$ happens to be non-negative definite at some iteration convergence may become problematic.

- Derivative-free methods: hansl offers simulated annealing, via the `simann` function, and the Nelder–Mead simplex algorithm (also known as the "amoeba" method) via the `NMmax` function. These methods work even when the function to be maximized has some form of disconinuity or is not everywhere differentiable; however, they may be slow and CPU-intensive.

7.2 Numerical differentiation

For numerical differentiation we have `fdjac`. For example:

```
set echo off
set messages off

function scalar beta(scalar x, scalar a, scalar b)
    return x^(a-1) * (1-x)^(b-1)
end function

function scalar ad_beta(scalar x, scalar a, scalar b)
    scalar g = beta(x, a-1, b-1)
    f1 = (a-1) * (1-x)
    f2 = (b-1) * x
    return (f1 - f2) * g
end function

function scalar nd_beta(scalar x, scalar a, scalar b)
    matrix mx = {x}
    return fdjac(mx, beta(mx, a, b))
end function

a = 3.5
b = 2.5

loop for (x=0; x<=1; x+=0.1)
    printf "x = %3.1f; beta(x) = %7.5f, ", x, beta(x, a, b)
    A = ad_beta(x, a, b)
    N = nd_beta(x, a, b)
    printf "analytical der. = %8.5f, numerical der. = %8.5f\n", A, N
endloop
```

returns

```
x = 0.0; beta(x) = 0.00000, analytical der. =  0.00000, numerical der. =  0.00000
x = 0.1; beta(x) = 0.00270, analytical der. =  0.06300, numerical der. =  0.06300
x = 0.2; beta(x) = 0.01280, analytical der. =  0.13600, numerical der. =  0.13600
x = 0.3; beta(x) = 0.02887, analytical der. =  0.17872, numerical der. =  0.17872
x = 0.4; beta(x) = 0.04703, analytical der. =  0.17636, numerical der. =  0.17636
x = 0.5; beta(x) = 0.06250, analytical der. =  0.12500, numerical der. =  0.12500
x = 0.6; beta(x) = 0.07055, analytical der. =  0.02939, numerical der. =  0.02939
x = 0.7; beta(x) = 0.06736, analytical der. = -0.09623, numerical der. = -0.09623
x = 0.8; beta(x) = 0.05120, analytical der. = -0.22400, numerical der. = -0.22400
x = 0.9; beta(x) = 0.02430, analytical der. = -0.29700, numerical der. = -0.29700
x = 1.0; beta(x) = 0.00000, analytical der. = -0.00000, numerical der. =       NA
```

Details on the algorithm used can be found in the *Gretl Command Reference*. Suffice it to say here that you have a `fdjac_quality` setting that goes from 0 to 2. The default value is to 0, which gives you forward-difference approximation: this is the fastest algorithm, but sometimes may not be precise enough. The value of 1 gives you bilateral difference, while 2 uses Richardson extrapolation. Higher values give greater accuracy but the method becomes considerably more CPU-intensive.

```
? matrix theta1 = { 0, 0 }
Replaced matrix theta1
? y1 = BFGSmax(theta1, "Himmelblau(11, theta1)")
Iteration 1: Criterion = -170.000000000
Parameters:       0.0000      0.0000
Gradients:        14.000      22.000 (norm 0.00e+00)

Iteration 2: Criterion = -128.264504038 (steplength = 0.04)
Parameters:      0.56000     0.88000
Gradients:        33.298      39.556 (norm 5.17e+00)

...

--- FINAL VALUES:
Criterion = -1.83015730011e-28 (steplength = 0.0016)
Parameters:       3.0000      2.0000
Gradients:    1.7231e-13 -3.7481e-13 (norm 7.96e-07)

Function evaluations: 39
Evaluations of gradient: 16
Replaced scalar y1 = -1.83016e-28
? matrix theta2 = { 0, -1 }
Replaced matrix theta2
? y2 = NRmax(theta2, "Himmelblau(11, theta2)")
Iteration 1: Criterion = -179.999876556 (steplength = 1)
Parameters:   1.0287e-05     -1.0000
Gradients:        12.000  2.8422e-06 (norm 7.95e-03)

Iteration 2: Criterion = -175.440691085 (steplength = 1)
Parameters:      0.25534     -1.0000
Gradients:        12.000  4.5475e-05 (norm 1.24e+00)

...

--- FINAL VALUES:
Criterion = -3.77420797114e-22 (steplength = 1)
Parameters:       3.5844     -1.8481
Gradients:   -2.6649e-10  2.9536e-11 (norm 2.25e-05)

Gradient within tolerance (1e-07)
Replaced scalar y2 = -1.05814e-07
? print y1 y2 theta1 theta2

             y1 = -1.8301573e-28

             y2 = -1.0581385e-07

theta1 (1 x 2)

  3   2

theta2 (1 x 2)

      3.5844      -1.8481
```

Table 7.1: Output from maximization

Chapter 8

Control flow

The primary means for controlling the flow of execution in a hansl script are the `if` statement (conditional execution), the `loop` statement (repeated execution), the `catch` modifier (which enables the trapping of errors that would otherwise halt execution), and the `quit` command (which forces termination).

8.1 The `if` statement

Conditional execution in hansl uses the `if` keyword. Its fullest usage is as follows

```
if <condition>
   ...
elif <condition>
   ...
else
   ...
endif
```

Points to note:

- The `<condition>` can be any expression that evaluates to a scalar: 0 is interpreted as "false", non-zero is interpreted as "true"; `NA` generates an error.
- Following `if`, "then" is implicit; there is no `then` keyword as found in, e.g., Pascal or Basic.
- The `elif` and `else` clauses are optional: the minimal form is just `if ... endif`.
- Conditional blocks of this sort can be nested up to a maximum depth of 1024.

Example:

```
scalar x = 15

# --- simple if ----------------------------------
if x >= 100
   printf "%g is more than two digits long\n", x
endif

# --- if with else -------------------------------
if x >= 0
   printf "%g is non-negative\n", x
else
   printf "%g is negative\n", x
endif

# --- multiple branches --------------------------
if missing(x)
   printf "%g is missing\n", x
elif x < 0
```

```
   printf "%g is negative\n", x
elif floor(x) == x
   printf "%g is an integer\n", x
else
   printf "%g is a positive number with a fractional part\n", x
endif
```

Note, from the example above, that the `elif` keyword can be repeated, making hansl's `if` statement a multi-way branch statement. There is no separate `switch` or `case` statement in hansl. With one or more `elif`s, hansl will execute the first one for which the logical condition is satisfied and then jump to `endif`.

☞ Stata users, beware: hansl's `if` statement is fundamentally different from Stata's `if` option: the latter selects a subsample of observations for some action, while the former is used to decide if a group of statements should be executed or not; hansl's `if` is what Stata calls "branching `if`".

The ternary query operator

Besides use of `if`, the ternary query operator, `?:`, can be used to perform conditional assignment on a more "micro" level. This has the form

```
result = <condition> ? <value-if-true> : <value-if-false>
```

If `<condition>` evaluates as "true" (non-zero) then the first following value is assigned to `result`, otherwise the value after the colon is so assigned.[1] This is obviously more compact than `if` ... `else` ... `endif`. The following example replicates the `abs` function by hand:

```
scalar ax = x>=0 ? x : -x
```

Of course, in the above case it would have been much simpler to just write `ax = abs(x)`. Consider, however, the following case, which exploits the fact that the ternary operator can be nested:

```
scalar days = (m==2) ? 28 : maxr(m.={4,6,9,11}) ? 30 : 31
```

This example deserves a few comments. We want to compute the number of days in a month, coded in the variable `m`. The value we assign to the scalar `days` comes from the following pathway.

1. First we check if the month is February (`m==2`); if so, we set `days` to 28 and we're done.[2]
2. Otherwise, we compute a matrix of zeros and ones via the operation `m.={4,6,9,11}` (note the use of the "dot" operator to perform an element-by element comparison—see section 4.2); if `m` equals any of the elements in the vector, the corresponding element of the result will be 1, and 0 otherwise;
3. The `maxr` function gives the maximum of this vector, so we're checking whether `m` is any one of the four values corresponding to 30-day months.
4. Since the above evaluates to a scalar, we put the correct value into `days`.

The ternary operator is more flexible than the ordinary `if` statement. With `if`, the `<condition>` to be evaluated must always come down to a scalar, but the query operator just requires that the condition is of "suitable" type in light of the types of the operands. So, for example, suppose you have a square matrix `A` and you want to switch the sign of the negative elements of `A` on and above its diagonal. You could use a loop (see below) and write a piece of code such as

[1] Some readers may find it helpful to note that the conditional assignment operator works in exactly the same way as the `=IF()` function in spreadsheets.

[2] OK, we're ignoring leap years here.

```
matrix A = mnormal(4,4)
matrix B = A

loop r = 1 .. rows(A)
  loop c = r .. cols(A)
     if A[r,c] < 0
       B[r,c] = -A[r,c]
     endif
  endloop
endloop
```

By using the ternary operator, you can achieve the same effect via a considerably shorter (and faster) construct:

```
matrix A = mnormal(4,4)
matrix B = upper(A.<0) ? -A : A
```

☞ At this point some readers may be thinking "Well, this may be as cool as you want, but it's way too complicated for me; I'll just use the traditional `if`". Of course, there's nothing wrong with that, but in some cases the ternary assignment operator can lead to substantially faster code, and it becomes surprisingly natural when one gets used to it.

8.2 Loops

The basic hansl command for looping is (doh!) `loop`, and takes the form

```
loop <control-expression> <options>
    ...
endloop
```

In other words, the pair of statements `loop` and `endloop` enclose the statements to repeat. Of course, loops can be nested. Several variants of the `<control-expression>` for a loop are supported, as follows:

1. unconditional loop
2. while loop
3. index loop
4. foreach loop
5. for loop.

These variants are briefly described below.

Unconditional loop

This is the simplest variant. It takes the form

```
loop <times>
   ...
endloop
```

where `<times>` is any expression that evaluates to a scalar, namely the required number of iterations. This is only evaluated at the beginning of the loop, so the number of iterations cannot be changed from within the loop itself. Example:

```
# triangular numbers
scalar n = 6
scalar count = 1
scalar x = 0
loop n
    scalar x += count
    count++
    print x
endloop
```

yields

```
x =  1.0000000
x =  3.0000000
x =  6.0000000
x =  10.000000
x =  15.000000
x =  21.000000
```

Note the usage of the increment (`count++`) and of the inflected assignment (`x += count`) operators.

Index loop

The unconditional loop is used quite rarely, as in most cases it is useful to have a counter variable (`count` in the previous example). This is easily accomplished via the *index loop*, whose syntax is

```
loop <counter>=<min>..<max>
   ...
endloop
```

The limits `<min>` and `<max>` must evaluate to scalars; they are automatically turned into integers if they have a fractional part. The `<counter>` variable is started at `<min>` and incremented by 1 on each iteration until it equals `<max>`.

The counter is "read-only" inside the loop. You can access either its numerical value through the scalar `i` or use the accessor `$i`, which will perform *string substitution*: inside the loop, the hansl interpreter will substitute for the expression `$i` the string representation of the current value of the index variable. An example should made this clearer: the following input

```
scalar a_1 = 57
scalar a_2 = 85
scalar a_3 = 13

loop i=1..3
    print i a_$i
endloop
```

has for output

```
    i = 1.0000000
  a_1 = 57.000000
    i = 2.0000000
  a_2 = 85.000000
    i = 3.0000000
  a_3 = 13.000000
```

In the example above, at the first iteration the value of `i` is 1, so the interpreter expands the expression `a_$i` to `a_1`, finds that a scalar by that name exists, and prints it. The same happens through the rest of the iterations. If one of the automatically constructed identifiers had not been defined, execution would have stopped with an error.

While loop

Here you have

```
loop while <condition>
   ...
endloop
```

where <condition> should evaluate to a scalar, which is re-evaluated at each iteration. Looping stops as soon as <condition> becomes false (0). If <condition> becomes NA, an error is flagged and execution stops. By default, while loops cannot exceed 100,000 iterations. This is intended as a safeguard against potentially infinite loops. This setting can be overridden if necessary by setting the loop_maxiter state variable to a different value.

Foreach loop

In this case the syntax is

```
loop foreach <counter> <catalogue>
   ...
endloop
```

where <catalogue> can be either a collection of space-separated strings, or a variable of type list (see section 11.2). The counter variable automatically takes on the numerical values 1, 2, 3, and so on as execution proceeds, but its string value (accessed by prepending a dollar sign) shadows the names of the series in the list or the space-separated strings; this sort of loop is designed for string substitution.

Here is an example in which the <catalogue> is a collection of names of functions that return a scalar value when given a scalar argument.

```
scalar x = 1
loop foreach f sqrt exp ln
    scalar y = $f(x)
    print y
endloop
```

This will produce

```
y =  1.0000000
y =  2.7182818
y =  0.0000000
```

For loop

The final form of loop control emulates the for statement in the C programming language. The syntax is loop for, followed by three component expressions, separated by semicolons and surrounded by parentheses, that is

```
loop for (<init>; <cont>; <modifier>)
   ...
endloop
```

The three components are as follows:

1. Initialization (<init>): this must be an assignment statement, evaluated at the start of the loop.

2. Continuation condition (<cont>): this is evaluated at the top of each iteration (including the first). If the expression evaluates as true (non-zero), iteration continues, otherwise it stops.
3. Modifier (<modifier>): an expression which modifies the value of some variable. This is evaluated prior to checking the continuation condition, on each iteration after the first.

Here's an example, in which we find the square root of a number by successive approximations:

```
# find the square root of x iteratively via Newton's method
scalar x = 256
d = 1
loop for (y=(x+1)/2; abs(d) > 1.0e-7; y -= d/(2*y))
    d = y*y - x
    printf "y = %15.10f, d = %g\n", y, d
endloop

printf "sqrt(%g) = %g\n", x, y
```

Running the example gives

```
y =  128.5000000000, d = 16256.3
y =   65.2461089494, d = 4001.05
y =   34.5848572866, d = 940.112
y =   20.9934703720, d = 184.726
y =   16.5938690915, d = 19.3565
y =   16.0106268314, d = 0.340172
y =   16.0000035267, d = 0.000112855
y =   16.0000000000, d = 1.23919e-11

Number of iterations: 8

sqrt(256) = 16
```

Be aware of the limited precision of floating-point arithmetic. For example, the code snippet below will iterate forever on most platforms because `x` will never equal *exactly* 0.01, even though it might seem that it should.

```
loop for (x=1; x!=0.01; x=x*0.1)
    printf "x = .18g\n", x
endloop
```

However, if you replace the condition `x!=0.01` with `x>=0.01`, the code will run as (probably) intended.

Loop options

Three options can be given to the `loop` statement. One is `--verbose`. This has simply the effect of printing extra output to trace progress of the loop; it has no other effect and the semantics of the loop contents remain unchanged.

The `--decr` option can be applied to an index loop to indicate that the counter should be decremented by 1, not incremented, on each iteration. Note that the default behavior with an index loop is that the code is skipped altogether if the starting index value exceeds the ending value.

The `--progressive` option is mostly used as a quick and efficient way to set up simulation studies. When this option is given, a few commands (notably `print` and `store`) are given a special, *ad hoc* meaning. Please refer to the *Gretl User's Guide* for more information.

Breaking and continuing

The `break` command makes it possible to break out of a loop if necessary. Note that if you nest loops, `break` in the innermost loop will interrupt that loop only and not the outer ones. Here is an example in which we use the `while` variant of the `loop` statement to perform calculation of the square root in a manner similar to the example above, using `break` to jump out of the loop when the job is done.

```
scalar x = 256
scalar y = 1
loop while 1
    d = y*y - x
    if abs(d) < 1.0e-7
        break
    else
        y -= d/(2*y)
        printf "y = %15.10f, d = %g\n", y, d
    endif
endloop

printf "sqrt(%g) = %g\n", x, y
```

The `continue` command can be used to short-circuit an iteration: execution jumps from the line on which `continue` occurs to the top of the loop. Iteration will then proceed if the continuation condition is met. This can promote efficiency if a condition is met at a certain point in an iteration such that the subsequent code becomes irrelevant.

8.3 The `catch` modifier

Hansl offers a simple form of exception handling via the `catch` keyword. This is not a command in its own right but can be used as a prefix to most regular commands: the effect is to prevent termination of a script if an error occurs in executing the command. If an error does occur, this is registered in an internal error code which can be accessed as `$error` (a zero value indicating success). The value of `$error` should always be checked immediately after using `catch`, and appropriate action taken if the command failed. Here is a simple example:

```
matrix a = floor(2*muniform(2,2))
catch ai = inv(a)
scalar err = $error
if err
    printf "The matrix\n%6.0f\nis singular!\n", a
else
    print ai
endif
```

Note that the catch keyword cannot be used before `if`, `elif` or `endif`. In addition, it should not be used on calls to user-defined functions; it is intended for use only with gretl commands and calls to "built-in" functions or operators. Suppose you're writing a function package which includes some subsidiary functionality which may fail under certain conditions, and you want to prevent such failure from aborting execution. In that case you should use `catch` *within* the particular function in question, and if an error condition is detected, signal this to the caller by returning a suitable "invalid" value—say, `NA` (for a function that returns a scalar) or an empty matrix. For example:

```
function scalar may_fail (matrix *m)
  catch scalar x = ... # call to built-in procedure
  if $error
```

```
    x = NA
  endif
  return x
end function

function scalar caller (...)
  matrix m = ... # whatever
  scalar x = may_fail(&m)
  if na(x)
    print "Couldn't calculate x"
  else
    printf "Calculated x = %g\n", x
  endif
end function
```

What you should *not* do here is apply catch to `may_fail()`

```
function scalar caller (...)
  matrix m = ... # whatever
  catch scalar x = may_fail(&m) # No, don't do this!
  ...
end function
```

as this is likely to leave gretl in a confused state.

8.4 The `quit` statement

When the `quit` statement is encountered in a hansl script, execution stops. If the command-line program gretlcli is running in batch mode, control returns to the operating system; if gretl is running in interactive mode, gretl will wait for interactive input.

The `quit` command is rarely used in scripts since execution automatically stops when script input is exhausted, but it could be used in conjunction with `catch`. A script author could arrange matters so that on encountering a certain error condition an appropriate message is printed and the script is halted. Another use for `quit` is in program development: if you want to inspect the output of an initial portion of a complex script, the most convenient solution may to insert a temporary "quit" at a suitable point.

Chapter 9

User-written functions

Hansl natively provides a reasonably wide array of pre-defined functions for manipulating variables of all kinds; the previous chapters contain several examples. However, it is also possible to extend hansl's native capabilities by defining additional functions.

Here's what a user-defined function looks like:

```
function type funcname(parameters)
  function body
end function
```

The opening line of a function definition contains these elements, in strict order:

1. The keyword `function`.
2. *`type`*, which states the type of value returned by the function, if any. This must be one of `void` (if the function does not return anything), `scalar`, `series`, `matrix`, `list`, `string`, `bundle`, or one of the array types, that is `bundles`, `lists`, `matrices` and `strings`;
3. *`funcname`*, the unique identifier for the function. Function names have a maximum length of 31 characters; they must start with a letter and can contain only letters, numerals and the underscore character. They cannot coincide with the names of native commands or functions.
4. The function's *`parameters`*, in the form of a comma-separated list enclosed in parentheses. Note: parameters are the only way hansl function can receive anything from "the outside". In hansl there are no global variables.

Function parameters can be of any of the types shown below.

Type	Description
`bool`	scalar variable acting as a Boolean switch
`int`	scalar variable acting as an integer
`scalar`	scalar variable
`series`	data series (see section 11.1)
`list`	named list of series (see section 11.2)
`matrix`	matrix or vector
`string`	string variable or string literal
`bundle`	all-purpose container
`matrices`	array of matrices
`bundles`	array of bundles
`strings`	array of strings
`arrays`	array of arrays

Each element in the listing of parameter must include two terms: a type specifier, and the name by which the parameter shall be known within the function.

The *function body* contains (almost) arbitrary hansl code, which should compute the *return value*, that is the value the function is supposed to yield. Any variable declared inside the function is *local*, so it will cease to exist when the function ends.

The `return` command is used to stop execution of the code inside the function and deliver its result to the calling code. This typically happens at the end of the function body, but doesn't have to. The function definition must end with the expression `end function`, on a line of its own.

☞ Beware: unlike some other languages (e.g. Matlab or GAUSS), you cannot directly return multiple outputs from a function. However, you can return a multiple-item object, such as an array for homogenous returns or a bundle for heterogenous items, and stuff it with as many objects as you want.

In order to get a feel for how functions work in practice, here's a simple example:

```
function scalar quasi_log (scalar x)
   /* popular approximation to the natural logarithm
      via Padé polynomials
   */
   if x < 0
      scalar ret = NA
   else
      scalar ret = 2*(x-1)/(x+1)
   endif
   return ret
end function

loop for (x=0.5; x<2; x+=0.1)
   printf "x = %4.2f; ln(x) = %g, approx = %g\n", x, ln(x), quasi_log(x)
endloop
```

The code above computes the rational function

$$f(x) = 2 \cdot \frac{x-1}{x+1},$$

which provides a decent approximation to the natural logarithm in the neighborhood of 1. Some comments on the code:

1. Since the function is meant to return a scalar, we put the keyword `scalar` after the intial `function`.
2. In this case the parameter list has only one element: it is named `x` and is specified to be a scalar.
3. On the next line the function definition begins; the body includes a comment and an `if` block.
4. The function ends by returning the computed value, `ret`.
5. The lines below the function definition give a simple example of usage. Note that in the `printf` command, the two functions `ln()` and `quasi_log()` behave in exactly the same way from a purely syntactic viewpoint, although the former is native and the latter is user-defined.

In ambitious uses of hansl you may end up writing several functions, some of which may be quite long. In order to avoid cluttering your script with function definitions, hansl provides the `include` command: you can put your function definitions in a separate file (or set of files) and read them in as needed. For example, suppose you saved the definition of `quasi_log()` in a separate file called `quasilog_def.inp`: the code above could then be written more compactly as

```
include quasilog_def.inp
```

```
loop for (x=0.5; x<2; x+=0.1)
   printf "x = %4.2f; ln(x) = %g, approx = %g\n", x, ln(x), quasi_log(x)
endloop
```

Moreover, `include` commands can be nested.

9.1 Parameter passing and return values

In hansl, parameters are by default passed *by value*, so what is used inside the function is a copy of the original argument. You may modify it, but you'll be just modifying the copy. The following example should make this point clear:

```
function void f(scalar x)
    x = x*2
    print x
end function

scalar x = 3
f(x)
print x
```

Running the above code yields

```
x =  6.0000000
x =  3.0000000
```

The first `print` statement is executed inside the function, and the displayed value is 6 because the input `x` is doubled; however, what really gets doubled is simply a *copy* of the original `x`: this is demonstrated by the second `print` statement. If you want a function to modify its arguments, you must use pointers.

Copying the content of the incoming parameter to a local version may have a sizeable impact on compute speed and memory usage when the object is large (say, a 1000×1000 matrix). To avoid this cost you can prepend the `const` modifier to the parameter type, thereby promising that the object in question will not be modified inside the function. In that case gretl will grant read-only access to the object at caller level instead of copying it (and will flag an error at any attempt to modify the object). Consult the the *Gretl User's Guide* for further details.

Pointers

Each of the type-specifiers, with the exception of `list`, may be modified by prepending an asterisk to the associated parameter name, as in

```
function scalar myfunc (matrix *y)
```

This indicates that the required argument is not a plain matrix but rather a *pointer-to-matrix*, or in other words the memory address at which the variable is stored.

This can seem a bit mysterious to people unfamiliar with the C programming language, so allow us to explain how pointers work by analogy. Suppose you set up a barber shop. Ideally, your customers would walk into your shop, sit on a chair and have their hair trimmed or their beard shaved. However, local regulations forbid you to modify anything coming in through your shop door. Of course, you wouldn't do much business if people must leave your shop with their hair untouched. Nevertheless, you have a simple way to get around this limitation: your customers can come to your shop, tell you their home address and walk out. Then, nobody stops you from going to their place and exercising your fine profession. You're OK with the law, because no modification of anything took place inside your shop.

While our imaginary restriction on the barber seems arbitrary, the analogous restriction in a programming context is not: it prevents functions from having unpredictable side effects. (You might be upset if it turned out that your person was modified after visiting the grocery store!)

In hansl (unlike C) you don't have to take any special care within the function to distinguish the variable from its address,[1] you just use the variable's name. In order to supply the address of a variable when you invoke the function, you use the ampersand (&) operator.

An example should make things clearer. The following code

```
function void swap(scalar *a, scalar *b)
    scalar tmp = a
    a = b
    b = tmp
end function

scalar x = 0
scalar y = 1000000
swap(&x, &y)
print x y
```

gives the output

```
x =  1000000.0
y =  0.0000000
```

So x and y have in fact been swapped. How?

First you have the function definition, in which the arguments are pointers to scalars. Inside the function body, the distinction is moot, as a is taken to mean "the scalar that you'll find at the address given by the first argument" (and likewise for b). The rest of the function simply swaps a and b by means of a local temporary variable.

Outside the function, we first initialize the two scalars x to 0 and y to a big number. When the function is called, it is given as arguments &a and &b, which hansl identifies as "the address of" the two scalars a and b, respectively.

Besides making it possible to modify function arguments in such a way that they stay modified at caller level, use of pointer arguments avoids the computational cost of copying arguments. However, it is not idiomatic in hansl to use the pointer-argument mechanism for cost-saving alone since the same effect can be achieved via the `const` modifier described above.

Advanced parameter passing and optional arguments

The parameters to a hansl function can be also specified in more sophisticated ways than outlined above. There are three additional features worth mentioning:

1. A descriptive string can be attached to each parameter for GUI usage.
2. For some parameter types, there is a special syntax construct for ensuring that its value is bounded; for example, you can stipulate a scalar argument to be positive, or constrained within a pre-specified range.
3. Some of the arguments can be made optional.

A thorough discussion is too long to fit in this document, and the interested reader should refer to the "User-defined functions" chapter of the the *Gretl User's Guide*. Here we'll just show you a simple, and hopefully self-explanatory, example which combines features 2 and 3. Suppose you have a function for producing smileys, defined as

[1] In C, this would be called *dereferencing* the pointer. The distinction is not required in hansl because there is no equivalent to operating on the supplied address itself, as in C.

```
function void smileys(int times[0::1], bool frown[0])
    if frown
        string s = ":-("
    else
        string s = ":-)"
    endif

    loop times
        printf "%s ", s
    endloop

    printf "\n"
end function
```

Then, running

```
smileys()
smileys(2, 1)
smileys(4)
```

produces

```
:-)
:-( :-(
:-) :-) :-) :-)
```

Embedding arguments in bundles

Some complex functions may require a large number of arguments. There is no limit to the number of arguments a function can have, but an overly complicated function signature is not pleasant to use. Some programming languages (R, for one) obviate this problem by using *named arguments*, so that you may call a function by supplying only the few arguments you actually need, leaving the other ones to their default values.

In hansl we don't have named arguments, but a commonly employed technique achieves the same result in a similar way: the idea is to package arguments into a bundle (see Section 6.1) and use the bundle syntax to handle its contents.

For example, say you want to write a function to extract a substring from a string and optionally capitalize it. You could start from something like

```
function string Sub(string s, scalar ini, scalar fin, bool capital)
    string ret = s[ini:fin]
    return capital ? toupper(ret) : ret
end function
```

so the call `Sub("nowhere", 4, 7, 1)` would produce the string `HERE`. A more sophisticated version of the function may have default values, so that you could call the function in a simplified form. Using the syntax shown in the previous subsection, one could set the defaults as

```
function string Sub(string s, scalar ini[1], scalar fin[3], bool capital[FALSE])
    string ret = s[ini:fin]
    return capital ? toupper(ret) : ret
end function
```

and the call `Sub("nowhere")` would return the string `now`. However, if we wanted the string to be capitalized, we would have to set the fourth parameter to 1: to get `NOW` the function would have to

be called as `Sub("nowhere", , ,1)`. With more than 5 or 6 parameters in the function signature, this becomes quite awkward.

This issue can be resolved by putting arguments 2 to 4 into a bundle, as in

```
function string Sub(string s, bundle opts)
    string ret = s[opts.ini:opts.fin]
    return opts.capital ? toupper(ret) : ret
end function
```

where the returned string is computed using the bundle contents, and you may call the function as

```
  bundle myopts = _(ini=1, fin=3, capital=TRUE)
  string out = Sub("nowhere", myopts)
```

Note that these two lines could be combined as

```
  string out = Sub("nowhere", _(ini=1, fin=3, capital=TRUE))
```

but in some cases it may be convenient to have the options bundle as a persistent object, so that successive calls to the function may take place with incremental changes.

This sort of mechanism lends itself naturally to handling default values in an elegant way. Consider the code below:

```
function string Sub(string s, bundle opts_in[null])
    bundle opts = _(ini=1, fin=3, capital=0)
    if exists(opts_in)
        opts = opts_in + opts
    endif
    string ret = s[opts.ini:opts.fin]
    return opts.capital ? toupper(ret) : ret
end function
```

Let's analyse the body of the function line by line:

1. the function signature contains only two arguments: the string to process and a bundle, which has a default value of `null` and so can be omitted.
2. A bundle `opts` is defined with default values for the scalars `ini` and `fin` and for the boolean flag `capital`.
3. If a bundle was passed as the second argument, then the line

   ```
   opts = opts_in + opts
   ```

   replaces the keys in `opts` with those present in `opts_in` (with the + operator, the left-hand bundle takes precedence). At this point, the bundle `opts` will contain a mixture of default and user-set keys.
4. From here on, everything proceeds as above.

This means that the call `Sub("nowhere")` would yield `now`, but if we want capitalized output we can call the function as

```
  string out = Sub("nowhere", _(capital=TRUE))
```

The "incremental variations" idea for the options bundle (mentioned above) can be now exploited as in the following code:

```
bundle myopts = _(capital=TRUE)
string out1 = Sub("nowhere", myopts)
myopts.fin = 2
string out2 = Sub("nowhere", myopts)
```

Execution this code gives strings `out1` containing `NOW` and `out2` containing `NO`.

At this point we have something virtually equivalent to named arguments. Note that the keys in `_()`, unlike individual function arguments, can be given in any order.

9.2 Recursion

Hansl functions can be recursive; what follows is the obligatory factorial example:

```
function scalar factorial(scalar n)
    if (n<0) || (n>floor(n))
        # filter out everything that isn't a
        # non-negative integer
        return NA
    elif n==0
        return 1
    else
        return n * factorial(n-1)
    endif
end function

loop i = 0 .. 6
    printf "%d! = %d\n", i, factorial(i)
endloop
```

This is fun, but in practice you'll be much better off using the pre-cooked gamma function (or better still, its logarithm).

Part II

With a dataset

Chapter 10

What is a dataset?

A dataset is a memory area designed to hold the data you want to work on, if any. It may be thought of a big global variable, containing a (possibly huge) matrix of data and a hefty collection of metadata.

R users may think that a dataset is similar to what you get when you `attach` a data frame in R. Not really: in hansl, you cannot have more than one dataset open at the same time. That's why we talk about *the* dataset.

When a dataset is present in memory (that is, "open"), a number of objects become available for your hansl script in a transparent and convenient way. Of course, the data themselves: the columns of the dataset matrix are called *series*, which will be described in section 11.1; sometimes, you will want to organize one or more series into a *list* (Section 11.2). Additionally, you have the possibility of using, as read-only global variables, some scalars or matrices, such as the number of observations, the number of variables, the nature of your dataset (cross-sectional, time series or panel), and so on. These are called *accessors*, and will be discussed in section 10.5.

You can open a dataset by reading data from a disk file, via the `open` command, or by creating one from scratch.

10.1 Creating a dataset from scratch

The primary commands in this context are `nulldata` and `setobs`. For example:

```
set verbose off

set seed 443322    # initialize the random number generator
nulldata 240       # stipulate how long your series will be
setobs 12 1995:1   # define as monthly data, starting Jan 1995
```

For more details see the *Gretl User's Guide*, and the *Gretl Command Reference* for the `nulldata` and `setobs` commands. The only important thing to say at this point, however, is that you can resize your dataset and/or change some of its characteristics, such as its periodicity, at nearly any point inside your script if necessary.

Once your dataset is in place, you can start populating it with series, either by reading them from files or by generating them via appropriate commands and functions.

10.2 Reading a dataset from a file

The primary commands here are `open`, `append` and `join`.

The `open` command is what you'll want to use in most cases. It handles transparently a wide variety of formats (native, CSV, spreadsheet, data files produced by other packages such as Stata, Eviews, SPSS and SAS) and takes care of setting up the dataset for you automatically.

```
open mydata.gdt      # native format
open yourdata.dta    # Stata format
open theirdata.xlsx  # Excel format
```

The open command can also be used to read data directly from the Internet, by using a URL instead of a filename, as in

```
open http://someserver.com/somedata.csv
```

The *Gretl User's Guide* describes the requirements on plain text data files of the "CSV" type for direct importation by gretl. It also describes gretl's native data formats (XML-based and binary).

The append and join commands can be used to add further series from file to a previously opened dataset. The join command is extremely flexible and has a chapter to itself in the *Gretl User's Guide*.

10.3 Saving datasets

The store command is used to write the current dataset (or a subset) out to file. Besides writing in gretl's native formats, store can also be used to export data as CSV or in the format of R. Series can be written out as matrices using the mwrite function. If you have special requirements that are not met by store or mwrite it is possible to use outfile plus printf (see chapter 5) to gain full control over the way data are saved.

10.4 The smpl command

Once you have opened a dataset somehow, the smpl command allows you to discard observations selectively, so that your series will contain only the observations you want (automatically changing the dimension of the dataset in the process). See chapter 4 in the *Gretl User's Guide* for further information.[1]

There are basically three variants to the smpl command:

1. Selecting a contiguous subset of observations: this will be mostly useful with time-series datasets. For example:

   ```
   smpl 4 122            # select observations for 4 to 122
   smpl 1984:1 2008:4    # the so-called "Great Moderation" period
   smpl 2008-01-01 ;     # observations from January 1st, 2008 onwards
   ```

2. Selecting observations on the basis of some criterion: this is typically what you want with cross-sectional datasets. Example:

   ```
   smpl male == 1 --restrict               # males only
   smpl male == 1 && age < 30 --restrict   # just the young guys
   smpl employed --dummy                   # via a dummy variable
   ```

   Note that, in this context, restrictions go "on top of" previous ones, or in other words are cumulated. In order to start from scratch, you either reset the full sample via smpl full or use the --replace option along with --restrict.

3. Restricting the active dataset to some observations so that a certain effect is achieved automatically: for example, drawing a random subsample, or ensuring that all rows that have missing observations are automatically excluded. This is achieved via the --no-missing, --contiguous, and --random options.

In the context of panel datasets, some extra qualifications have to be made; see the *Gretl User's Guide*.

[1] Users with a Stata background may find the hansl way of doing things a little disconcerting at first. In hansl, you first restrict your sample through the smpl command, which applies until further notice, then you do what you have to. There is no equivalent to Stata's if clause to commands.

10.5 Dataset accessors

Several characteristics of the current dataset can be determined by reference to built-in accessor ("dollar") variables. The main ones, which all return scalar values, are shown in Table 10.1.

Accessor	Value returned
`$datatype`	Coding for the type of dataset: 0 = no data; 1 = cross-sectional (undated); 2 = time-series; 3 = panel
`$nobs`	The number of observations in the current sample range
`$nvars`	The number of series (including the constant)
`$pd`	The data frequency (1 for cross-sectional, 4 for quarterly, and so on)
`$t1`	1-based index of the first observation in the current sample
`$t2`	1-based index of the last observation in the current sample

Table 10.1: The principal dataset accessors

In addition there are a few more specialized accessors: `$obsdate`, `$obsmajor`, `$obsminor`, `$obsmicro` and `$unit`. These are specific to time-series and/or panel data, and they all return series. See the *Gretl Command Reference* for details.

Chapter 11

Series and lists

Scalars, matrices and strings can be used in a hansl script at any point; series and lists, on the other hand, are inherently tied to a dataset and therefore can be used only when a dataset is currently open.

11.1 The `series` type

Series are just what any applied economist would call "variables", that is, repeated observations of a given quantity; a dataset is an ordered array of series, complemented by additional information, such as the nature of the data (time-series, cross-section or panel), descriptive labels for the series and/or the observations, source information and so on. Series are the basic data type on which gretl's built-in estimation commands depend.

The series belonging to a dataset are named via standard hansl identifiers (strings of maximum length 31 characters as described above). In the context of commands that take series as arguments, series may be referenced either by name or by *ID number*, that is, the index of the series within the dataset. Position 0 in a dataset is always taken by the automatic "variable" known as `const`, which is just a column of 1s. The IDs of the actual data series can be displayed via the `varlist` command. (But note that in *function calls*, as opposed to commands, series must be referred to by name.) A detailed description of how a dataset works can be found in chapter 4 of the *Gretl User's Guide*.

Some basic rules regarding series follow:

- If `lngdp` belongs to a time series or panel dataset, then the syntax `lngdp(-1)` yields its first lag, and `lngdp(+1)` its first lead.
- To access individual elements of a series, you use square brackets enclosing
  - the progressive (1-based) number of the observation you want, as in `lngdp[15]`, or
  - the corresponding date code in the case of time-series data, as in `lngdp[2008:4]` (for the 4th quarter of 2008), or
  - the corresponding observation marker string, if the dataset contains any, as in `GDP["USA"]`.

The rules for assigning values to series are just the same as for other objects, so the following examples should be self-explanatory:

```
series k = 3         # implicit conversion from scalar; a constant series
series x = normal()  # pseudo-rv via a built-in function
series s = a/b       # element-by-element operation on existing series

series movavg = 0.5*(x + x(-1)) # using lags
series y[2012:4] = x[2011:2]    # using individual data points
series x2000 = 100*x/x[2000:1]  # constructing an index
```

☞ In hansl, you don't have separate commands for *creating* series and *modifying* them. Other popular packages make this distinction, but we still struggle to understand why this is supposed to be useful.

Converting series to or from matrices

The reason why hansl provides a specific series type, distinct from the matrix type, is historical. However, is also a very convenient feature. Operations that are typically performed on series in applied work can be awkward to implement using "raw" matrices—for example, the computation of leads and lags, or regular and seasonal differences; the treatment of missing values; the addition of descriptive labels, and so on.

Anyway, it is straightforward to convert data in either direction between the series and matrix types.

- To turn series into matrices, you use the curly braces syntax, as in

  ```
  matrix MACRO = {outputgap, unemp, infl}
  ```

  where you can also use lists; the number of rows of the resulting matrix will depend on your currently selected sample.

- To turn matrices into series, you can just use matrix columns, as in

  ```
  series y = my_matrix[,4]
  ```

  but only if the number of rows in `my_matrix` matches the length of the dataset, or the currently selected sample range.

Also note that the `lincomb` and `filter` functions are quite useful for creating and manipulating series in complex ways without having to convert the data to matrix form (which could be computationally costly with large datasets).

The ternary operator with series

Consider this assignment:

```
worker_income = employed ? income : 0
```

Here we assume that `employed` is a dummy series coding for employee status. Its value will be tested for each observation in the current sample range and the value assigned to `worker_income` at that observation will be determined accordingly. It is therefore equivalent to the following much more verbose formulation (where `$t1` and `$t2` are accessors for the start and end of the sample range):

```
series worker_income
loop i=$t1..$t2
    if employed[i]
        worker_income[i] = income[i]
    else
        worker_income[i] = 0
    endif
endloop
```

11.2 The `list` type

In hansl parlance, a *list* is an array of integers, representing the ID numbers of a set (in a loose sense of the word) of series. For this reason, the most common operations you perform on lists are set operations such as addition or deletion of members, union, intersection and so on. Unlike sets, however, hansl lists are ordered, so individual list members can be accessed via the `[]` syntax, as in `X[3]` to access the third series in list `X`.

There are several ways to assign values to a list. The most basic sort of expression that works in this context is a space-separated list of series, given either by name or by ID number. For example,

```
list xlist = 1 2 3 4
list reglist = income price
```

An empty list is obtained by using the function `deflist` without any arguments, as in

```
list W = deflist()
```

or simply by bare declaration. Some more special forms (for example, using wildcards) are described in the *Gretl User's Guide*.

The main idea is to use lists to group, under one identifier, one or more series that logically belong together somehow (for example, as explanatory variables in a model). So, for example,

```
list xlist = x1 x2 x3 x4
ols y 0 xlist
```

is an idiomatic way of specifying the OLS regression that could also be written as

```
ols y 0 x1 x2 x3 x4
```

Note that we used here the convention, mentioned in section 11.1, by which a series can be identified by its ID number when used as an argument to a command, typing `0` instead of `const`.

Lists can be concatenated, as in as in `list L3 = L1 L2` (where `L1` and `L2` are names of existing lists). This will not necessarily do what you want, however, since the resulting list may contain duplicates. It's more common to use the following set operations:

Operator	Meaning
`||`	Union
`&&`	Intersection
`-`	Set difference

So for example, if `L1` and `L2` are existing lists, after running the following code snippet

```
list UL = L1 || L2
list IL = L1 && L2
list DL = L1 - L2
```

the list `UL` will contain all the members of `L1`, plus any members of `L2` that are not already in `L1`; `IL` will contain all the elements that are present in both `L1` and `L2` and `DL` will contain all the elements of `L1` that are not present in `L2`.

To *append* or *prepend* variables to an existing list, we can make use of the fact that a named list stands in for a "longhand" list. For example, assuming that a list `xlist` is already defined (possibly as `null`), we can do

```
list xlist = xlist 5 6 7
xlist = 9 10 xlist 11 12
```

Another option for appending terms to, or dropping terms from, an existing list is to use `+=` or `-=`, respectively, as in

```
xlist += cpi
zlist -= cpi
```

A nice example of the above is provided by a common idiom: you may see in hansl scripts something like

```
list C -= const
list C = const C
```

which ensures that the series `const` is included (exactly once) in the list `C`, and comes first.

Converting lists to or from matrices

The idea of converting from a list, as defined above, to a matrix may be taken in either of two ways. You may want to turn a list into a matrix (vector) by filling the latter with the ID numbers contained in the former, or rather to create a matrix whose columns contain the series to which the ID numbers refer. Both interpretations are legitimate (and potentially useful in different contexts) so hansl lets you go either way.

If you assign a list to a matrix, as in

```
list L = moo foo boo zoo
matrix A = L
```

the matrix `A` will contain the ID numbers of the four series as a row vector. This operation goes both ways, so the statement

```
list C = seq(7,10)
```

is perfectly valid (provided, of course, that you have at least 10 series in the currently open dataset).

If instead you want to create a data matrix from the series which belong to a given list, you have to enclose the list name in curly brackets, as in

```
matrix X = {L}
```

The `foreach` loop variant with lists

Lists can be used as the "catalogue" in the `foreach` variant of the `loop` construct (see section 8.2). This is especially handy when you have to perform some operation on multiple series. For example, the following syntax can be used to calculate and print the mean of each of several series:

```
list X = age income experience
loop foreach i X
    printf "mean($i) = %g\n", mean($i)
endloop
```

Chapter 12

Estimation methods

You can, of course, estimate econometric models via hansl without having a dataset (in the sense in which we're using that term here) in place—just as you might in Matlab, for instance. You'll need *data*, but these can be loaded in matrix form (see the `mread` function in the *Gretl Command Reference*), or generated artificially via functions such as `mnormal` or `muniform`. You can roll your own estimator using hansl's linear algebra primitives, and you also have access to more specialized functions such as `mols` (see section 4.2) and `mrls` (restricted least squares) if you need them.

However, unless you need to use an estimation method which is not currently supported by gretl, or have a strong desire to reinvent the wheel, you will probably want to make use of the built-in estimation commands available in hansl. These commands are series-oriented and therefore require a dataset. They fall into two main categories: "canned" procedures, and generic tools that can be used to estimate a wide variety of models based on common principles.

12.1 Canned estimation procedures

"Canned" maybe doesn't sound very appetizing but it's the term that's commonly used. Basically it means two things, neither of them in fact unappetizing.

- The user is presented with a fairly simple interface. A few inputs must be specified, and perhaps a few options selected, then the heavy lifting is done within the gretl library. Full results are printed (parameter estimates plus numerous auxiliary statistics).
- The algorithm is written in C, by experienced coders. It is therefore faster (possibly *much* faster) than an implementation in an interpreted language such as hansl.

Most such procedures share the syntax

commandname parameters options

where *parameters* usually takes the form of a listing of series: the dependent variable followed by the regressors.

The line-up of procedures can be crudely categorized as follows:

Linear, single equation:	`ols`, `tsls`, `ar1`, `mpols`
Linear, multi-equation:	`system`, `var`, `vecm`
Nonlinear, single equation:	`logit`, `probit`, `poisson`, `negbin`, `tobit`, `intreg`, `logistic`, `duration`
Panel:	`panel`, `dpanel`
Miscellaneous:	`arima`, `garch`, `heckit`, `quantreg`, `lad`, `biprobit`

Don't let names deceive you: for example, the `probit` command can estimate ordered models, random-effects panel probit models, ... The hansl "house style" is to keep to a relatively small number of command words and to distinguish variants within a class of estimators such as Probit by means of options, or the character of the data supplied.

Simultaneous systems (SUR, FIML and so on) constitute the main exception to the syntax summary above; these require a `system` block—see the chapter on Multivariate models in the *Gretl User's Guide*.

12.2 Generic estimation tools

Hansl offers three main toolkits for defining estimators beyond the canned selection. Here's a quick overview:

command	estimator	*User's Guide*
`nls`	nonlinear least squares	chapter 25
`mle`	maximum likelihood estimation	chapter 26
`gmm`	generalized method of moments	chapter 27

Each of these commands takes the form of a block of statements (e.g. `nls ... end nls`). The user must supply a function to compute the fitted dependent variable (`nls`), the log-likelihood (`mle`), or the GMM residuals (`gmm`). With `nls` and `mle`, analytical derivatives of the function in question with respect to the parameters may (optionally) be supplied.

The most widely used of these tools is probably `mle`. Hansl offers several canned ML estimators, but if you come across a model that you want to estimate via maximum likelihood and it is not supported natively, all you have to do is write down the log-likelihood in hansl's notation and run it through the `mle` apparatus.

12.3 Post-estimation accessors

All of the methods mentioned above are *commands*, not functions; they therefore do not *return* any values. However, after estimating a model—either using a canned procedure or one of the toolkits—you can grab most of the quantities you might wish to have available for further analysis via accessors.

Some such accessors are generic, and are available after using just about any estimator. Examples include `$coeff` and `$stderr` (to get the vectors of coefficients and standard errors, respectively), `$uhat` and `$yhat` (residuals and fitted values), and `$vcv` (the covariance matrix of the coefficients). Some, on the other hand, are specific to certain estimators. Examples here include `$jbeta` (the cointegration matrix, following estimation of a VECM), `$h` (the estimated conditional variance series following GARCH estimation), and `$allprobs` (the matrix of per-outcome probabilities following ordered logit and probit, and multinomial logit estimation).

A full listing and description of accessors can be found in the *Gretl Command Reference*.

12.4 Formatting the results of estimation

The commands mentioned in this chapter produce by default quite verbose (and, hopefully, nicely formatted) output. However, in some cases you may want to use built-in commands as auxiliary steps in implementing an estimator that is not itself built in. In that context the standard printed output may be inappropriate and you may want to take charge of presenting the results yourself.

This can be accomplished quite easily. First, you can suppress the usual output by using the `--quiet` option with built-in estimation commands.[1] Second, you can use the `modprint` command to generate the desired output. As usual, see the *Gretl Command Reference* for details.

[1] For some commands, `--quiet` reduces but does not eliminate gretl's usual output. In these cases you can give the `--silent` option. Consult the *Gretl Command Reference* to determine which commands accept this option.

12.5 Named models

We said above that estimation commands in hansl don't return anything. This should be qualified in one respect: it is possible to use a special syntax to push a model onto a stack of named models. Rather than the usual assignment symbol, the form "<-" is used for this purpose. This is mostly intended for use in the gretl GUI but it can also be used in hansl scripting.

Once a model is saved in this way, the accessors mentioned above can be used in a special way, joined by a dot to the name of the target model. A little example follows. (Note that `$ess` accesses the error sum of squares, or sum of squared residuals, for models estimated via least squares.)

```
diff y x
ADL <- ols y const y(-1) x(0 to -1)
ECM <- ols d_y const d_x y(-1) x(-1)
# the following two values should be equal
ssr_a = ADL.$ess
ssr_e = ECM.$ess
```

Part III

Reference

Chapter 13

Rules regarding white space

Programming languages differ in their rules regarding the use of white space in a program. Here we set out the rules in hansl. The rules differ somewhat between *commands* on the one hand and *function calls* plus *assignment* on the other.

13.1 White space in commands

Hansl commands are structured as follows: first comes a *command word* (e.g. `ols`, `summary`); then come zero or more *arguments* (often the names of series); then come zero or more *options* (some of which may take parameters). The relevant rules are:

1. The individual elements just mentioned must always be separated by at least one space, and where one space is required you are free to insert as many as you like.
2. Whenever a parameter is supplied with an option flag, the parameter must be attached to the flag with an equals sign, with *no* intervening space:

```
ols y 0 x --cluster=clustvar  # correct
ols y 0 x --cluster =clustvar # broken!
```

13.2 Spaces in function calls and assignment

For the most part, white space in function calls and assignment is not significant; it can be inserted or not at will. For example, in the following sets of statements each member is equally acceptable syntactically (though some are ugly!):

```
# set 1
y = sqrt(x)
y=sqrt(x)
# set 2
c = cov(y1, y2)
c=cov(y1,y2)
c  = cov( y1 , y2 )
```

But when an assignment starts with a type keyword such as `series` or `matrix`, this must be separated from what follows by at least one space, as in

```
series y = normal() # or: series y=normal()
```

Chapter 14

Operators

14.1 Precedence

Table 14.1 lists the operators available in gretl in order of decreasing precedence. That is, the operators on the first row have the highest precedence, those on the second row have the second highest, and so on, while operators on any given row have equal precedence. Where successive operators have the same precedence the order of evaluation is in general left to right. The exceptions are exponentiation and matrix transpose-multiply. The expression `a^b^c` is equivalent to `a^(b^c)`, not `(a^b)^c`, and similarly `A'B'C'` is equivalent to `A'(B'(C'))`.

Table 14.1: Operator precedence

```
()   []   .    {}
!    ++   --   ^    '
*    /    %    \    **
+    -    ~    |
>    <    >=   <=   ..
==   !=
&&
||
?:
```

In addition to the basic forms shown in the Table, several operators also have a "dot form" (as in "`.+`" which is read as "dot plus"). These are element-wise versions of the basic operators, for use with matrices exclusively; they have the same precedence as their basic counterparts. The available dot operators are as follows.

```
.^   .*   ./   .+   .-   .>   .<   .>=   .<=   .=
```

Each basic operator is shown once again in the following list along with a brief account of its meaning. Apart from the first three sets of grouping symbols, all operators are binary except where noted.

`()` Function call
`[]` Subscripting
`.` Bundle membership (see below)
`{}` Matrix definition
`!` Unary logical NOT
`++` Increment (unary)
`--` Decrement (unary)
`^` Exponentiation

'	Matrix transpose (unary) or transpose-multiply (binary)
*	Multiplication
/	Division, matrix "right division"
%	Modulus
\	Matrix "left division"
**	Kronecker product
+	Addition
-	Subtraction
~	Matrix horizontal concatenation
|	Matrix vertical concatenation
>	Boolean greater than
<	Boolean less than
>=	Greater than or equal
<=	Less than or equal
..	Range from–to (in constructing lists)
==	Boolean equality test
!=	Boolean inequality test
&&	Logical AND
||	Logical OR
?:	Conditional expression

The interpretation of "`.`" as the bundle membership operator is confined to the case where it is immediately preceded by the identifier for a bundle, and immediately followed by a valid identifier (key).

Details on the use of the matrix-related operators (including the dot operators) can be found in the chapter on matrices in the *Gretl User's Guide*.

14.2 Assignment

The operators mentioned above are all intended for use on the right-hand side of an expression which assigns a value to a variable (or which just computes and displays a value—see the `eval` command). In addition we have the assignment operator itself, "=". In effect this has the lowest precedence of all: the entire right-hand side is evaluated before assignment takes place.

Besides plain "=" several "inflected" versions of assignment are available. These may be used only when the left-hand side variable is already defined. The inflected assignment yields a value that is a function of the prior value on the left and the computed value on the right. Such operators are formed by prepending a regular operator symbol to the equals sign. For example,

```
y += x
```

The new value assigned to `y` by the statement above is the prior value of `y` plus `x`. The other available inflected operators, which work in an exactly analogous fashion, are as follows.

```
-=   *=   /=   %=   ^=   ~=   |=
```

In addition, a special form of inflected assignment is provided for matrices. Say matrix `M` is 2×2. If you execute `M = 5` this has the effect of replacing `M` with a 1×1 matrix with single element 5. But if you do `M .= 5` this assigns the value 5 to all elements of `M` without changing its dimensions.

14.3 Increment and decrement

The unary operators ++ and -- follow their operand,[1] which must be a variable of scalar type. Their simplest use is in stand-alone expressions, such as

```
j++  # shorthand for j = j + 1
k--  # shorthand for k = k - 1
```

However, they can also be embedded in more complex expressions, in which case they first yield the original value of the variable in question, then have the side-effect of incrementing or decrementing the variable's value. For example:

```
scalar i = 3
k = i++
matrix M = zeros(10, 1)
M[i++] = 1
```

After the second line, `k` has the value 3 and `i` has value 4. The last line assigns the value 1 to element 4 of matrix `M` and sets `i` = 5.

Warning: as in the C programming language, the unary increment or decrement operator should be not be applied to a variable in conjunction with regular reference to the same variable in a single statement. This is because the order of evaluation is not guaranteed, giving rise to ambiguity. Consider the following:

```
M[i++] = i # don't do this!
```

This is supposed to assign the value of `i` to `M[i]`, but is it the original or the incremented value? This is not actually defined.

[1] The C programming language also supports prefix versions of ++ and --, which increment or decrement their operand before yielding its value. Only the postfix form is supported by gretl.

Chapter 15

Greek-letter identifiers

As mentioned in chapter 3, the sole exception to the requirement that hansl identifiers must be plain ASCII is that they may take the form of a single Greek letter. Here are the details.

- This exception applies only to names of variables *other than series*; the names of series must always be ASCII.
- The supported Greek characters are the 24 (unaccented) letters in the basic Greek alphabet, minus omicron, which is indistinguishable from the Latin 'o'.
- These letters may be used in lower or upper case (constituting distinct identifiers, as usual in hansl), except for the several upper-case letters which are indistinguishable in the Latin and Greek alphabets ('A', 'B', 'E', 'K', 'M', 'N', ...).
- The Greek letters must be encoded in UTF-8.

In gretl's graphical interface (script editor and GUI "console"), the acceptable Greek letters can be entered by typing a Latin letter while the `Alt` key is depressed. The mappings from Latin to Greek are shown below: lower case first, then upper case.

Latin	Greek		Latin	Greek	
a	α	alpha	n	ν	nu
b	β	beta	p	π	pi
c	χ	chi	q	θ	theta
d	δ	delta	r	ρ	rho
e	ϵ	epsilon	s	σ	sigma
f	ϕ	phi	t	τ	tau
g	γ	gamma	u	υ	upsilon
h	η	eta	v	ν	nu
i	ι	iota	w	ω	omega
j	ψ	psi	x	ξ	xi
k	κ	kappa	y	υ	upsilon
l	λ	lambda	z	ζ	zeta
m	μ	mu			

Latin	Greek	
D	Δ	Delta
F	Φ	Phi
G	Γ	Gamma
J	Ψ	Psi
L	Λ	Lambda
P	Π	Pi
Q	Θ	Theta
S	Σ	Sigma
U	ϒ	Upsilon
W	Ω	Omega
X	Ξ	Xi
Y	ϒ	Upsilon

A word of advice: it's probably not a good idea to employ Greek-letter identifiers in hansl scripts that you intend to share via the internet, since one cannot assume that text encodings are preserved unchanged. This warning applies in particular if you, or any of the intended recipients of your scripts, work on MS Windows, since Windows does not natively support UTF-8, the mandatory encoding of such identifiers for use with gretl.