Chapter 1: An overview of Scilab

Since GROCER is basically a toolbox for Scilab, it is necessary to have a basic knowledge of Scilab: the aim of this chapter is thus to provide this minimum knowledge, and perhaps a little more. The presentation will be succinct and will not cover the subjects that are part of Scilab but not a priori useful for an econometrician. The interested reader will find some documentation at http://scilabsoft.inria.fr/product/index_product.php?page=overview.html. There are also introductions to Scilab in English, French, German, Portuguese and Spanish, as long as an English introduction to Scilab « from a Matlab point of view », all available at http://scilabsoft.inria.fr/contrib/index_contrib.php?page=download.php&category=MANUALS. The French one, by Bruno Pinçon, is very good and has proved very useful to us when we started using Scilab.

Part 1 of this chapter presents Scilab main characteristics. Part 2 indicates how to get Scilab. Part 3 explains how to start with Scilab. Part 4 presents Scilab main objects. Part 5 is devoted to the specific objects that are functions, part 6 to input/output and part 7 presents how to customize Scilab. Graphics are rejected to chapter 15, since GROCER contains itself its own graphic procedures, based obviously on Scilab ones.

1. Scilab main characteristics

Scilab is a matrix oriented software, similar to Gauss, and much more still, to Matlab: a user accustomed to one of these two applications should have no difficulties with Scilab. Scilab even contains a tool for translating Matlab functions into Scilab ones: although this tool is not perfect (in particular, it does not translate correctly Matlab structures), it does a good job at translating the greatest part of Matlab functions and I have used it intensively to build GROCER.

Scilab has therefore the advantages and drawbacks of these applications: on the positive side, because econometric tools rely heavily on matrices formulas, the implementation in scilab of econometric methods is very easy and not very time-consuming; on the negative side, functions written in scilab will be slower than functions written in fortran or C.

Scilab, which makes heavy use of fortran and C routines, allows also the user to incorporate (« link ») her own fortran and C routines: the use of this capability in GROCER could be envisaged for further developments.

Scilab contains a rich menu of types: besides usual types such as, « constant » (that is real or complex numbers), string, boolean, list, function, Scilab contains more exotic types, such as polynomial rational. All data types exist also in sparse forms. Lastly, Scilab contains a very useful type, not existing in Gauss and Matlab: the typed-list type (tlist). This last type allows one to create new data structures, that are similar to new types of variable. When a data structure has been defined as a tlist, Scilab indeed considers it represents a new type and allows you to define very easily for this new data structure the usual operations, such as addition, multiplication… This capability is used by GROCER to create a structure that does not exist in Scilab, but is very useful to an econometrician: the structure timeseries.

Scilab runs as well on unix, linux or windows (from windows 95 to XP at least !) and, apart from small differences as regards the appearance and the input/output, behaves identically in all 3 operating systems. This should be the same as regard GROCER. However, for the time being, GROCER has been

tested only on Windows. So, although there should not be any problem with Unix or Linux OS, do not hesitate to signal them to us if it were not the case.

2. How to get Scilab

Scilab can be downloaded from the following address: http://scilabsoft.inria.fr/download/index_download.php?page=release.html. The user should download the version adapted to her operating system and needs. The current version is the 4.0 one, but for some OS (Windows NT), you should download the 3.0 version: grocer works also on this version, although grocer graphs are not totally up-to-date in the version available for Scilab 3.0.

3. Getting started with Scilab

Once you have installed Scilab, running scilab (by run scilab on unix and by clicking on the Scilab 4.0 on the desk) creates a window, that contains the following message:

```
_____________________________________________
                  scilab-4.0

          Copyright (c) 1989-2006
      Consortium Scilab (INRIA, ENPC)
_____________________________________________


Startup execution:
  loading initial environment

-->
```

You are now ready to use Scilab, and, once you have downloaded and installed it, to use GROCER. This implies typing a command at the prompt (see below for the first examples and the whole manual for many others):

```
-->
```

At the top of the window, a menu with 7 items appears: file; edit; preferences; control; editor; application; ? (help). Menu file contains a few commands used in Scilab and the command exit to terminate Scilab session. Menu edit contains mainly a Copy / paste capability. Menu Preferences can be used to change default settings (language, font,…) and to clear Scilab window. Menu Control contains notably the resume command to restart a function that has stopped at a pause and an abort command that aborts the running function. Menu Editor launches Scilab editor if you want to create a script or a Scilab function. Applications is a menu that collects various Scilab applications, in particular m2sci, that allows the translation from Matlab to scilab and browsevar that allows to know the type of the variables loaded in Scilab. Menu ? contaisn in particular Scilab help, and notably the description of Scilab –and, once the toolbox has been installed, grocer– functions. Note (see part 7 in this chapter) that you can add your own items in the menu toolbar.

4. Scilab objects

This part reviews main scilab objects. Functions are dealt in part 5 and sparse objects, mlist (very similar to tlist), polynomial and rational objects are ignored (see the « Introduction to Scilab » or Scilab manual for a presentation of these types). We therefore review successively constant type, string type, list type and briefly (since chapter 3 deals in detail with an application which is part of GROCER) tlist type boolean type.

4.1 Creating Objects

When working on the working space, you can simply perform calculations without keeping the result, or save the result for later calculations: the saved result will then be available until you leave Scilab or until you clear it, if you decide so.

For instance, creating a constant equal to 2+2 and storing it into variable v involves writing on screen:

```
-->v=2+2
```

And you obtain on screen:

```
v  =

4.
```

This example, as all examples contained in this manual, has been processed by Scilab and is typed with *Courier New* font whereas the main text is written with Times New Roman font. Here the result of the command is displayed. If you do not want to display the results of a command, then end it with a ';'. Several commands can be typed on the same line if they are separated by a ';'.

For instance:

```
-->v=[1 2 3 5]; a=2
 a  =

2
```

If you do no want to keep the result of the calculation, then you simply type:

```
-->2+2
```

and you obtain on screen:

```
ans  =

4.
```

Scilab then creates a variable called ans, which can be used immediately for other calculations:

```
-->ans+2
```

and you obtain on screen:

```
ans  =

6.
```

The result has again been saved in ans, which is no longer equal to 4.

4.2 Constant types.

Creating constant typed objects

The constant type recovers all types of numbers. The most simple ones are real constants, such as 2 (already encountered in part 4.1), 3.456 or 1.234E6. Scilab contains also specific constants, prefixed by %: %pi (π), %e (the trigonometric constant e), %inf (infinity), %eps (Scilab precision, that is the small value used by Scilab for rounding results down). The constant %nan is used for a non definite number.

Scilab also handles complex numbers. The complex $\sqrt{-1}$ is defined as %i; a complex number will then be written as a + %i*b. This capability has, alas, from GROCER point of view some drawbacks: the inverse of a badly-defined variance matrix can for instance be a complex matrix!

The constant type covers also vectors or matrices made up with any of all the constants defined above. The creation of a row vector involves the following command: v= [v1 v2 … vn] or v=[v1, v2,…,vn] where each vi is one of the constants defined above.

For instance, here is the result of the following instruction on Scilab screen:

```
-->v=[1 2 3 5]
 v  =

!   1.    2.    3.    5. !
```

Only the result of the last instruction has been displayed on screen. Vector v has nevertheless been created in the working space and can be recovered at any moment.

The creation of a column vector involves the command: v=[v1; v2;…; vn]. Coordinates of a column are therefore separated y a semi-column, coordinates of a row vector by a blank or a column.

Lines of a matrix are separated by the character ";". For instance, here is a command creating a (2x3) matrix:

```
-->a=[4 5 1 ; -2 0.5 1.4E2]
```

which results in:

```
 a  =

!   4.    5.     1.   !
! - 2.     .5    140. !
```

There are some predefined matrices: zeros(m,n) creates a (mxn) null matrix; ones(m,n) creates a (mxn) matrix of ones; eye (m,n) creates a matrix with ones on the main diagonal, 0 otherwise (eye(n,n) is then the identity matrix); [] creates the empty matrix; toeplitz(c,r) creates a Toeplitz matrix… (see Scilab manual for more details).

You can also create linearly spaced vectors with the instruction:
v=n1:step:n2
or with the instruction:
v=linspace(n1,n2,nnumb)
with nnumb=1+(n2-n1)/step

In the first case, the increment "step" can be omitted: it is then assumed to be equal to 1.

For instance, here are two equivalent commands to produce a sequence starting from Π/2 and going to Π with a Π/4 step (at least on my computer: Bruno Pinçon signals that it can be different according to the O/S):

```
-->%pi/2:%pi/4:%pi
-->linspace(%pi/2,%pi,3)
```

which both give the following result:

```
ans  =

!   1.5707963    2.3561945    3.1415927 !
```

Operations on constant matrices

Scilab authorizes all classical operations on matrices: addition, subtraction, multiplication (see table 1, taken from the "Introduction to Scilab", for a recapitulation), Kronecker product… It authorizes also element-wise operations, provided that the formats of the objects are conformable: in that case, the operation symbol is preceded by the dot.

For instance, assuming that A is the matrix [1:3;4:6] and B the matrix ones(2,3), then compare the results of the 2 following instructions:

```
-->A*B'
-->A.*B
```

whose results are respectively:

```
ans  =

!   6.     6.  !
!   15.    15. !
```

and:

```
ans  =

!   1.    2.    3. !
!   4.    5.    6. !
```

In the first case, what you have obtained is the product of matrix A by the transposition of matrix B; in the second one, you have the result of the product of A and B elementwise: the first coordinate of the result is the product of the first element of A by the first element of B, the (1,2) coordinate of the result is the product of the (1,2) coordinate of A by (1,2) coordinate of B,…

Table 1: Basic matrix operations in Scilab

SYMBOL	OPERATION
`[]`	matrix definition, concatenation
`;`	row separator
`( )`	extraction: `m=a(k)`
`( )`	insertion: `a(k)=m`
`'`	transpose
`+`	addition
`-`	subtraction
`*`	multiplication
`\`	left division
`/`	right division
`^`	exponent
`.*`	elementwise multiplication
`.\`	elementwise left division
`./`	elementwise right division
`.^`	elementwise exponent
`.*.`	kronecker product
`./.`	kronecker right division
`.\.`	kronecker left division

There can be some ambiguity with the dot, when it applies to numbers, since real numbers can end with a dot. So, the best way is to write the second command with blanks between the operator .* and the operands A and B, like this[1]:

```
-->A .* B
```

Note also there is one case where Scilab allows adding or subtracting matrices of different sizes: this is when one matrix is in fact a constant (a (1x1) matrix) or empty. In the first case, Scilab adds the constant to each element of the other matrix. In the second case, it leaves the matrix unchanged.

For instance, with the already defined matrix A:

```
-->A+1.5
 ans  =

!   2.5    3.5    4.5 !
!   5.5    6.5    7.5 !

-->A+[]
 ans  =

!   1.    2.    3. !
!   4.    5.    6. !
```

Scilab contains also numerous functions, such as sqrt, log, exp, trigonometric functions,… (see Scilab manual for a complete description). All these functions apply to numbers as well to matrices: in that case, the function is simply applied to each element of the matrix.

[1] I must confess that I have not adopted this principle in GROCER functions (mainly because I have applied the element-wise operations to named vectors), but I should have….

4.3 String objects.

Creating string objects

A string in Scilab is a text between quotes. For instance, to create the string 'this is a string', type the following command:

```
-->s='this is a string'
```

with the following result:

```
 s  =

 this is a string
```

You can create vectors and matrices of strings exactly in the same way you create a vector or matrix of real values. For instance:

```
-->m=['a' 'b' ; 'c' 'd']
```

results in:

```
 m  =

!a  b  !
!      !
!c  d  !
```

Note that Scilab presentation on screen introduces a blank line between each line of the matrix… Note also that m will have the type 'string' exactly as the string s.

If you want to create a string which contains quotes, then you have to double the quote. For instance:

```
-->s='let''s go!'
```

results in:

```
 s  =

 let's go!
```

Operations on strings

Scilab contains a lot of useful tools for the manipulation of strings: length (to know the length of a string); part (to compute a substring); strsubst (to replace in a string all occurrences of a specified text)… (see scilab manual for details).

The concatenation of 2 strings is made very simply with the + operator:

```
st='no'+'thing'
```

results in:

```
 st  =

 nothing
```

Two Scilab tools play a great role in GROCER: the functions evstr and execstr. These functions evaluate and execute respectively the string which is given to them as input. For instance, with then same A and B matrix as before:

```
-->M=evstr('B ./ A')
```

or:

```
--> execstr(' M=B ./ A')
```

results in:

```
 M  =

!   1.       .5     .3333333 !
!    .25     .2     .1666667 !
```

GROCER adds one function, which can be useful: function joinstr (see chapter 4), which concatenates altogether vectors of strings elementwise and simple strings into a unique simple string.

4.4 Lists

Lists in Scilab are ordered and finite[2] collections of objects of any type, eventually mixed in the same list. The creation of a list involves the command list: mylist=list(object1,…,objectn). For instance: chap1_list=list(A,B,v,M,s) will collect most objects created so far in this chapter, which have constant and string types (but an element of a list can also be a boolean, a function or even a list,…).

```
-->chap1_list=list(A,B,v,M,s)

chap1_list(1)

!   1.    2.    3. !
!   4.    5.    6. !

       chap1_list(2)

!   1.    1.    1. !
!   1.    1.    1. !

       chap1_list(3)

!   1.    0.    3. !

       chap1_list(4)

!   1.       .5     .3333333 !
!    .25     .2     .1666667 !

       chap1_list(5)

 this is a string
```

4.5 Typed lists

[2] Contrary to programming languages such as Haskell or Caml, you cannot therefore not create infinite lists.

Typed lists are specific and very useful lists. The corresponding keyword is tlist instead of list and the first element must be a string column vector, say v. The first coordinate of v is the type of the list and the following ones are the fields of the list. For instance:

```
-->T=tlist(['portfolio';'bonds';'shares'],25000,12234)
```

results in:

```
 T  =


       T(1)

!portfolio  !
!           !
!bonds      !
!           !
!shares     !

       T(2)

    25000.

       T(3)

    12234.
```

Typed lists have two advantages. The first is that you can extract an element of the list by its name. For instance:

```
-->T('bonds')
```

results in:

```
 ans  =

    25000.
```

This feature allows in particular one to mimic Matlab structures: most of Lesage's Matlab programs that have been translated into Scilab, contains results structures, which have been translated into Scilab results tlists. But the greatest advantage is that you can define on typed lists the usual operators and functions. You could for instance define the addition of two portfolios T1 and T2: this involves writing a function %portfolio_a_porfolio[3]… And once this function exists, the addition of portfolios T1 and T2 will be made simply by typing T1+T2. This is the feature I have used to define timeseries in Scilab (see chapter 3).

4.6 Updating vectors, matrices and lists

Extracting an element from a matrix is done by writing: mat[i;j]. For instance:

```
-->A=[1:3;4:6];A(2,3)
results in:
ans  =

6.
```

Replacing an element in a matrix is done by the instruction: mat[i;j]=new_value. For instance:

[3] Function that has not been implemented in GROCER…

```
-->A(2,3)=18
```

results in:

```
 A  =

!   1.    2.    3.  !
!   4.    5.    18. !
```

For a vector, the index of the row (for a column vector) or of the column (for a row vector) can be omitted. For instance:

```
-->v=[1 2 3 5];v(3)=%e
```

results in:

```
 v  =

!   1.    2.    2.7182818    5. !
```

Adding an element at the end of a vector can be done in two ways: first by specifying mat(newi,newj)=new_value; second, by writing vec=[vec new_value] (for a row vector) or vec=[vec ; new_value] (for a column vector). For instance:

```
-->v(5)=0
```

which results in:

```
 v  =

!   1.    2.    2.7182818    5.    0. !
```

or:

```
-->v=[v 0]
```

which gives the same result:

```
v  =

!   1.    2.    2.7182818    5.    0. !
```

The second way should be preferred, because it is more efficient in Scilab. If you know in advance the size of the final vector, then it is still better to define in advance the vector in the following way:

```
-->v=ones(1,5);v(1:4)=[1 2 %e 5];v(5)=0
```

Which gives also the same result (recall that 1:4=[1 2 3 4]).

The same principles apply to add a vector to a matrix.

Deleting elements in a matrix is done by the command: vec(i:j)=[]. For instance:

```
-->v(2:4)=[]
```

results in the (2,1) vector[4]:

```
 v  =

!   1.    0. !
```

In that context, a very useful tool in Scilab is the character $. This character refers to the end of a vector or matrix. For instance:

[4] Technically, this can be seen as a sort of dynamic array.

```
-->v($)
```

results in

```
 ans  =

    0.
```

and:

```
-->v($+1)=3
```

results in:

```
 v  =

!   1.    0.    3. !
```

Extracting an element of a *list* is done as with matrices:

```
-->chap1_list(3)
```

results in:

```
 ans  =

!   1.    0.    3. !

-->chap1_list($)
```

results in:

```
 ans  =

 this is a string
```

You can also extract in one step a coordinate of a matrix stored in a list:

```
-->chap1_list(4)($,$)
```

results in:

```
 ans  =

     .1666667
```

Lastly, you can *add* an element to a list by typing L(place)=new_element. For instance:

```
-->chap1_list(7)=part(chap1_list(5),10:16)
```

results in:

```
 chap1_list  =


       chap1_list(1)

!   1.    2.    3. !
!   4.    5.    6. !

       chap1_list(2)

!   1.    1.    1. !
!   1.    1.    1. !

       chap1_list(3)

!   1.    0.    3. !
```

```
       chap1_list(4)

!   1.       .5     .3333333 !
!    .25     .2     .1666667 !

       chap1_list(5)

 this is a string

       chap1_list(6)

    Undefined

       chap1_list(7)

  string
```

Note that the sixth element appears now as “undefined”: to avoid creating such lists with “holes” (and except you intend to do so explicitly), it is always better to use the $ symbol. The last example could for instance be rewritten as:

```
-->chap1_list($+1)=part(chap1_list(5),10:16)
```

4.7 Booleans

Boolean constants are named %t and %f (although Scilab is generally case-sensitive, %T and %F are also authorized). You can also build Boolean vectors or matrices in the same way you create constant or string matrices. For instance:

```
-->a=[%t %f %t ;%t %f %f]
```

results in:

```
 a  =


! T F T !
! T F F !
```

Extracting, modifying, deleting or adding elements from a matrix is done exactly as with constant or string matrix.

Note that the representation of booleans on screen is different from the way they are been entered.

The most common way of creating boolean vectors or matrices is however by writing conditions that apply elementwise on constant matrices. The conditions are the following ones: = = (is equal to); ~= (is different from) ; > (is greater than); >= (is greater than or equal); < (is lower than); <= (is lower than or equal). For instance:

```
-->[1:3;4:6]==ones(2,3)
```

results in:

```
 ans  =
```

```
! T F F !
! F F F !

-->[1:3;4:6] >3
```

results in:

```
ans  =

! F F F !
! T T T !
```

The operators “or” and “and” take respectively the form: | and &.

Booelans are also used in loops and conditionals: loops and conditional are mostly used in functions (see below) but can also be used interactively on screen.

Loops

Loops begin by the keyword *for* and end with the keyword *end*. The syntax is the following:
for var = <expression1>:<step>: <expression2>
….
end

<expression1>, <step>, <expression2> are expressions that result in integers. The text between the line *for <expression1>:<step>: <expression2>* and the line *end* is executed with var= <expression1>, var= <expression1>+step, var= <expression1>+2*<step> … until var is strictly greater than <expression2> (the text is then not executed). If <step> is equal to 1 (the most common case), then it can be omitted. If <step> is >0 (resp. <0) and <expression2> is strictly lower (resp. >0) than <expression1>, then the loop is not executed.

Assuming that you have a vector of residuals called resid, of length 50, then the well known Newey-West formula with a truncation lag equal to 10 can be implemented as:

```
-->for j=1:10;  nw=nw+2*(1-j/11)*(resid(1:50-j)'*resid(j+1:50)); end
```

And if you want to eliminate from a list of strings ls all elements beginning with ‘a’:

```
-->l=lengh(ls); for i=l:-1:1; if part(ls(i),1)==a then; ls(i)=null(); end
```

Note that the step –1 is used here, because it is the simplest way to eliminate (by the instruction ls(i) = null(): see in Scilab or through on-line help the entry deletion) in a list elements fulfilling certain conditions.

The *while* form of a loop also exists in Scilab. It takes the form
while <expression> then
…
end

Where <expression> is a logical expression, e.g lhs cond rhs, where cond is = = , > (greater than), > =, (greater than or equal), < (lower than), < =, (lower than or equal). <expression> can also be a boolean variable.

For instance, to calculate 50!:

```
-->f=50; j=49 ; while j>0 then ; f=f*j; j=j-1; end
```

Conditionals

Scilab allows the traditional *if*... *then*...*else*... The syntax is the following:

if <expression> then
  text1
else
  text2
end

where <expression> can have the same form than in a while loop. The else branch is optional.

For instance:

```
-->a=1;b=5;c=6;delta=b^2-4*a*c;
-->if delta >= 0 then v='the roots of a*x^2+b*x+c=0 are: '+string((b-sqrt(delta))/2/a)+' and '+string((b+sqrt(delta))/2/a);
-->else v='there is no real root to a*x^2+b*x+c=0';
-->end
-->v
v=

the roots of a*x^2+b*x+c=0 are: 2 and 3
```

Scilab allows another traditional conditional: the *select*...*case* one, whose syntax is the following:

select <expression>
case <value1> then
  <text1>
case <value2> then
  <text2>
…
case <valuen> then
  <textn>
else
  <text0>
end

Where <expression> is a constant, string… expression, <value1> until <valuen> are possible values for <expression> and <tetx1>,… <textn>, <text0> are instructions that are executed whenever <expression> has the corresponding value. Note that the else branch is not compulsory, but highly recommended: when it is lacking and the program encounters a case not foreseen in <value1> to <valuen>, it does not terminate.

For instance:

```
-->traffic_light='green';
-->select traffic_light;
-->case 'green' then v='you may go on' ;
-->case 'yellow' then v='you must stop unless you are too close to the crossroad';
-->case 'red' then 'v= you must stop';
-->else v=' error: this is not the color for a traffic light'; end;
```

There is lastly a variant of the if then else form. This is the form *elseif* which means literally *else if*:

if <expression1> then

<text1>
elseif <expression2> then
<text2>
…
elseif <expressionn> then
<textn>
else <text0>
end

This formulation avoids multiplying indentations and writing a cascade of ends: contrary to the form *else if*, the formulation *elseif* does not call an *end*; there must be only one end when the cascade of *elseif* terminates. For instance, in the example above, if you want to deal with the case when delta is 0, you should have written:

```
-->if delta > 0 then v='the roots of a*x^2+b*x+c=0 are: '+string((b-sqrt(delta))/2/a)+'
and '+string((b+sqrt(delta))/2/a);
-->elseif delta == 0 then v='the only root of a*x^2+b*x+c=0 is: '+string(b/2/a) ;
-->else v='there is no real root to a*x^2+b*x+c=0' ;
-->end
```

5. Functions

Although you can use Scilab and GROCER without writing any function, using the whole power of Scilab and GROCER should involve writing functions. GROCER itself is a collection of more than 300 functions …

In the simplest form (see part 5.3 for a more complex form), a function is a parameterized collection of Scilab commands, stored in a text file (and created with a text editor[5]) suffixed by .sci. It is highly recommended to have exactly one function per file and to give the same name to the function itself and to the text file where this function is created[6].

Once they have been compiled, either permanently (such as GROCER functions) or temporarily (see below part 5.4), functions become Scilab objects that share many characteristics with other Scilab objects. They can for instance be used as parameters of other functions. For instance, minimization of a function f can be done with the functions maxlik (GROCER function) or optim (Scilab function). Each of these functions then takes f as parameter (see chapter 5 for details).

5.1. Syntax of functions

Each function should begin with a line which has the form[7]:
function [output1, output2,…,outputn]=name(input1,…inputn)

A function may return 0, 1 or several results whose name and number have been defined or not.

If there is no result, then you can replace [] by nothing: GROCER uses mainly the last form. For instance, GROCER function drawx begins with the line:

[5] Since the release of Scilab 2.7 version, you can use the one embodied in Scilab.
[6] If you later create a compiled version of your function, then it is necessary that the function name and the fil
[7] In the rest of this part, texts of functions are signalled by garamond 11 font.

function drawx(xscale0,ref_nbinter,font_axis,y0)

If there is only one result, say output, then instead of [output]=name(…) you can write output=name(…). GROCER uses the 2 forms. For instance, GROCER function ols (see chapter 6) begins with the line:

function [rols]=ols(grocer_namey,varargin)

and GROCER function values (see chapter 3) with the line:

function val=values(ts,dat)

A function may also have several results. For instance, GROCER function auto_test (see chapter 13) begins with the line:

function [test_func,name_test]=auto_test(auto_names,auto_ltest,nobs)

Lastly, functions can have an undetermined number of results. In that case, you must write varargout=name(…). In that case varargout must be defined in the function as a list and the user will recover the results of the function as usual by writing [output1, output2,…,outputn]=name(input1,…inputn), where output1 until outputn are the expected results, provided that there are at least n elements in the varargout list. There is no function in GROCER that uses this possibility.

Similarly, a function can have as input 0,1 or several arguments.

If the function has no argument, then the term between brackets is empty. Most demo functions in GROCER have no input argument. For instance, function robust_d, which is demo of the function robust (see chapter 6) begins with the following line:

function [bsave]=robust_d()

(Note that this function has 1 output argument called bsave)

If the function has one input argument, then the term between brackets is simply this argument. For instance GROCER function date2num (see chapter 3) begins with the following line:

function num=date2num(dat)

A function can also have a variable number of input arguments. In that case, the text between brackets is varargin. For instance, GROCER function bounds (see chapter 6) begins with the following line:

function []=bounds(varargin)

The unspecified arguments can be entered along with other defined input arguments, say arg1,… ,argn. In that case, the is arg1,…,argn, varargin. Varargin must then be in the last place. For instance, GROCER function tvp (see chapter 12) begins with the following line:

function [rtvp]=tvp(grocer_namey,varargin)

Note that varargin is considered as a list and that it can be passed to another function as varargin(:).

5.2 Special function commands

The main special function commands are the following: argn, error, warning, pause, break, resume.

Command *argn* returns the number of arguments of the function: it is useful for functions with a variable number of arguments or for a function with a fixed number of arguments called with less arguments (the other have then often default values). The call is the following (see for instance function archz, described in chapter 7):

[nargout, nargin]=argn(0)
where nargout is the number of output arguments and nargin the number of input arguments.

Command *error* can be used in a function to make the program interrupt when it encounters an invalid case, apart from the ones already coded in Scilab (such as a division by 0, trying to add matrices that do not have conformable sizes,…). The syntax is the following error(msg), where *msg* is any string expression. When the function encounters this command, it stops and displays on screen the message *msg*. Examples have already been encountered above (in the select case example).

Note that when a function displays an error message, then it can be an error foreseen in the function or a Scilab error. In the first case, the message should be found in the text of the function, not in the second case.

Command *warning* has a similar syntax. Contrary to the command error, it does not stop the execution of the program. It signals that the program has encountered something curious, but that this oddity does not justify stopping the program. For instance, in function explox, you will find the following piece of text:

```
if size(grocer_indcte,2) > 1 then
   warning('only one among the '+string(size(grocer_indcte,2))+...
    ' constants you have entered has been taken into consideration')
end
```

Command *break*, not used in GROCER, forces the end of a loop. Command *pause* is used to temporarily suspend the execution of a function, which resumes when the user types “return” or “resume”. It is a very useful function for debugging: when a program pauses, you can display on screen all objects that exist in that program and therefore check easily if the program has done what it was intended to do.

Command *resume* can be used in a function to stop its execution and transmit a variable to the calling environment (working space or calling function). The syntax is [varout1,…, varoutn] = resume(varin1,… ,varinn), where varout1,…,varoutn are the names that the variables will have in the calling environment and varin1,…,varinn are the names that they have in the function. This capability, although not very clean on an pure computing point of view, is useful when you want to impose the name of the object in the calling environment. For instance, you will find in function bounds (see chapter 6) the following line:

```
grocer_boundsvar=resume(boundsvar)
```

In that case, the variable boundsvar will be transferred to the calling environment (that is the working space or another function) under the name grocer_boundsvar: this variable will then be used by any econometric program that will be called in the environment.

5.3. Defining functions directly in the environment.

You can also define directly a function in any environment (that is in the working space as well as in another function) by using the instruction deff. This is a little bit more complicated than through a file, so this capability is mainly useful in functions and not in Scilab working space. The syntax is indeed the following:

deff(‘[varout1,…, varoutn]=namefunc(varin1,…,varinp)’,’line1,…,linek’)

where varout1 to varoutn are the output arguments of the function, varin1 to varinp are the input arguments to the function and line1 to linek are the instructions of the function separated by commas.

For instance, you find in GROCER function garch the following line:
deff('[f,g,ind]=cost(p,ind)','f=garch_like(p,nar,nma,y,x),g=garch_grad2(p,nar,nma,y,x)')

Here the function cost is defined in function garch, to be used later in this function as an argument of Scilab function optim. Function cost has f,g,ind as output arguments, p,nar,nma,y,x as input arguments and the instructions are:
f=garch_like(p,nar,nma,y,x)
g=garch_grad2(p,nar,nma,y,x)'
These instructions define f and g (garch_like and garch_grad2 are GROCER functions)

5.4 Loading functions

Once a function has been written, it must be loaded in Scilab environment. This is done by the following command: getf('filename') where filename is the name of the file. Assume that your function has been written in a file named myfunc.sci in the library SCI/Mylib. Then you load it by typing:

```
-->getf('SCI/Mylib/myfunc.sci')
```

You can load all functions contained in a library by the command getd, which has the same syntax as getf. For instance, if you want to load your library SCI/Mylib, then type:

```
-->getd('SCI/Mylib')
```

Note that it can be convenient to define SCI/Mylib as your working space. This is done with the instruction:

```
-->chdir('SCI/Mylib')
```

In that case, loading all functions contained in this library involves simply the following instruction:

```
-->getd()
```

while loading all functions contained in the library SCI/Mylib/Lib1 involves the following instruction:

```
-->getd('Lib1')
```

The beginner must be aware that a modification to a function made in the corresponding text file is not taken into account by Scilab until the function has been loaded in Scilab by a getf or getd instruction. If you wonder why a bug you think to have fixed still occurs, check first that you have really loaded the corrected text!

5.5. Interruption of the execution of a function

When you execute a function in the working space, it may be interrupted (if it does not terminate, for instance) by using the menu items abort or interrupt.

6. Input/output

During a session, objects are stored in the working space. To see what objects are currently in the working space, simply type:

```
-->who
```

Data are destroyed when you leave Scilab. Saving and loading *data* (in binary format) can be made by the commands save and load. For instance saving matrices A and B in file SCI/Examples/mat.dat involves the following command:

```
-->save('SCI/Examples/mat.dat',A,B)
```

The data can be then retrieved during another session by typing:

```
-->load('SCI/Examples/mat.dat', 'A', 'B ')
```

or simply, if you want to retrieve all the data base:

```
-->load('SCI/Examples/mat.dat')
```

Data can also be destroyed when working. It is done by the command clear. For instance:

```
-->clear A; A
```

results in:

```
!--error     4
undefined variable : A
```

But:

```
-->load('SCI/Examples/mat.dat '); A
```

results in:

```
A  =

!   1.    2.    3. !
!   4.    5.    6. !
```

The commands load and save can of course also be used in functions (see for instance the text of GROCER functions impexc2bd.sci or expbd2exc.sci, that will be stored in subdirectory macros/grocer/basic of your scilab directory once you will have installed GROCER).

Saving the content of a session, that is the commands executed and their results, involves the command:

```
-->diary('name_of_a_file')
```

Once the results that interest you have been produced, close the file 'name_of_a_file' by typing:

```
-->diary(0)
```

The file 'name_of_a_file' can then be edited. Note that the file must not exist when you type the first diary; if this is not the case, then Scilab will generate an error.

7. Customizing Scilab

Since Scilab code is freely available, you can obviously customize Scilab along all directions you want, a task which assumes modifying Scilab code. This is so not the topic of this part, but there are however a few very useful things that can be made very simply.

** stacksize*

The first of these regards the stacksize. If you type the following command:

```
-->x=ones(5000,5000);
```

Then you may obtain:

```
    !--error    17
stack size exceeded! (Use stacksize function to increase it)
Memory used for variables :     29380
Intermediate memory needed:   2250002
Total  memory available   :   1000001
```

So the first thing to do is to increase the stacksize. It is highly dependent on the power of your computer:

```
-->stacksize(75000000);
```
[8]

This option can be rendered permanent by modifying the startup file (scilab.star): open this file; search 'stacksize' and modify the corresponding command.

** lines*

By default, display of results and the command or program being executed are halted after a page of results has been filled. Scilab then displays:

```
[More (y or n ) ?]
```

The display and the execution of the command or program underway resume when you type anything but N. When you type N, the display is interrupted, but the function ends normally.

If you prefer that the display goes on without interruption, you can change it by typing:
-->lines(0);

You can even make this option permanent by introducing it in the startup file…

** menus*

When you enter Scilab, a host of menus appears at the head of Scilab Window (File, Edit, Control,…) . You can add your own menus, by modifying the file wscilabf.mnu in directory bin. A convenient modification is to enter in a menu a command modifying the default working space and commands storing all functions contained in subdirectories. This should be done by adding after the bloc:

[8] The figures here (5000 and 75000000) can differ from one Scilab version and from one computer to another…

```
&A propos
                    [ABOUT]
[EndMenu]
```

the following bloc:

```
[Menu]
&access
&myworkingspace
chdir('SCI/myworkingspace');getd();
&work1
getd('work1')
&work2
getd('work2')
…
&workn
getd('workn')
[EndMenu]
```

(Everything except the keywords [Menu] and [EndMenu] and the & at the beginning of each menu definition can and must be customized by the user).