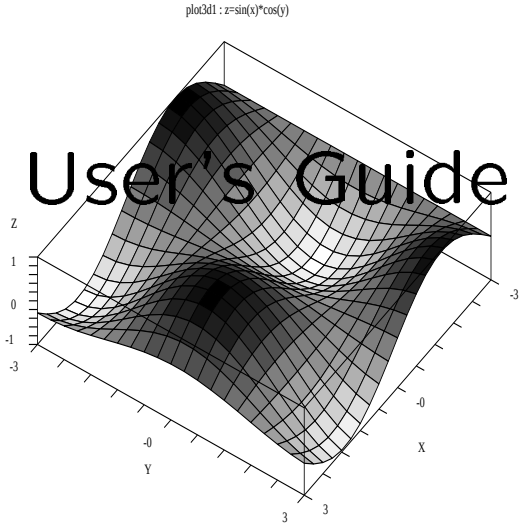
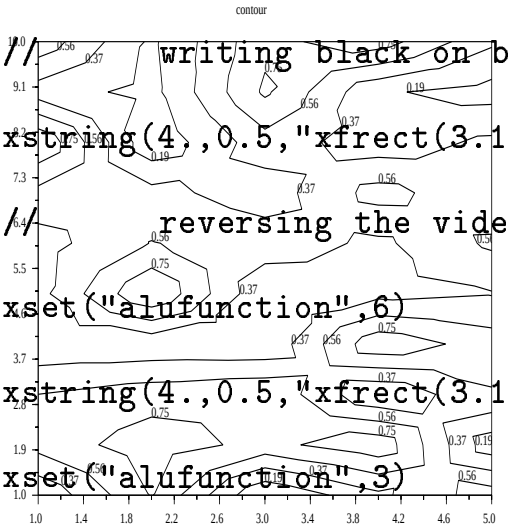
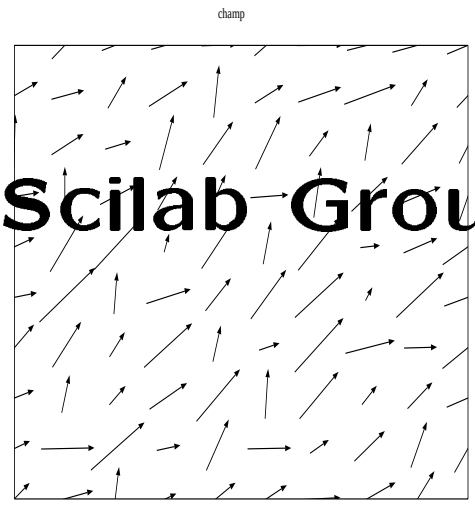
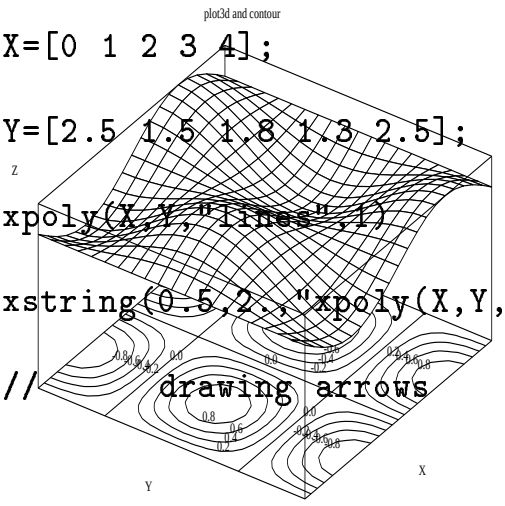
Introduction to Scilab

User's Guide









Scilab Group

```
-->plot(1:10)
-->xbasc()
-->//      simple rectangle
-->xrect(0,1,3,1)
-->//      filling a rectangle
-->xfrect(3.1,1,3,1)
-->//      writing in the rectangle
-->xstring(0.5,0.5,"xrect(0,1,3,1)")
-->//      writing black on black !
-->xstring(4.,0.5,"xfrect(3.1,1,3,1)")
-->//      reversing the video
-->xset("alufunction",6)
-->xstring(4.,0.5,"xfrect(3.1,1,3,1)")
-->xset("alufunction",3)
-->//      drawing a polyline
-->X=[0 1 2 3 4];
-->Y=[2.5 1.5 1.8 1.3 2.5];
-->xpoly(X,Y,"lines",1)
-->xstring(0.5,2.,"xpoly(X,Y,""lines""
-->//      drawing arrows
```

Table des matières

Chapitre 1

Introduction

1.1 What is Scilab

Developed at INRIA, Scilab has been developed for system control and signal processing applications. It is freely distributed in source code format (see the copyright file).

Scilab is made of three distinct parts : an interpreter, libraries of functions (Scilab procedures) and libraries of Fortran and C routines. These routines (which, strictly speaking, do not belong to Scilab but are interactively called by the interpreter) are of independent interest and most of them are available through Netlib. A few of them have been slightly modified for better compatibility with Scilab's interpreter.

A key feature of the Scilab syntax is its ability to handle matrices : basic matrix manipulations such as concatenation, extraction or transpose are immediately performed as well as basic operations such as addition or multiplication. Scilab also aims at handling more complex objects than numerical matrices. For instance, control people may want to manipulate rational or polynomial transfer matrices. This is done in Scilab by manipulating lists and typed lists which allows a natural symbolic representation of complicated mathematical objects such as transfer functions, linear systems or graphs (see Section 2.7).

Polynomials, polynomials matrices and transfer matrices are also defined and the syntax used for manipulating these matrices is identical to that used for manipulating constant vectors and matrices.

Scilab provides a variety of powerful primitives for the analysis of non-linear systems. Integration of explicit and implicit dynamic systems can be accomplished numerically. The `scicos` toolbox allows the graphic definition and simulation of complex interconnected hybrid systems.

There exist numerical optimization facilities for non linear optimization (including non differentiable optimization), quadratic optimization and linear optimization.

Scilab has an open programming environment where the creation of functions and libraries of functions is completely in the hands of the user (see Chapter 3). Functions are recognized as data objects in Scilab and, thus, can be manipulated or created as other data objects. For example, functions can be defined inside Scilab and passed as input or output arguments of other functions.

In addition Scilab supports a character string data type which, in particular, allows the on-line creation of functions. Matrices of character strings are also manipulated with the same syntax as ordinary matrices.

Finally, Scilab is easily interfaced with Fortran or C subprograms. This allows use of standardized packages and libraries in the interpreted environment of Scilab.

The general philosophy of Scilab is to provide the following sort of computing environment :

– To have data types which are varied and flexible with a syntax which is natural and easy to use.
– To provide a reasonable set of primitives which serve as a basis for a wide variety of calculations.
– To have an open programming environment where new primitives are easily added. A useful tool distributed with Scilab is `intersci` which is a tool for building interface programs to add new primitives i.e. to add new modules of Fortran or C code into Scilab.
– To support library development through "toolboxes" of functions devoted to specific applications (linear control, signal processing, network analysis, non-linear control, etc.)

The objective of this introduction manual is to give the user an idea of what Scilab can do. On line documentation on all functions is available (`help` command).

1.2 Software Organization

Scilab is divided into a set of directories. The main directory `SCIDIR` contains the following files : `scilab.star` (startup file), the copyright file `notice.tex`, and the `configure` files (see(1.3)). The subdirectories are the following :

– `bin` is the directory of the executable files. The starting script `scilab` on Unix/Linux systems and `runscilab.exe` on Windows95/NT, The executable code of Scilab : `scilex` on Unix/Linux systems and `scilex.exe` on Windows95/NT are there. This directory also contains Shell scripts for managing or printing Postscript/LaTeX files produced by Scilab.
– `demos` is the directory of demos. This directory contains the codes corresponding to various demos. They are often useful for inspiring new users. The file `alldems.dem` is used by the "Demos" button. Most of plot commands are illustrated by simple demo examples. Note that running a graphic function without input parameter provides an example of use for this function (for instance `plot2d()` displays an example for using `plot2d` function).
– `examples` contains useful examples of how to link external programs to scilab, using dynamic link or `intersci`
– `doc` is the directory of the Scilab documentation : LaTeX , dvi and Postscript files. This documentation is `SCIDIR/doc/intro/intro.tex`.
– `geci` contains source code and binaries for GeCI which is an interactive communication manager created in order to manage remote executions of softwares and allow exchanges of messages beetwen those programs. It offers the possibility to exploit numerous machines on a network, as a virtual computer, by creating a distributed group of independent softwares (`help communications` for a detailed description). GeCI is used for the link of Xmetanet with Scilab.
– `pvm3` contains source code and binaries of the PVM version 3 which is another interactive communication manager.
– `imp` is the directory of the routines managing the Postscript files for print.
– `libs` contains the Scilab libraries (compiled code).
– `macros` contains the libraries of functions which are available on-line. New libraries can easily be added (see the Makefile). This directory is divided into a number of subdirectories which contain "Toolboxes" for control, signal processing, etc... Strictly speaking Scilab is not organized in toolboxes : functions of a specific subdirectory can call functions of other directories ; so, for example, the subdirectory `signal` is not self-contained but its functions are all devoted to signal processing.

- `man` is the directory containing the manual divided into submanuals, corresponding to the on-line help and to a LaTeX format of the reference manual. The LaTeX code is produced by a translation of the Unix format Scilab manual (see the subdirectory `SCIDIR/man`). To get information about an item, one should enter `help item` in Scilab or use the help window facility obtained with help button. To get information corresponding to a key-word, one should enter `apropos key-word` or use `apropos` in the help window. All the `item`s and `key-words` known by the `help` and `apropos` commands are in `.cat` and `whatis` files located in the `man` subdirectories.
  To add new items to the `help` and `apropos` commands the user can extend the list of directories available to the help browser by adapting the variable `%helps`. See the README file in the `man` directory and the example given in `examples/man-examples` directory
- `maple` is the directory which contains the source code of Maple functions which allow the transfer of Maple objects into Scilab functions. For efficiency, the transfer is made through Fortran code generation which is dynamically linked to Scilab.
- `routines` is a directory which contains the source code of all the numerical routines. The subdirectory `default` is important since it contains the source code of routines which are necessary to customize Scilab. In particular user's C or Fortran routines for ODE/DAE simulation or optimization can be included here (they can be also dynamically linked).
- `examples` contains examples of specific topics. It is shown in appropriate subdirectories how to add new C or Fortran program to Scilab (see `addinter-tutorial`). More complex examples are given in `addinter-examples`. The directory `mex-examples` contains examples of interfaces realized by emulating the Matlab mexfiles. The directory `link-examples` illustrates the use of the `call` function which allows to call external function within Scilab.
- `intersci` contains a program which can be used to build interface programs for adding new Fortran or C primitives to Scilab. This program is executed by the `intersci` script in the `bin/intersci` directory.
- `scripts` is the directory which contains the source code of shell scripts files. Note that the list of printers names known by Scilab is defined there by an environment variable.
- `tests` : this directory contains evaluation programs for testing Scilab's installation on a machine. The file "demos.tst" tests all the demos.
- `wless, xless` is the Berkeley file browsing tool
- `xmetanet` is the directory which contains `xmetanet`, a graphic display for networks. Type `metanet()` in Scilab to use it.

1.3 Installing Scilab. System Requirements

Scilab is distributed in source code format ; binaries for Windows95/NT systems and several popular Unix/Linux-XWindow systems are also available : Dec Alpha (OSF V4), Dec Mips (ULTRIX 4.2), Sun Sparc stations (Sun OS), Sun Sparc stations (Sun Solaris), HP9000 (HP-UX V10), SGI Mips Irix, PC Linux. All of these binaries versions include tk/tcl interface.

The installation requirements are the following :

- for the source version : Scilab requires approximately 130Mb of disk storage to unpack and install (all sources included). You need X Window (X11R4, X11R5 or X11R6, C compiler and Fortran compiler (e.g. f2c or g77 or Visual C++ for Windows systems).

- for the binary version : the minimum for running Scilab (without sources) is about 40 Mb when decompressed. These versions are partially statically linked and in principle do not require a fortran compiler.

Scilab uses a large internal stack for its calculations. This size of this stack can be reduced or enlarged by the `stacksize`. command. The default dimension of the internal stack can be adapted by modifying the variable `newstacksize` in the `scilab.star` script.

- For more information on the installation, please look at the README files

1.4 Documentation

The documentation is made of this User's guide (Introduction to Scilab) and the Scilab on-line manual. There are also reports devoted to specific toolboxes : Scicos (graphic system builder and simulator), Signal (Signal processing toolbox), Lmitool (interface for LMI problems), Metanet (graph and network toolbox). An FAQ is available at Scilab home page :
(`http://www-rocq.inria.fr/scilab`).

1.5 Scilab at a Glance. A Tutorial

1.5.1 Getting Started

Scilab is called by running the `scilab` script in the directory `SCIDIR/bin` (`SCIDIR` denotes the directory where Scilab is installed). This shell script runs Scilab in an Xwindow environment (this script file can be invoked with specific parameters such as `-nw` for "no-window"). You will immediatly get the Scilab window with the following banner and prompt represented by the `-->` :

```
           ===========
           S c i l a b
           ===========


      Scilab-2.x  ( 12 July 1998 )
      Copyright (C) 1989-98 INRIA


Startup execution:
  loading initial environment

 -->
```

A first contact with Scilab can be made by clicking on `Demos` with the left mouse button and clicking then on `Introduction to SCILAB` : the execution of the session is then done by entering empty lines and can be stopped with the buttons `Stop` and `Abort`.

Several libraries (see the `SCIDIR/scilab.star` file) are automatically loaded.

To give the user an idea of some of the capabilities of Scilab we will give later a sample session in Scilab.

1.5.2 Editing a command line

Before the sample session, we briefly present how to edit a command line. You can enter a command line by typing after the prompt or clicking with the mouse on a part on a window and copy it at the prompt in the Scilab window. The pointer may be moved using the directionnal arrows ($\leftarrow\updownarrow\rightarrow$). For Emacs customers, the usual Emacs commands are at your disposal for modifying a command (Ctrl-<chr> means hold the CONTROL key while typing the character <chr>), for example :

- Ctrl-p recall previous line
- Ctrl-n recall next line
- Ctrl-b move backward one character
- Ctrl-f move forward one character
- Delete delete previous character
- Ctrl-h delete previous character
- Ctrl-d delete one character (at cursor)
- Ctrl-a move to beginning of line
- Ctrl-e move to end of line
- Ctrl-k delete to the end of the line
- Ctrl-u cancel current line
- Ctrl-y yank the text previously deleted
- !`prev` recall the last command line which begins by `prev`
- Ctrl-c interrupt Scilab and pause after carriage return. Clicking on the Control/stop button enters a Ctrl-c.

As said before you can also cut and paste using the mouse. This way will be useful if you type your commands in an editor. Another way to "load" files containing Scilab statements is available with the `File/File Operations` button.

1.5.3 Buttons

The Scilab window has the following **Control** buttons.

- Stop interrupts execution of Scilab and enters in `pause` mode
- Resume continues execution after a `pause` entered as a command in a function or generated by the `Stop` button or Control C.
- Abort aborts execution after one (or several) `pause`, and returns to top-level prompt
- Restart clears all variables and executes startup files
- Quit quits Scilab
- Kill kills Scilab shell script
- Demos for interactive run of some demos
- File Operations facility for loading functions or data into Scilab, or executing script files.
- Help : invokes on-line help with the tree of the man and the names of the corresponding items. It is possible to type directly `help <item>` in the Scilab window.
- Graphic Window : select active graphic window

New buttons can be added by the `addmenu` command. Note that the command :
`SCIDIR/bin/scilab -nw`
invokes Scilab in the "no-window" mode.

1.5.4 Customizing your Scilab - Unix only

The parameters of the different windows opened by Scilab can be easily changed. The way for doing that is to edit the files contained in the directory `X11-defaults`. The first possibility is to directly customize these files. Another way is to copy the right lines with the modifications in the `.Xdefaults` file of the home directory. These modifications are activated by starting again Xwindow or with the command `xrdb .Xdefaults`. Scilab will read the `.Xdefaults` file : the lines of this file will cancel and replace the corresponding lines of X11-defaults.

A simple example :

```
Xscilab.color*Scrollbar.background:red
Xscilab*vpane.height: 500
Xscilab*vpane.width:  500
```

in `.Xdefaults` will change the 500x650 window to a square window of 500x500 and the scrollbar background color changes from green to red.

An important parameter for customizing Scilab is `stacksize` discussed in 1.3.

1.5.5 Sample Session for Beginners

We present now some simple commands. At the carriage return all the commands typed since the last prompt are interpreted.

.....

```
-->a=1;

-->A=2;

-->a+A
 ans  =

    3.

-->//Two commands on the same line

-->c=[1 2];b=1.5
 b  =

    1.5

-->//A command on several lines

-->u=1000000*(a*sin(A))^2+...
-->     2000000*a*b*sin(A)*cos(A)+...
-->     1000000*(b*cos(A))^2
 u  =

    81268.994
```

Give the values of 1 and 2 to the variables `a` and `A` . The semi-colon at the end of the command suppresses the display of the result. Note that Scilab is case-sensitive. Then two commands are processed and the second result is displayed because it is not followed by a semi-colon. The last command shows how to write a command on several lines by using "`...`". This sign is only needed in the on-line typing for avoiding the effect of the carriage return. The chain of characters which follow the `//` is not interpreted (it is a comment line).

.....

```
-->a=1;b=1.5;

-->2*a+b^2
 ans  =

    4.25

-->//We have now created variables and can list them by typing:

-->who
your variables are...

 ans       b         a         bugmes    scicos_pal          demolist
 %helps    LANGUAGE  MSDOS     home      PWD       TMPDIR    xdesslib
 percentlib          polylib   intlib    elemlib   utillib   statslib
 alglib    siglib    optlib    autolib   roblib    soundlib  metalib
 armalib   tkscilib  tdcslib   s2flib    mtlblib   SCI       %F
 %T        %z        %s        %nan      %inf      COMPILER  %gtk
 %pvm      %tk       $         %t        %f        %eps      %io
 %i        %e
 using       5358 elements  out of    1000000.
          and         50 variables out of       1791

your global variables are...

 LANGUAGE  %helps    demolist  %browsehelp         %toolboxes
 %toolboxes_dir
 using       1421 elements  out of       1661.
          and          6 variables out of        255
```

We get the list of previously defined variables `a b c A` together with the initial environment composed of the different libraries and some specific "permanent" variables.

Below is an example of an expression which mixes constants with existing variables. The result is retained in the standard default variable `ans`.

.....

```
-->I=1:3
 I  =

!   1.    2.    3. !

-->W=rand(2,4);

-->W(1,I)
 ans  =

!   0.2113249    0.0002211    0.6653811 !

-->W(:,I)
 ans  =

!   0.2113249    0.0002211    0.6653811 !
!   0.7560439    0.3303271    0.6283918 !

-->W($,$-1)
 ans  =

    0.6283918
```

Defining `I`, a vector of indices, `W` a random 2 x 4 matrix, and extracting submatrices from `W`. The `$` symbol stands for the last row or last column index of a matrix or vector. The colon symbol stands for "all rows" or "all columns".

.....

```
-->sqrt([4  -4])
 ans  =

!   2.    2.i !
```

Calling a function (or primitive) with a vector argument. The response is a complex vector.

.....

```
-->p=poly([1 2 3],'z','coeff')
 p  =

                2
    1 + 2z + 3z

-->//p is the polynomial in z with coefficients 1,2,3.
```

```
-->//p can also be defined by :

-->s=poly(0,'s');p=1+2*s+s^2
 p  =

              2
    1 + 2s + s
```

A more complicated command which creates a polynomial.

.....

```
-->M=[p, p-1; p+1 ,2]
 M  =

!             2          2 !
!   1 + 2s + s     2s + s  !
!                          !
!             2            !
!   2 + 2s + s     2       !

-->det(M)
 ans  =

          2    3   4
    2 - 4s - 4s - s
```

Definition of a polynomial matrix. The syntax for polynomial matrices is the same as for constant matrices. Calculation of the determinant of the polynomial matrix by the `det` function.

.....

```
-->F=[1/s    ,(s+1)/(1-s)
-->      s/p    ,   s^2  ]
 F  =

!   1              1 + s  !
!   -              -----  !
!   s              1 - s  !
!                         !
!                   2     !
!       s           s     !
!   ---------       -     !
```

```
!             2           !
!   1 + 2s + s     1      !
```

```
-->F.num
 ans  =

!   1     1 + s  !
!                !
!          2     !
!   s     s      !
```

```
-->F.den
 ans  =

!   s              1 - s  !
!                        !
!             2          !
!   1 + 2s + s     1      !
```

```
-->F.num(1,2)
 ans  =

    1 + s
```

Definition of a matrix of rational polynomials. (The internal representation of `F` is a typed list of the form `tlist('the type',num,den)` where `num` and `den` are two matrix polynomials). Retrieving the numerator and denominator matrices of `F` by extraction operations in a typed list. Last command is the direct extraction of entry `1,2` of the numerator matrix `F.num`.

.....

```
-->pause

-1->pt=return(s*p)

-->pt
 pt  =

          2   3
    s + 2s + s
```

Here we move into a new environment using the command `pause` and we obtain the new prompt `-1->` which indicates the level of the new environment (level 1). All variables that are available in the first environment are also available in the new environment. Variables created in the new environment can be returned to the original environment by using `return`. Use of `return` without

an argument destroys all the variables created in the new environment before returning to the old environment. The `pause` facility is very useful for debugging purposes.

.....

```
-->F21=F(2,1);v=0:0.01:%pi;frequencies=exp(%i*v);

-->response=freq(F21.num,F21.den,frequencies);

-->plot2d(v,abs(response),style=-1,rect=[0,0,3.5,0.7],nax=[5,4,5,7]);

-->xtitle(' ','radians','magnitude');
```

Definition of a rational polynomial by extraction of an entry of the matrix `F` defined above. This is followed by the evaluation of the rational polynomial at the vector of complex frequency values defined by `frequencies`. The evaluation of the rational polynomial is done by the primitive `freq`. `F12.num` is the numerator polynomial and `F12.den` is the denominator polynomial of the rational polynomial `F12`. Note that the polynomial `F12.num` can be also obtained by extraction from the matrix `F` using the syntax `F.num(1,2)`. The visualization of the resulting evaluation is made by using the basic plot command `plot2d` (see Figure 1.1).

.....

```
-->w=(1-s)/(1+s);f=1/p
 f  =

        1
    ---------
              2
    1 + 2s + s

-->horner(f,w)
 ans  =

              2
    1 + 2s + s
    ----------
        4
```

The function `horner` performs a (possibly symbolic) change of variables for a polynomial (for example, here, to perform the bilinear transformation f(w(s))).

.....

```
-->A=[-1,0;1,2];B=[1,2;2,3];C=[1,0];

-->Sl=syslin('c',A,B,C);

-->ss2tf(Sl)
 ans  =

!     1         2    !
!   -----     -----  !
!   1 + s     1 + s  !
```

Definition of a linear system in state-space representation. The function `syslin` defines here the continuous time (`'c'`) system `Sl` with state-space matrices (`A,B,C`). The function `ss2tf` transforms `Sl` into transfer matrix representation.

.....

```
-->s=poly(0,'s');

-->R=[1/s,s/(1+s),s^2]
 R  =

!                     2 !
!   1       s       s  !
!   -     -----     -  !
!   s     1 + s     1  !

-->Sl=syslin('c',R);

-->tf2ss(Sl)
 ans  =


       ans(1)   (state-space system:)

!lss  A  B  C  D  X0  dt  !

       ans(2) = A matrix =

! - 0.5  - 0.5 !
! - 0.5  - 0.5 !

       ans(3) = B matrix =

! - 1.    1.    0. !
!   1.    1.    0. !
```

```
       ans(4) = C matrix =

! - 1.    1.212D-16 !

       ans(5) = D matrix =

!               2 !
!   0    1     s  !

       ans(6) = X0 (initial state) =

!   0. !
!   0. !

       ans(7) = Time domain =

 c
```

Definition of the rational matrix `R`. `Sl` is the continuous-time linear system with (improper) transfer matrix `R`. `tf2ss` puts `Sl` in state-space representation with a polynomial `D` matrix. Note that linear systems are represented by specific typed lists (with 7 entries).

.....

```
-->sl1=[Sl;2*Sl+eye()]
 sl1  =

!                         2  !
!   1           s       s    !
!   -         -----     -    !
!   s         1 + s     1    !
!                            !
!                           2 !
!   2 + s      2s       2s   !
!   -----     ----     ---   !
!     s       1 + s     1    !

-->size(sl1)
 ans  =

!   2.    3. !

-->size(tf2ss(sl1))
 ans  =
```

```
!   2.    3. !
```

`sl1` is the linear system in transfer matrix representation obtained by the parallel inter-connection of `Sl` and `2*Sl +eye()`. The same syntax is valid with `Sl` in state-space representation.

.....

```
-->function Cl=compen(Sl,Kr,Ko)
-->  [A,B,C,D]=abcd(Sl);
-->  A1=[A-B*Kr ,B*Kr; 0*A ,A-Ko*C]; Id=eye(A);
-->  B1=[B; 0*B];
-->  C1=[C ,0*C];Cl=syslin('c',A1,B1,C1)
-->endfunction
```

On-line definition of a function, called `compen` which calculates the state space representation (`Cl`) of a linear system (`Sl`) controlled by an observer with gain `Ko` and a controller with gain `Kr`. Note that matrices are constructed in block form using other matrices.

.....

```
-->A=[1,1 ;0,1];B=[0;1];C=[1,0];Sl=syslin('c',A,B,C);

-->Cl=compen(Sl,ppol(A,B,[-1,-1]),...
-->                    ppol(A',C',[-1+%i,-1-%i])');

-->Aclosed=Cl.A,spec(Aclosed)
 Aclosed  =

!   1.    1.    0.    0. !
! - 4.  - 3.    4.    4. !
!   0.    0.  - 3.    1. !
!   0.    0.  - 5.    1. !
 ans  =

! - 1.        !
! - 1.0000000 !
! - 1. + i    !
! - 1. - i    !
```

Call to the function `compen` defined above where the gains were calculated by a call to the primitive `ppol` which performs pole placement. The resulting `Aclosed` matrix is displayed and the placement of its poles is checked using the primitive `spec` which calculates the eigenvalues of a matrix. (The function `compen` is defined here on-line by as an example of function which receive a

linear system (`Sl`) as input and returns a linear system (`Cl`) as output. In general Scilab functions are defined in files and loaded in Scilab by `exec` or by `getf`).

.....

```
-->//Saving the environment in a file named : myfile

-->save('myfile')

-->//Request to the host system to perform a system command

-->unix_s('rm myfile')

-->//Request to the host system with output in this Scilab window

-->unix_w('date')
Thu Oct 30 15:27:45 CET 2003
```

Relation with the Unix environment.

.....

```
-->foo=['void foo(a,b,c)';
-->     'double *a,*b,*c;'
-->      '{ *c = *a + *b;}']
 foo  =

!void foo(a,b,c)   !
!                  !
!double *a,*b,*c;  !
!                  !
!{ *c = *a + *b;}  !

-->//A 3 x 1 matrix of strings

-->write('foo.c',foo);      //Editing

-->unix_s('make foo.o')     //Compiling

-->link('foo.o','foo','C'); //Dynamic link
shared archive loaded

Link done


-->//On line definition of myplus function.
```

```
-->//(Calling external C code).

-->deff('[c]=myplus(a,b)',...
-->   'c=call(''foo'',a,1,''d'',b,2,''d'',''out'',[1,1],3,''d'')')

-->myplus(5,7)
 ans  =

    12.
```

Definition of a column vector of character strings used for defining a C function file. The routine is compiled (needs a compiler), dynamically linked to Scilab by the `link` command, and interactively called by the function `myplus`.

.....

```
-->function ydot=f(t,y),ydot=[a-y(2)*y(2) -1;1 0]*y,endfunction

-->a=1;y0=[1;0];t0=0;instants=0:0.02:20;

-->y=ode(y0,t0,instants,f);

-->plot2d(y(1,:),y(2,:),style=-1,rect=[-3,-3,3,3],nax=[10,2,10,2])

-->xtitle('Van der Pol')
```

Definition of a function which calculates a first order vector differential `f(t,y)`. This is followed by the definition of the constant `a` used in the function. The primitive `ode` then integrates the differential equation defined by the Scilab function `f(t,y)` for `y0=[1 ;0]` at `t=0` and where the solution is given at the time values $t = 0, .02, .04, \ldots, 20$. (Function `f` can be defined as a C or Fortran program). The result is plotted in Figure 1.2 where the first element of the integrated vector is plotted against the second element of this vector.

.....

```
-->m=['a' 'cos(b)';'sin(a)' 'c']
 m  =

!a       cos(b)  !
!                !
!sin(a)  c       !

-->//m*m'   --> error message : not implemented in scilab

-->function x=%c_m_c(a,b)
```

```
-->  [l,m]=size(a);[m,n]=size(b);x=[];
-->  for j=1:n,
-->    y=[];
-->    for i=1:l,
-->      t=' ';
-->      for k=1:m;
-->if k>1 then
-->  t=t+'+('+a(i,k)+')*'+'('+b(k,j)+')';
-->else
-->  t='(' + a(i,k) + ')*' + '(' + b(k,j) + ')';
-->end
-->      end
-->      y=[y;t]
-->    end
-->    x=[x y]
-->  end
-->endfunction

-->m*m'
 ans  =

!(a)*(a)+(cos(b))*(cos(b))  (a)*(sin(a))+(cos(b))*(c)  !
!                                                      !
!(sin(a))*(a)+(c)*(cos(b))  (sin(a))*(sin(a))+(c)*(c)  !
```

Definition of a matrix containing character strings. By default, the operation of symbolic multiplication of two matrices of character strings is not defined in Scilab. However, the (on-line) function definition for `%cmc` defines the multiplication of matrices of character strings. The `%` which begins the function definition for `%cmc` allows the definition of an operation which did not previously exist in Scilab, and the name `cmc` means "chain multiply chain". This example is not very useful : it is simply given to show how *operations* such as `*` can be defined on complex data structures by mean of scpecific Scilab functions.

.....

```
-->function y=calcul(x,method),z=method(x),y=poly(z,'x'),endfunction

-->function z=meth1(x),z=x,endfunction

-->function z=meth2(x),z=2*x,endfunction

-->calcul([1,2,3],meth1)
 ans  =

                2   3
  - 6 + 11x - 6x + x
```

```
-->calcul([1,2,3],meth2)
 ans  =

                  2   3
  - 48 + 44x - 12x + x
```

A simple example which illustrates the passing of a function as an argument to another function. Scilab functions are objects which may be defined, loaded, or manipulated as other objects such as matrices or lists.

. . . . . .

```
-->quit
```

Exit from Scilab.

. . . . . .

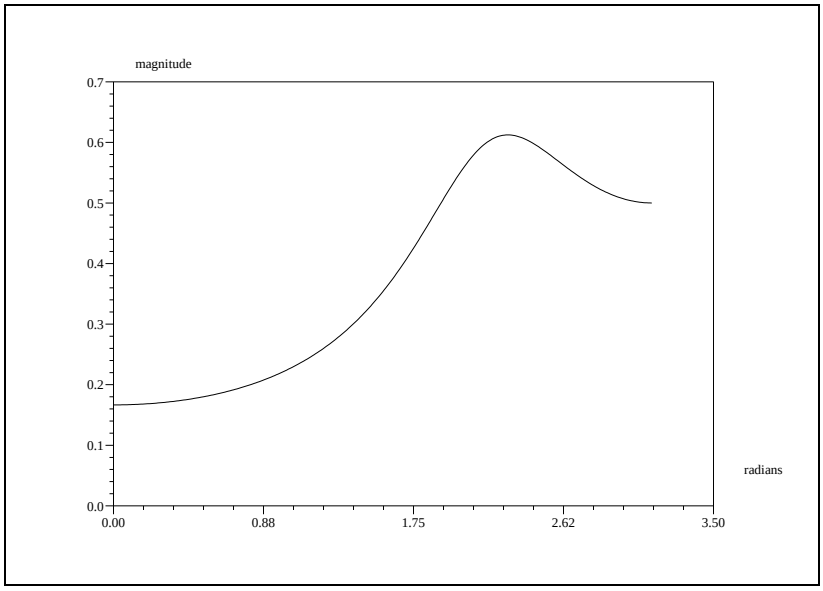


FIG. 1.1 – A Simple Response

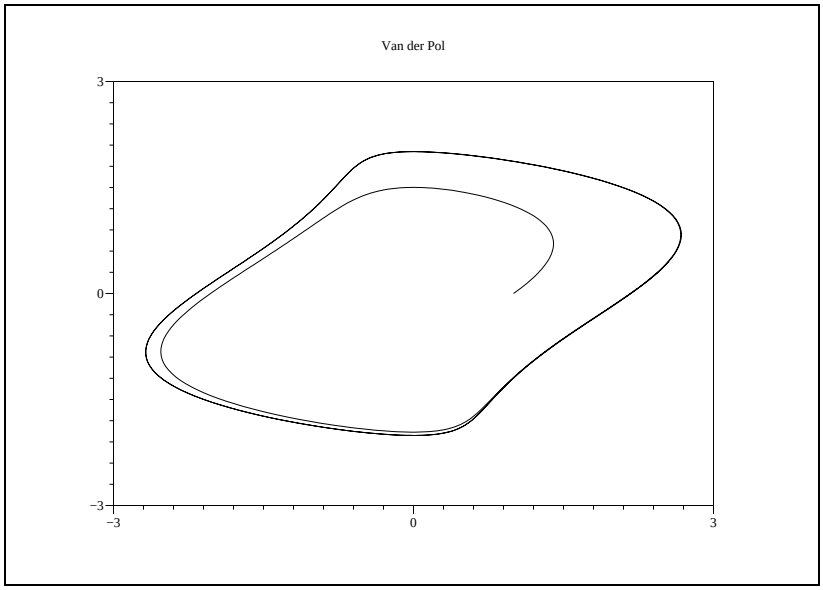


FIG. 1.2 – Phase Plot

Chapitre 2

Data Types

Scilab recognizes several data types. Scalar objects are constants, booleans, polynomials, strings and rationals (quotients of polynomials). These objects in turn allow to define matrices which admit these scalars as entries. Other basic objects are lists, typed-lists and functions. Only constant and boolean sparse matrices are defined. The objective of this chapter is to describe the use of each of these data types.

2.1 Special Constants

Scilab provides special constants `%i`, `%pi`, `%e`, and `%eps` as primitives. The `%i` constant represents $\sqrt{-1}$, `%pi` is $\pi = 3.1415927\cdots$, `%e` is the trigonometric constant $e = 2.7182818\cdots$, and `%eps` is a constant representing the precision of the machine (`%eps` is the biggest number for which $1 +$ `%eps` $= 1$). `%inf` and `%nan` stand for "Infinity" and "NotANumber" respectively. `%s` is the polynomial s=`poly(0,'s')` with symbol `s`.

(More generally, given a vector `rts`, `p=poly(rts,'x')` defines the polynomial p(x) with variable `x` and such that `roots(p)` = `rts`).

Finally boolean constants are `%t` and `%f` which stand for "true" and "false" respectively. Note that `%t` is the same as `1==1` and `%f` is the same as `~%t`.

These variables are considered as "predefined". They are protected, cannot be deleted and are not saved by the `save` command. It is possible for a user to have his own "predefined" variables by using the `predef` command. The best way is probably to set these special variables in his own startup file `<home dir>/.scilab`. Of course, the user can use e.g. `i=sqrt(-1)` instead of `%i`.

2.2 Constant Matrices

Scilab considers a number of data objects as matrices. Scalars and vectors are all considered as matrices. The details of the use of these objects are revealed in the following Scilab sessions.

Scalars Scalars are either real or complex numbers. The values of scalars can be assigned to variable names chosen by the user.

```
--> a=5+2*%i
 a         =

    5. + 2.i
```

```
--> B=-2+%i;

--> b=4-3*%i
 b  =

    4. - 3.i

--> a*b
 ans  =

    26. - 7.i

-->a*B
 ans  =

  - 12. + i
```

Note that Scilab evaluates immediately lines that end with a carriage return. Instructions that ends with a semi-colon are evaluated but are not displayed on screen.

Vectors The usual way of creating vectors is as follows, using commas (or blanks) and semi-columns :

```
--> v=[2,-3+%i,7]
 v         =

!   2.  - 3. + i      7. !

--> v'
 ans       =

!   2.       !
! - 3. - i   !
!   7.       !

--> w=[-3;-3-%i;2]
 w         =

! - 3.       !
! - 3. - i   !
!   2.       !

--> v'+w
 ans       =

! - 1.       !
```

```
! - 6. - 2.i !
!   9.       !

--> v*w
 ans       =

    18.

--> w'.*v
 ans       =

! - 6.    8. - 6.i    14. !
```

Notice that vector elements that are separated by commas (or by blanks) yield row vectors and those separated by semi-colons give column vectors. The empty matrix is `[]` ; it has zero rows and zero columns. Note also that a single quote is used for transposing a vector (one obtains the complex conjugate for complex entries). Vectors of same dimension can be added and subtracted. The scalar product of a row and column vector is demonstrated above. Element-wise multiplication (`.*`) and division (`./`) is also possible as was demonstrated.

Note with the following example the role of the position of the blank :

```
-->v=[1 +3]
 v  =

!   1.    3. !

-->w=[1 + 3]
 w  =

!   1.    3. !

-->w=[1+ 3]
 w  =

    4.

-->u=[1, + 8- 7]
 u  =

!   1.    1. !
```

Vectors of elements which increase or decrease incrementely are constructed as follows

```
--> v=5:-.5:3
 v         =

!   5.    4.5    4.    3.5    3. !
```

The resulting vector begins with the first value and ends with the third value stepping in increments of the second value. When not specified the default increment is one. A constant vector can be created using the `ones` and `zeros` facility

```
--> v=[1 5 6]
 v         =

!   1.    5.    6. !

--> ones(v)
 ans       =

!   1.    1.    1. !

--> ones(v')
 ans       =

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

--> ones(1:4)
 ans       =

!   1.    1.    1.    1. !

--> 3*ones(1:4)
 ans       =

!   3.    3.    3.    3. !

-->zeros(v)
 ans  =

!   0.    0.    0. !

-->zeros(1:5)
 ans  =

!   0.    0.    0.    0.    0. !
```

Notice that `ones` or `zeros` replace its vector argument by a vector of equivalent dimensions filled with ones or zeros.

Matrices Row elements are separated by commas or spaces and column elements by semi-colons. Multiplication of matrices by scalars, vectors, or other matrices is in the usual sense. Addition

and subtraction of matrices is element-wise and element-wise multiplication and division can be accomplished with the `.*` and `./` operators.

```
--> A=[2 1 4;5 -8 2]
 A         =

!   2.    1.    4. !
!   5.  - 8.    2. !

--> b=ones(2,3)
 b         =

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

--> A.*b
 ans       =

!   2.    1.    4. !
!   5.  - 8.    2. !

--> A*b'
 ans       =

!   7.    7. !
! - 1.  - 1. !
```

Notice that the `ones` operator with two real numbers as arguments separated by a comma creates a matrix of ones using the arguments as dimensions (same for `zeros`). Matrices can be used as elements to larger matrices. Furthermore, the dimensions of a matrix can be changed.

```
--> A=[1 2;3 4]
 A         =

!   1.    2. !
!   3.    4. !

--> B=[5 6;7 8];

--> C=[9 10;11 12];

--> D=[A,B,C]
 D         =

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

```
--> E=matrix(D,3,4)
 E         =

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

-->F=eye(E)
 F  =

!   1.    0.    0.    0. !
!   0.    1.    0.    0. !
!   0.    0.    1.    0. !

-->G=eye(4,3)
 G  =

!   1.    0.    0. !
!   0.    1.    0. !
!   0.    0.    1. !
!   0.    0.    0. !
```

Notice that matrix `D` is created by using other matrix elements. The `matrix` primitive creates a new matrix `E` with the elements of the matrix `D` using the dimensions specified by the second two arguments. The element ordering in the matrix `D` is top to bottom and then left to right which explains the ordering of the re-arranged matrix in `E`.

The function `eye` creates an $m \times n$ matrix with 1 along the main diagonal (if the argument is a matrix `E` , m and n are the dimensions of `E`) .

Sparse constant matrices are defined through their nonzero entries (type help `sparse` for more details). Once defined, they are manipulated as full matrices.

2.3 Matrices of Character Strings

Character strings can be created by using single or double quotes. Concatenation of strings is performed by the `+` operation. Matrices of character strings are constructed as ordinary matrices, e.g. using brackets. A very important feature of matrices of character strings is the capacity to manipulate and create functions. Furthermore, symbolic manipulation of mathematical objects can be implemented using matrices of character strings. The following illustrates some of these features.

```
--> A=['x' 'y';'z' 'w+v']
 A         =

!x  y    !
!        !
!z  w+v  !
```

```
--> At=trianfml(A)
 At        =

!z  w+v          !
!                !
!0  z*y-x*(w+v)  !

--> x=1;y=2;z=3;w=4;v=5;

--> evstr(At)
 ans       =

!   3.    9. !
!   0.  - 3. !
```

Note that in the above Scilab session the function `trianfml` performs the symbolic triangularization of the matrix `A`. The value of the resulting symbolic matrix can be obtained by using `evstr`.

A very important aspect of character strings is that they can be used to automatically create new functions (for more on functions see Section 3.2). An example of automatically creating a function is illustrated in the following Scilab session where it is desired to study a polynomial of two variables `s` and `t`. Since polynomials in two independent variables are not directly supported in Scilab, we can construct a new data structure using a list (see Section 2.7). The polynomial to be studied is $(t^2 + 2t^3) - (t + t^2)s + ts^2 + s^3$.

```
-->getf("macros/make_macro.sci");

-->s=poly(0,'s');t=poly(0,'t');

-->p=list(t^2+2*t^3,-t-t^2,t,1+0*t);

-->pst=makefunction(p) //pst is a function t->p (number->polynomial)
 pst       =

[p]=pst(t)

-->pst(1)
 ans       =

              2   3
    3 - 2s + s + s
```

Here the polynomial is represented by the command which puts the coefficients of the variable `s` in the list `p`. The list `p` is then processed by the function `makefunction` which makes a new function `pst`. The contents of the new function can be displayed and this function can be evaluated at values of `t`. The creation of the new function `pst` is accomplished as follows

```
function [newfunction]=makefunction(p)
// Copyright INRIA
```

```
num=mulf(makestr(p(1)),'1');
for k=2:size(p);
   new=mulf(makestr(p(k)),'s^'+string(k-1));
   num=addf(num,new);
end,
text='p='+num;
deff('[p]=newfunction(t)',text),


function [str]=makestr(p)
n=degree(p)+1;c=coeff(p);str=string(c(1));x=part(varn(p),1);
xstar=x+'^',
for k=2:n,
   if c(k)<>0 then,
   str=addf(str,mulf(string(c(k)),(xstar+string(k-1))));
   end;
end
```

Here the function `makefunction` takes the list `p` and creates the function `pst`. Inside of `makefunction` there is a call to another function `makestr` which makes the string which represents each term of the new two variable polynomial. The functions `addf` and `mulf` are used for adding and multiplying strings (i.e. `addf(x,y)` yields the string `x+y`). Finally, the essential command for creating the new function is the primitive `deff`. The `deff` primitive creates a function defined by two matrices of character strings. Here the function `p` is defined by the two character strings `'[p]=newfunction(t)'` and `text` where the string `text` evaluates to the polynomial in two variables.

2.4 Polynomials and Polynomial Matrices

Polynomials are easily created and manipulated in Scilab. Manipulation of polynomial matrices is essentially identical to that of constant matrices. The `poly` primitive in Scilab can be used to specify the coefficients of a polynomial or the roots of a polynomial.

```
-->p=poly([1 2],'s')   //polynomial defined by its roots
 p         =

              2
    2 - 3s + s

-->q=poly([1 2],'s','c')  //polynomial defined by its coefficients
 q         =

    1 + 2s

-->p+q
 ans       =

            2
```

```
    3 - s + s

-->p*q
 ans       =

              2    3
    2 + s - 5s + 2s

--> q/p
 ans       =

      1 + 2s
    -----------
              2
    2 - 3s + s
```

Note that the polynomial `p` has the *roots* 1 and 2 whereas the polynomial `q` has the *coefficients* 1 and 2. It is the third argument in the `poly` primitive which specifies the coefficient flag option. In the case where the first argument of `poly` is a square matrix and the roots option is in effect the result is the characteristic polynomial of the matrix.

```
--> poly([1 2;3 4],'s')
 ans       =

              2
  - 2 - 5s + s
```

Polynomials can be added, subtracted, multiplied, and divided, as usual, but only between polynomials of same formal variable.

Polynomials, like real and complex constants, can be used as elements in matrices. This is a very useful feature of Scilab for systems theory.

```
-->s=poly(0,'s');

-->A=[1 s;s 1+s^2]
 A         =

!   1     s      !
!                !
!               2 !
!   s     1 + s  !

--> B=[1/s 1/(1+s);1/(1+s) 1/s^2]
 B         =

!   1           1    !
! ------      ------ !
```

```
!   s         1 + s  !
!                    !
!     1       1      !
!    ---     ---     !
!              2     !
!   1 + s     s      !
```

From the above examples it can be seen that matrices can be constructed from polynomials and rationals.

2.4.1 Rational polynomial simplification

Scilab automatically performs pole-zero simplifications when the the built-in primitive `simp` finds a common factor in the numerator and denominator of a rational polynomial `num/den`. Pole-zero simplification is a difficult problem from a numerical viewpoint and `simp` function is usually conservative. When making calculations with polynomials, it is sometimes desirable to avoid pole-zero simplifications : this is possible by switching Scilab into a "no-simplify" mode : `help simp_mode`. The function `trfmod` can also be used for simplifying specific pole-zero pairs.

2.5 Boolean Matrices

Boolean constants are `%t` and `%f`. They can be used in boolean matrices. The syntax is the same as for ordinary matrices i.e. they can be concatenated, transposed, etc...

Operations symbols used with boolean matrices or used to create boolean matrices are `==` and `~`.

If `B` is a matrix of booleans `or(B)` and `and(B)` perform the logical `or` and `and`.

```
-->%t
 %t  =

  T

-->[1,2]==[1,3]
 ans  =

! T F !

-->[1,2]==1
 ans  =

! T F !

-->a=1:5; a(a>2)
 ans  =

!   3.    4.    5. !
```

```
-->A=[%t,%f,%t,%f,%f,%f];

-->B=[%t,%f,%t,%f,%t,%t]
 B  =

! T F T F T T !

-->A|B
 ans  =

! T F T F T T !

-->A&B
 ans  =

! T F T F F F !
```

Sparse boolean matrices are generated when, e.g., two constant sparse matrices are compared. These matrices are handled as ordinary boolean matrices.

2.6 Integer Matrices

There are 6 integer data types defined in Scilab, all these types have the same major type (see the `type` function) which is 8 and different sub-types (see the `inttype` function)

- 32 bit signed integers (sub-type 4)
- 32 bit unsigned integers (sub-type 14)
- 16 bit signed integers (sub-type 2)
- 16 bit unsigned integers (sub-type 23)
- 8 bit signed integers (sub-type 2)
- 8 bit unsigned integers (sub-type 12)

It is possible to build these integer data types from standard matrix (see 2.2) using the `int32`, `uint32`, `int16`, `uint16`, `int8`, `uint8` conversion functions

```
-->x=[0 3.2 27 135] ;

-->int32(x)
 ans  =

!0   3  27  135 !

-->int8(x)
 ans  =

!0   3  27  -121!
-->uint8(x)
 ans  =
```

```
!0   3  27  135 !
```

The same function can also convert from one sub-type to another one. The `double` function transform any of the integer type in a standard type :

```
-->y=int32([2 5 285])
 y  =

!2  5  285 !

-->uint8(y)
 ans  =

!2  5  29 !

-->double(ans)
 ans  =

!   2.    5.    29. !
```

Arithmetic and comparison operations can be applied to this type

```
-->x=int16([1 5 12])
 x  =

!1  5  12 !

-->x([1 3])
 ans  =

!1  12 !

-->x+x

 ans  =

!2  10  24 !

-->x*x'
 ans  =

 170
-->y=int16([1 7 11])
 y  =

!1  7  11 !
```

```
-->x>y
 ans  =

! F F T !
```

The operators `&`, `|` and used with these datatypes correspond to AND, OR and NOT bit-wise operations.

```
-->x=int16([1 5 12])
 x  =

!1  5  12 !

-->x|int16(2)
 ans  =

!3  7  14 !

-->int16(14)&int16(2)
 ans  =

 2
-->~uint8(2)
 ans  =

 253
```

2.7 Lists

Scilab has a list data type. The list is a collection of data objects not necessarily of the same type. A list can contain any of the already discussed data types (including functions) as well as other lists. Lists are useful for defining structured data objects.

There are two kinds of lists, ordinary lists and typed-lists. A list is defined by the `list` function. Here is a simple example :

```
-->L=list(1,'w',ones(2,2))  //L is a list made of 3 entries
 L  =


       L(1)

    1.

       L(2)

 w
```

```
       L(3)

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

-->L(3)   //extracting entry 3 of list L
 ans  =

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

-->L(3)(2,2)  //entry 2,2 of matrix L(3)
 ans  =

    1.

-->L(2)=list('w',rand(2,2)) //nested list: L(2) is now a list
 L  =


       L(1)

    1.

       L(2)


        L(2)(1)

 w

        L(2)(2)

!   0.6653811    0.8497452 !
!   0.6283918    0.6857310 !

       L(3)

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

-->L(2)(2)(1,2)  //extracting entry 1,2 of entry 2 of L(2)
 ans  =

    0.8497452

-->L(2)(2)(1,2)=5; //assigning a new value to this entry.
```

Typed lists have a specific first entry. This first entry must be a character string (the type) or a vector of character string (the first component is then the type, and the following elements the names given to the entries of the list). Typed lists entries can be manipulated by using character strings (the names) as shown below.

```
-->L=tlist(['Car';'Name';'Dimensions'],'Nevada',[2,3])
 L  =


       L(1)

!Car         !
!            !
!Name        !
!            !
!Dimensions  !

       L(2)

 Nevada

       L(3)

!   2.    3. !

-->L.Name    //same as L(2)
 ans  =

 Nevada

-->L.Dimensions(1,2)=2.3

 L  =


       L(1)

!Car         !
!            !
!Name        !
!            !
!Dimensions  !

       L(2)

 Nevada
```

```
       L(3)

!   2.    2.3 !


-->L(3)(1,2)
 ans  =

    2.3

-->L(1)(1)
 ans  =

 Car
```

An important feature of typed-lists is that it is possible to define operators acting on them (overloading), i.e., it is possible to define e.g. the multiplication `L1*L2` of the two typed lists `L1` and `L2`. An example of use is given below, where linear systems manipulations (concatenation, addition, multiplication,...) are done by such operations.

2.8 N-dimensionnal arrays

N-dimensionnal array can be defined and handled in simple way :

```
-->M(2,2,2)=3
 M  =

(:,:,1)

!   0.    0. !
!   0.    0. !
(:,:,2)

!   0.    0. !
!   0.    3. !

-->M(:,:,1)=rand(2,2)
 M  =

(:,:,1)

!   0.9329616    0.312642  !
!   0.2146008    0.3616361 !
(:,:,2)

!   0.    0. !
!   0.    3. !
```

```
-->M(2,2,:)
 ans  =

(:,:,1)

    0.3616361
(:,:,2)

    3.
-->size(M)
 ans  =

!   2.    2.    2. !

-->size(M,3)
 ans  =

    2.
```

They can be created from a vector of data and a vector of dimension

```
-->hypermat([2 3,2],1:12)
 ans  =

(:,:,1)

!   1.    3.    5. !
!   2.    4.    6. !
(:,:,2)

!   7.    9.     11. !
!   8.    10.    12. !
```

N-dimensionnal matrices are coded as `mlists` with 2 fields :

```
-->M=hypermat([2 3,2],1:12);
-->M.dims
 ans  =

!   2.    3.    2. !
-->M.entries
 ans  =

!   1.  !
!   2.  !
```

```
!   3.  !
!   4.  !
!   5.  !
!   6.  !
!   7.  !
!   8.  !
!   9.  !
!   10. !
!   11. !
!   12. !
```

2.9 Linear system representation

Linear systems are treated as specific typed lists `tlist`. The basic function which is used for defining linear systems is `syslin`. This function receives as parameters the constant matrices which define a linear system in state-space form or, in the case of system in transfer form, its input must be a rational matrix. To be more specific, the calling sequence of `syslin` is either `Sl=syslin('dom',A,B,C,D,x0)` or `Sl=syslin('dom',trmat)`. `dom` is one of the character strings `'c'` or `'d'` for continuous time or discrete time systems respectively. It is useful to note that `D` can be a polynomial matrix (improper systems) ; `D` and `x0` are optional arguments. `trmat` is a rational matrix i.e. it is defined as a matrix of rationals (ratios of polynomials). `syslin` just converts its arguments (e.g. the four matrices A,B,C,D) into a typed list `Sl`. For state space representation `Sl` is the `tlist(['lss','A','B','C','D'],A,B,C,D,'dom')`. This tlist representation allows to access the A-matrix i.e. the second entry of `Sl` by the syntax `Sl('A')` (equivalent to `Sl(2)`). Conversion from a representation to another is done by `ss2tf` or `tf2ss`. Improper systems are also treated. `syslin` defines linear systems as specific `tlist`. (`help syslin`).

```
-->//list defining a linear system

-->A=[0 -1;1 -3];B=[0;1];C=[-1 0];

-->Sys=syslin('c',A,B,C)
 Sys  =


       Sys(1)   (state-space system:)

!lss  A  B  C  D  X0  dt  !

       Sys(2) = A matrix =

!   0.  - 1. !
!   1.  - 3. !

       Sys(3) = B matrix =
```

```
!   0. !
!   1. !

       Sys(4) = C matrix =

! - 1.    0. !

       Sys(5) = D matrix =

    0.

       Sys(6) = X0 (initial state) =

!   0. !
!   0. !

       Sys(7) = Time domain =

 c

-->//conversion from state-space form to transfer form

-->Sys.A  //The A-matrix
 ans  =

!   0.  - 1. !
!   1.  - 3. !

-->Sys.B
 ans  =

!   0. !
!   1. !

-->hs=ss2tf(Sys)
 hs  =

        1
    ---------
              2
    1 + 3s + s

-->size(hs)
 ans  =

!   1.    1. !
```

```
-->hs.num
 ans  =

    1

-->hs.den
 ans  =

              2
    1 + 3s + s

-->typeof(hs)
 ans  =

 rational

-->//inversion of transfer matrix

-->inv(hs)
 ans  =

              2
    1 + 3s + s
    ----------
        1

-->//inversion of state-space form

-->inv(Sys)
 ans  =


       ans(1)   (state-space system:)

!lss  A  B  C  D  X0  dt  !

       ans(2) = A matrix =

     []

       ans(3) = B matrix =

     []

       ans(4) = C matrix =

     []
```

```
       ans(5) = D matrix =

                2
    1 + 3s + s

       ans(6) = X0 (initial state) =

     []

       ans(7) = Time domain =

 c

-->//converting this inverse to transfer representation

-->ss2tf(ans)
 ans  =

                2
    1 + 3s + s
```

The list representation allows manipulating linear systems as abstract data objects. For example, the linear system can be combined with other linear systems or the transfer function representation of the linear system can be obtained as was done above using `ss2tf`. Note that the transfer function representation of the linear system is itself a tlist. A very useful aspect of the manipulation of systems is that a system can be handled as a data object. Linear systems can be inter-connected, their representation can easily be changed from state-space to transfer function and vice versa.

The inter-connection of linear systems can be made as illustrated in Figure 2.1. For each of the possible inter-connections of two systems `S1` and `S2` the command which makes the inter-connection is shown on the right side of the corresponding block diagram in Figure 2.1. Note that feedback interconnection is performed by `S1/.S2`.

The representation of linear systems can be in state-space form or in transfer function form. These two representations can be interchanged by using the functions `tf2ss` and `ss2tf` which change the representations of systems from transfer function to state-space and from state-space to transfer function, respectively. An example of the creation, the change in representation, and the inter-connection of linear systems is demonstrated in the following Scilab session.

```
-->//system connecting

-->s=poly(0,'s');

-->S1=1/(s-1)
 S1  =

      1
```

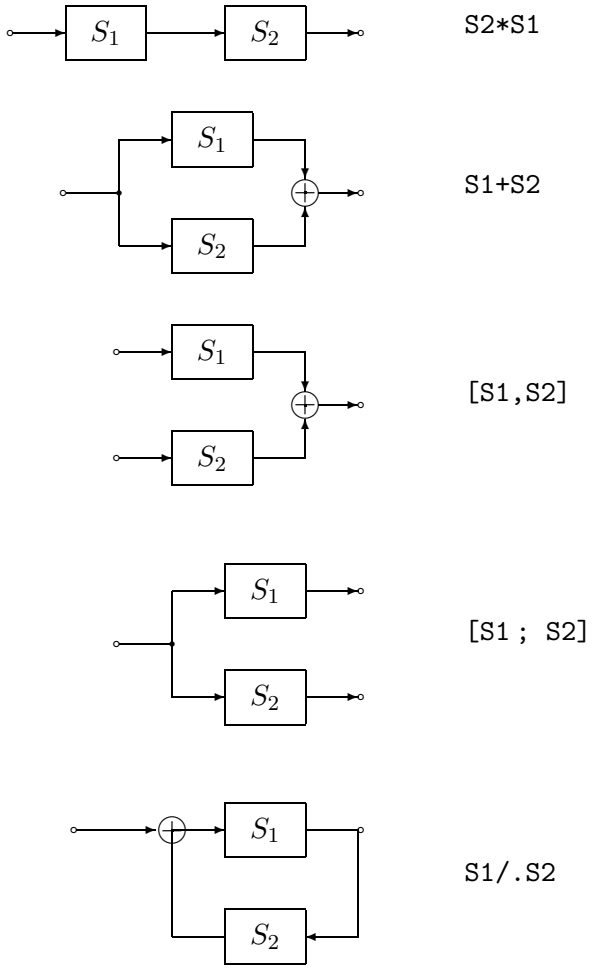


FIG. 2.1 – Inter-Connection of Linear Systems

```
    -----
  - 1 + s

-->S2=1/(s-2)
 S2  =

      1
    -----
  - 2 + s

-->S1=syslin('c',S1);

-->S2=syslin('c',S2);

-->Gls=tf2ss(S2);

-->ssprint(Gls)

.
x = | 2 |x + | 1 |u

y = | 1 |x

-->hls=Gls*S1;

-->ssprint(hls)

.   | 2  1 |    | 0 |
x = | 0  1 |x + | 1 |u

y = | 1  0 |x

-->ht=ss2tf(hls)
 ht  =

        1
    ---------
              2
    2 - 3s + s

-->S2*S1
 ans  =

        1
    ---------
              2
    2 - 3s + s
```

```
-->S1+S2
 ans  =

    - 3 + 2s
    ----------
              2
    2 - 3s + s

-->[S1,S2]
 ans  =

!     1         1    !
!   -----     -----  !
! - 1 + s   - 2 + s  !

-->[S1;S2]
 ans  =

!     1    !
!   -----  !
! - 1 + s  !
!          !
!     1    !
!   -----  !
! - 2 + s  !

-->S1/.S2
 ans  =

    - 2 + s
    ---------
              2
    3 - 3s + s

-->S1./(2*S2)
 ans  =

  - 2 + s
    -----
  - 2 + 2s
```

The above session is a bit long but illustrates some very important aspects of the handling of linear systems. First, two linear systems are created in transfer function form using the function called `syslin`. This function was used to label the systems in this example as being continuous (as opposed to discrete). The primitive `tf2ss` is used to convert one of the two transfer functions to its equivalent state-space representation which is in list form (note that the function `ssprint` creates

a more readable format for the state-space linear system). The following multiplication of the two systems yields their series inter-connection. Notice that the inter-connection of the two systems is effected even though one of the systems is in state-space form and the other is in transfer function form. The resulting inter-connection is given in state-space form. Finally, the function `ss2tf` is used to convert the resulting inter-connected systems to the equivalent transfer function representation.

2.10 Functions (Macros)

Functions are collections of commands which are executed in a new environment thus isolating function variables from the original environments variables. Functions can be created and executed in a number of different ways. Furthermore, functions can pass arguments, have programming features such as conditionals and loops, and can be recursively called. Functions can be arguments to other functions and can be elements in lists. The most useful way of creating functions is by using a text editor, however, functions can be created directly in the Scilab environment using the syntax `function` or the `deff` primitive.

```
--> function [x]=foo(y)
-->    if y>0 then, x=1; else, x=-1; end
--> endfunction

--> deff('[x]=foo(y)','if y>0 then, x=1; else, x=-1; end')

--> foo(5)
 ans       =

    1.

--> foo(-3)
 ans       =

  - 1.
```

Usually functions are defined in a file using an editor and loaded into Scilab with :
`exec('filename')`.
This can be done also by clicking in the `File operation` button. This latter syntax loads the function(s) in `filename` and compiles them. The first line of `filename` must be as follows :

```
function [y1,...,yn]=macname(x1,...,xk)
```

where the `yi`'s are output variables and the `xi`'s the input variables.

For more on the use and creation of functions see Section 3.2.

2.11 Libraries

Libraries are collections of functions which can be either automatically loaded into the Scilab environment when Scilab is called, or loaded when desired by the user. Libraries are created by the `lib` command. Examples of librairies are given in the `SCIDIR/macros` directory. Note that in these directory there is an ASCII file "names" which contains the names of each function of the library, a

set of `.sci` files which contains the source code of the functions and a set of `.bin` files which contains the compiled code of the functions. The Makefile invokes `scilab` for compiling the functions and generating the `.bin` files. The compiled functions of a library are automatically loaded into Scilab at their first call. To build a library the command `genlib` can be used (`help genlib`).

2.12 Objects

We conclude this chapter by noting that the function `typeof` returns the type of the various Scilab objects. The following objects are defined :

- `usual` for matrices with real or complex entries.
- `polynomial` for polynomial matrices : coefficients can be real or complex.
- `boolean` for boolean matrices.
- `character` for matrices of character strings.
- `function` for functions.
- `rational` for rational matrices (`syslin` lists)
- `state-space` for linear systems in state-space form (`syslin` lists).
- `sparse` for sparse constant matrices (real or complex)
- `boolean sparse` for sparse boolean matrices.
- `list` for ordinary lists.
- `tlist` for typed lists.
- `mlist` for matrix oriented typed lists.
- `state-space (or rational)` for syslin lists.
- `library` for library definition.

2.13 Matrix Operations

The following table gives the syntax of the basic matrix operations available 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

2.14 Indexing

The following sample sessions shows the flexibility which is offered for extracting and inserting entries in matrices or lists. For additional details enter `help extraction` or `help insertion`.

2.14.1 Indexing in matrices

Indexing in matrices can be done by giving the indices of selected rows and columns or by boolean indices or by using the `$` symbol.

```
-->A=[1 2 3;4 5 6]
 A  =

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

-->A(1,2)
 ans  =

    2.

-->A([1 1],2)
 ans  =

!   2. !
!   2. !
```

```
-->A(:,1)
 ans  =

!   1. !
!   4. !

-->A(:,3:-1:1)
 ans  =

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

-->A(1)
 ans  =

    1.

-->A(6)
 ans  =

    6.

-->A(:)
 ans  =

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

-->A([%t %f %f %t])
 ans  =

!   1. !
!   5. !

-->A([%t %f],[2 3])
 ans  =

!   2.    3. !

-->A(1:2,$-1)
 ans  =
```

```
!   2. !
!   5. !

-->A($:-1:1,2)
 ans  =

!   5. !
!   2. !

-->A($)
 ans  =

    6.

-->//

-->x='test'
 x  =

 test

-->x([1 1;1 1;1 1])
 ans  =

!test  test  !
!            !
!test  test  !
!            !
!test  test  !

-->//

-->B=[1/%s,(%s+1)/(%s-1)]
 B  =

!   1     1 + s  !
!   -     -----  !
!   s   - 1 + s  !

-->B(1,1)
 ans  =

    1
    -
    s

-->B(1,$)
```

```
 ans  =

    1 + s
    -----
  - 1 + s

-->B(2) // the numerator
 ans  =

!   1     1 + s  !

-->//

-->A=[1 2 3;4 5 6]
 A  =

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

-->A(1,2)=10
 A  =

!   1.    10.    3. !
!   4.    5.     6. !

-->A([1 1],2)=[-1;-2]
 A  =

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

-->A(:,1)=[8;5]
 A  =

!   8.  - 2.    3. !
!   5.    5.    6. !

-->A(1,3:-1:1)=[77 44 99]
 A  =

!   99.    44.    77. !
!   5.     5.     6.  !

-->A(1,:)=10
 A  =

!   10.    10.    10. !
```

```
!   5.     5.     6.  !

-->A(1)=%s
 A  =

!   s     10     10  !
!                    !
!   5     5      6   !

-->A(6)=%s+1
 A  =

!   s     10     10      !
!                        !
!   5     5      1 + s   !

-->A(:)=1:6
 A  =

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

-->A([%t %f],1)=33
 A  =

!   33.    3.    5. !
!   2.     4.    6. !

-->A(1:2,$-1)=[2;4]
 A  =

!   33.    2.    5. !
!   2.     4.    6. !

-->A($:-1:1,1)=[8;7]
 A  =

!   7.    2.    5. !
!   8.    4.    6. !

-->A($)=123
 A  =

!   7.    2.    5.   !
!   8.    4.    123. !

-->//
```

```
-->x='test'
 x  =

 test

-->x([4 5])=['4','5']
 x  =

!test        4  5  !
```

2.14.2 Indexing in lists

The following session illustrates how to create lists and insert/extract entries in `list` and `tlist` or `mlist`. Enter `help insertion` and `help extraction` for additinal examples.

```
-->a=33;b=11;c=0;

-->l=list();l(0)=a
 l  =


       l(1)

    33.

-->l=list();l(1)=a
 l  =


       l(1)

    33.

-->l=list(a);l(2)=b
 l  =


       l(1)

    33.

       l(2)

    11.
```

```
-->l=list(a);l(0)=b
 l  =


       l(1)

    11.

       l(2)

    33.

-->l=list(a);l(1)=c
 l  =


       l(1)

    0.

-->l=list();l(0)=null()
 l  =

     ()

-->l=list();l(1)=null()
 l  =

     ()

-->//

-->i='i';

-->l=list(a,list(c,b),i);l(1)=null()
 l  =


       l(1)


        l(1)(1)

    0.

        l(1)(2)
```

```
    11.

       l(2)

 i

-->l=list(a,list(c,list(a,c,b),b),'h');

-->l(2)(2)(3)=null()
 l  =


       l(1)

    33.

       l(2)


        l(2)(1)

    0.

        l(2)(2)


         l(2)(2)(1)

    33.

         l(2)(2)(2)

    0.

        l(2)(3)

    11.

       l(3)

 h

-->//

-->dts=list(1,tlist(['x';'a';'b'],10,[2 3]));

-->dts(2).a
```

```
 ans  =

    10.

-->dts(2).b(1,2)
 ans  =

    3.

-->[a,b]=dts(2)(['a','b'])
 b  =

!   2.    3. !
 a  =

    10.

-->//

-->l=list(1,'qwerw',%s)
 l  =


       l(1)

    1.

       l(2)

 qwerw

       l(3)

    s

-->l(1)='Changed'
 l  =


       l(1)

 Changed

       l(2)

 qwerw
```

```
      l(3)

   s

-->l(0)='Added'
 l  =


      l(1)

 Added

      l(2)

 Changed

      l(3)

 qwerw

      l(4)

   s

-->l(6)=['one more';'added']
 l  =


      l(1)

 Added

      l(2)

 Changed

      l(3)

 qwerw

      l(4)

   s

      l(5)

   Undefined
```

```
       l(6)

!one more  !
!          !
!added     !

-->//

-->dts=list(1,tlist(['x';'a';'b'],10,[2 3]));

-->dts(2).a=33
 dts  =


       dts(1)

    1.

       dts(2)


        dts(2)(1)

!x  !
!   !
!a  !
!   !
!b  !

        dts(2)(2)

    33.

        dts(2)(3)

!   2.    3. !

-->dts(2).b(1,2)=-100
 dts  =


       dts(1)

    1.

       dts(2)
```

```
        dts(2)(1)

!x  !
!   !
!a  !
!   !
!b  !

        dts(2)(2)

    33.

        dts(2)(3)

!   2.  - 100. !

-->//

-->l=list(1,'qwerw',%s);

-->l(1)
 ans  =

    1.

-->[a,b]=l([3 2])
 b  =

 qwerw
 a  =

    s

-->l($)
 ans  =

    s

-->//

-->L=list(33,list(l,33))
 L  =


       L(1)
```

33.

L(2)

L(2)(1)

L(2)(1)(1)

1.

L(2)(1)(2)

qwerw

L(2)(1)(3)

s

L(2)(2)

33.

Chapitre 3

Programming

One of the most useful features of Scilab is its ability to create and use functions. This allows the development of specialized programs which can be integrated into the Scilab package in a simple and modular way through, for example, the use of libraries. In this chapter we treat the following subjects :

- Programming Tools
- Defining and Using Functions
- Definition of Operators for New Data Types
- Debbuging

Creation of libraries is discussed in a later chapter.

3.1 Programming Tools

Scilab supports a full list of programming tools including loops, conditionals, case selection, and creation of new environments. Most programming tasks should be accomplished in the environment of a function. Here we explain what programming tools are available.

3.1.1 Comparison Operators

There exist five methods for making comparisons between the values of data objects in Scilab. These comparisons are listed in the following table.

==	equal to
<	smaller than
>	greater than
<=	smaller or equal to
>=	greater or equal to
<> or ~=	not equal to

These comparison operators are used for evaluation of conditionals.

3.1.2 Loops

Two types of loops exist in Scilab : the `for` loop and the `while` loop. The `for` loop steps through a vector of indices performing each time the commands delimited by `end`.

```
--> x=1;for k=1:4,x=x*k,end
 x         =

    1.
 x         =

    2.
 x         =

    6.
 x         =

    24.
```

The `for` loop can iterate on any vector or matrix taking for values the elements of the vector or the columns of the matrix.

```
--> x=1;for k=[-1 3 0],x=x+k,end
 x         =

    0.
 x         =

    3.
 x         =

    3.
```

The `for` loop can also iterate on lists. The syntax is the same as for matrices. The index takes as values the entries of the list.

```
-->l=list(1,[1,2;3,4],'str')

-->for k=l, disp(k),end

    1.

!   1.    2. !
!   3.    4. !

 str
```

The `while` loop repeatedly performs a sequence of commands until a condition is satisfied.

```
--> x=1; while x<14,x=2*x,end
 x         =
```

```
    2.
 x         =

    4.
 x         =

    8.
 x         =

    16.
```

A `for` or `while` loop can be ended by the command `break` :

```
-->a=0;for i=1:5:100,a=a+1;if i > 10  then  break,end; end

-->a
 a  =

    3.
```

In nested loops, `break` exits from the innermost loop.

```
-->for k=1:3; for j=1:4; if k+j>4 then break;else disp(k);end;end;end

    1.

    1.

    1.

    2.

    2.

    3.
```

3.1.3 Conditionals

Two types of conditionals exist in Scilab : the `if-then-else` conditional and the `select-case` conditional. The `if-then-else` conditional evaluates an expression and if true executes the instructions between the `then` statement and the `else` statement (or `end` statement). If false the statements between the `else` and the `end` statement are executed. The `else` is not required. The `elseif` has the usual meaning and is a also a keyword recognized by the interpreter.

```
--> x=1
```

```
 x         =

    1.

--> if x>0 then,y=-x,else,y=x,end
 y         =

  - 1.

--> x=-1
 x         =

  - 1.

--> if x>0 then,y=-x,else,y=x,end
 y         =

  - 1.
```

The `select-case` conditional compares an expression to several possible expressions and performs the instructions following the first case which equals the initial expression.

```
--> x=-1
 x         =

  - 1.

--> select x,case 1,y=x+5,case -1,y=sqrt(x),end
 y         =

    i
```

It is possible to include an `else` statement for the condition where none of the cases are satisfied.

3.2 Defining and Using Functions

It is possible to define a function directly in the Scilab environment, however, the most convenient way is to create a file containing the function with a text editor. In this section we describe the structure of a function and several Scilab commands which are used almost exclusively in a function environment.

3.2.1 Function Structure

Function structure must obey the following format

```
function [y1,...,yn]=foo(x1,...,xm)
   .
   .
   .
```

where `foo` is the function name, the `xi` are the m input arguments of the function, the `yj` are the n output arguments from the function, and the three vertical dots represent the list of instructions performed by the function. An example of a function which calculates $k!$ is as follows

```
function [x]=fact(k)
  k=int(k)
  if k<1 then k=1,end
  x=1;
  for j=1:k,x=x*j;end
endfunction
```

If this function is contained in a file called `fact.sci` the function must be "loaded" into Scilab by the `exec` or `getf` command and before it can be used :

```
--> exists('fact')
 ans       =

    0.

--> exec('../macros/fact.sci',-1);

--> exists('fact')
 ans       =

    1.

--> x=fact(5)
 x         =

    120.
```

In the above Scilab session, the command `exists` indicates that `fact` is not in the environment (by the 0 answer to `exist`). The function is loaded into the environment using `exec` and now `exists` indicates that the function is there (the 1 answer). The example calculates 5!.

3.2.2 Loading Functions

Functions are usually defined in files. A file which contains a function must obey the following format

```
function [y1,...,yn]=foo(x1,...,xm)
   .
   .
   .
```

where `foo` is the function name. The `xi`'s are the input parameters and the the `yj`'s are the output parameters, and the three vertical dots represent the set of instructions performed by the function to evaluate the `yj`'s, given the `xi`'s. Inputs and ouputs parameters can be *any* Scilab object (including functions themeselves).

Functions are Scilab objects and should not be considered as files. To be used in Scilab, functions defined in files *must* be loaded by the command `getf(filename)` or `exec(filename,-1)` ; . If the file `filename` contains the function `foo`, the function `foo` can be executed only if it has been previously loaded by the command `getf(filename)`. A file may contain *several* functions. Functions can also be defined "on line" by the command using the `function/endfunction` syntax or by using the function `deff`. This is useful if one wants to define a function as the output parameter of a other function.

Collections of functions can be organized as libraries (see `lib` command). Standard Scilab librairies (linear algebra, control,...) are defined in the subdirectories of `SCIDIR/macros/`.

3.2.3 Global and Local Variables

If a variable in a function is not defined (and is not among the input parameters) then it takes the value of a variable having the same name in the calling environment. This variable however remains local in the sense that modifying it within the function does not alter the variable in the calling environment unless `resume` is used (see below). Functions can be invoked with less input or output parameters. Here is an example :

```
function [y1,y2]=f(x1,x2)
  y1=x1+x2
  y2=x1-x2
endfunction

-->[y1,y2]=f(1,1)
 y2  =
    0.
 y1  =
    2.

-->f(1,1)
 ans  =
    2.

-->f(1)
y1=x1+x2;
        !--error     4
undefined variable : x2
at line       2 of function f

-->x2=1;

-->[y1,y2]=f(1)
 y2  =
    0.
 y1  =
    2.

-->f(1)
```

```
 ans  =

    2.
```

Note that it is not possible to call a function if one of the parameter of the calling sequence is not defined :

```
function [y]=f(x1,x2)
  if x1<0 then y=x1, else y=x2;end
endfunction


-->f(-1)
 ans  =

  - 1.

-->f(-1,x2)
       !--error     4
undefined variable : x2

-->f(1)
 undefined variable : x2
at line       2 of function    f     called by :
f(1)

-->x2=3;f(1)

-->f(1)
 ans  =

    3
```

Global variable are defined by the `global` command. They can be read and modified inside functions. Enter `help global` for details.

3.2.4 Special Function Commands

Scilab has several special commands which are used almost exclusively in functions. These are the commands

- `argn` : returns the number of input and output arguments for the function
- `error` : used to suspend the operation of a function, to print an error message, and to return to the previous level of environment when an error is detected.
- `warning`,
- `pause` : temporarily suspends the operation of a function.
- `break` : forces the end of a loop

– `return` or `resume` : used to return to the calling environment and to pass local variables from the function environment to the calling environment.

The following example runs the following `foo` function which illustrates these commands.

– The function definition

```
function [z]=foo(x,y)
[out,in]=argn(0);
if x==0 then,
     error('division by zero');
end,
slope=y/x;
pause,
z=sqrt(slope);
s=resume(slope);
endfunction
```

– The function use

```
--> z=foo(0,1)
error('division by zero');
                            !--error 10000
division by zero
at line       4 of function    foo      called by :
 z=foo(0,1)


--> z=foo(2,1)

-1-> resume
 z         =

    0.7071068

--> s
 s         =

    0.5
```

In the example, the first call to `foo` passes an argument which cannot be used in the calculation of the function. The function discontinues operation and indicates the nature of the error to the user. The second call to the function suspends operation after the calculation of `slope`. Here the user can examine values calculated inside of the function, perform plots, and, in fact perform any operations allowed in Scilab. The `-1->` prompt indicates that the current environment created by the `pause` command is the environment of the function and not that of the calling environment. Control is returned to the function by the command `return`. Operation of the function can be stopped by the command `quit` or `abort`. Finally the function terminates its calculation returning the value of `z`. Also available in the environment is the variable `s` which is a local variable from the function which is passed to the global environment.

3.3 Definition of Operations on New Data Types

It is possible to transparently define fundamental operations for new data types in Scilab (enter `help overloading` for a full description of this feature). That is, the user can give a sense to multiplication, division, addition, etc. on any two data types which exist in Scilab. As an example, two linear systems (represented by lists) can be added together to represent their parallel inter-connection or can be multiplied together to represent their series inter-connection. Scilab performs these user defined operations by searching for functions (written by the user) which follow a special naming convention described below.

The naming convention Scilab uses to recognize operators defined by the user is determined by the following conventions. The name of the user defined function is composed of four (or possibly three) fields. The first field is always the symbol `%`. The third field is one of the characters in the following table which represents the type of operation to be performed between the two data types.

Third field	
SYMBOL	OPERATION
a	+
b	: (range generator)
c	[a,b] column concatenation
d	./
e	() extraction : m=a(k)
f	[a;b] row concatenation
g	| logical or
h	& logical and
i	() insertion : a(k)=m
j	.^ element wise exponent
k	.*.
l	\ left division
m	*
n	<> inequality comparison
o	== equality comparison
p	^ exponent
q	.\
r	/ right division
s	-
t	' (transpose)
u	*.
v	/.
w	\.
x	.*
y	./.
z	.\.
0	.'
1	<
2	>
3	<=
4	>=
5	~ (not)

The second and fourth fields represent the type of the first and second data objects, respectively, to be treated by the function and are represented by the symbols given in the following table.

Second and Fourth fields	
SYMBOL	VARIABLE TYPE
s	scalar
p	polynomial
l	list (untyped)
c	character string
b	boolean
sp	sparse
spb	boolean sparse
m	function
xxx	list (typed)

A typed list is one in which the first entry of the list is a character string where the first characters of the string are represented by the xxx in the above table. For example a typed list representing a linear system has the form :

```
tlist(['lss','A','B','C','D','X0','dt'],a,b,c,d,x0,'c').
```

and, thus, the xxx above is lss.

An example of the function name which multiplies two linear systems together (to represent their series inter-connection) is %lss_m_lss. Here the first field is %, the second field is lss (linear state-space), the third field is m "multiply" and the fourth one is lss. A possible user function which performs this multiplication is as follows

```
function [s]=%lss_m_lss(s1,s2)
[A1,B1,C1,D1,x1,dom1]=s1(2:7),
[A2,B2,C2,D2,x2]=s2(2:6),
B1C2=B1*C2,
s=lsslist([A1,B1C2;0*B1C2' ,A2],...
       [B1*D2;B2],[C1,D1*C2],D1*D2,[x1;x2],dom1),
endfunction
```

An example of the use of this function after having loaded it into Scilab (using for example getf or inserting it in a library) is illustrated in the following Scilab session

```
-->A1=[1 2;3 4];B1=[1;1];C1=[0 1;1 0];

-->A2=[1 -1;0 1];B2=[1 0;2 1];C2=[1 1];D2=[1,1];

-->s1=syslin('c',A1,B1,C1);

-->s2=syslin('c',A2,B2,C2,D2);

-->ssprint(s1)

.   | 1  2 |    | 1 |
x = | 3  4 |x + | 1 |u
```

```
    | 0  1 |
y = | 1  0 |x

-->ssprint(s2)

.   | 1 -1 |    | 1  0 |
x = | 0  1 |x + | 2  1 |u

y = | 1  1 |x + | 1  1 |u

-->s12=s1*s2;   //This is equivalent to s12=%lss_m_lss(s1,s2)

-->ssprint(s12)

    | 1  2  1  1 |    | 1  1 |
.   | 3  4  1  1 |    | 1  1 |
x = | 0  0  1 -1 |x + | 1  0 |u
    | 0  0  0  1 |    | 2  1 |

    | 0  1  0  0 |
y = | 1  0  0  0 |x
```

Notice that the use of `%lss_m_lss` is totally transparent in that the multiplication of the two lists `s1` and `s2` is performed using the usual multiplication operator `*`.

The directory `SCIDIR/macros/percent` contains all the functions (a very large number...) which perform operations on linear systems and transfer matrices. Conversions are automatically performed. For example the code for the function `%lss_m_lss` is there (note that it is much more complicated that the code given here!).

3.4 Debbuging

The simplest way to debug a Scilab function is to introduce a `pause` command in the function. When executed the function stops at this point and prompts `-1->` which indicates a different "level" ; another `pause` gives `-2->` ... At the level 1 the Scilab commands are analog to a different session but the user can display all the current variables present in Scilab, which are inside or outside the function i.e. local in the function or belonging to the calling environment. The execution of the function is resumed by the command `return` or `resume` (the variables used at the upper level are cleaned). The execution of the function can be interrupted by `abort`.

It is also possible to insert breakpoints in functions. See the commands `setbpt`, `delbpt`, `disbpt`. Finally, note that it is also possible to trap errors during the execution of a function : see the commands `errclear` and `errcatch`. Finally the experts in Scilab can use the function `debug(i)` where i=0,..,4 denotes a debugging level.

Chapitre 4

Basic Primitives

This chapter briefly describes some basic primitives of Scilab. More detailed information is given in the "manual" document.

4.1 The Environment and Input/Output

In this chapter we describe the most important aspects of the environment of Scilab : how to automatically perform certain operations when entering Scilab, and how to read and write data from and to the Scilab environment.

4.1.1 The Environment

Scilab is loaded with a number of variables and primitives. The command `who` lists the variables which are available. `whos()` lists the variables which are available in a more detailled fashion.

The `who` command also indicates how many elements and variables are available for use. The user can obtain on-line help on any of the functions listed by typing `help <function-name>`.

Variables can be saved in an external binary file using `save`. Similarly, variables previously saved can be reloaded into Scilab using `load`.

Note that after the command `clear x y` the variables `x` and `y` no longer exist in the environment. The command `save` without any variable arguments saves the entire Scilab environment. Similarly, the command `clear` used without any arguments clears all of the variables, functions, and libraries in the environment.

Libraries of functions are loaded using `lib`.

The list of functions available in the library can be obtained by using `disp`.

4.1.2 Startup Commands by the User

When Scilab is called the user can automatically load into the environment functions, libraries, variables, and perform commands using the the file `.scilab` in his home directory. This is particularly useful when the user wants to run Scilab programs in the background (such as in batch mode). Another useful aspect of the `.scilab` file is when some functions or libraries are often used. In this case the commands `getf exec` or `load` can be used in the `.scilab` file to automatically load the desired functions and libraries whenever Scilab is invoked.

4.1.3 Input and Output

Although the commands `save` and `load` are convenient, one has much more control over the transfer of data between files and Scilab by using the Fortran like functions `read` and `write`. These two functions work similarly to the read and write commands found in Fortran. The syntax of these two commands is as follows.

```
--> x=[1 2 %pi;%e 3 4]
 x         =

!   1.           2.    3.1415927 !
!   2.7182818    3.    4.        !

--> write('x.dat',x)

--> clear x

--> xnew=read('x.dat',2,3)
 xnew      =

!   1.           2.    3.1415927 !
!   2.7182818    3.    4.        !
```

Notice that `read` specifies the number of rows and columns of the matrix `x`. Complicated formats can be specified.

The C like function `mfscanf` and `mfprintf` can be also used

```
--> x=[1 2 %pi;%e 3 4]
 x         =

!   1.           2.    3.1415927 !
!   2.7182818    3.    4.        !

--> fd=mopen('x_c.dat','w')

--> mfprintf(fd,'%f %f %f\n',x)

--> mclose(fd)

--> clear x

--> fd=mopen('x_c.dat','r')

--> xnew(1,1:3)=mfscanf(fd,'%f %f %f\n') ;

--> xnew(2,1:3)=mfscanf(fd,'%f %f %f\n')
```

```
 xnew  =

!   1.          2.    3.141593 !
!   2.718282    3.    4.       !
--> mclose(fd)
```

4.2 Help

On-line help is available either by clicking on the `help` button or by entering `help item` (where `item` is usually the name of a function or primitive). `apropos keyword` looks for `keyword` in a `whatis` file.

To add a new item or keyword is easy. Just create a `.cat` ASCII file describing the item and a `whatis` file in your directory. Then add your directory path (and a title) in the variable `%helps` (see also the README file there). You can use the standard format of the scilab manual (see the `SCIDIR/man/subdirectories` and `SCIDIR/examples/man-examples`). The Scilab LaTeX manual is automatically obtained from the manual items by a `Makefile`. See the directory `SCIDIR/man/Latex-doc`.

4.3 Useful functions

We give here a short list of useful functions and keywords that can be used as entry points in the Scilab manual. All the functions available can be obtained by entering `help`. For each manual entry the `SEE ALSO` line refers to related functions.

- Elementary functions : `sum, prod, sqrt, diag, cos, max, round, sign, fft`
- Sorting : `sort, gsort, find`
- Specific Matrices : `zeros, eye, ones, matrix, empty`
- Linear Algebra : `det, inv, qr, svd, bdiag, spec, schur`
- Polynomials : `poly, roots, coeff, horner, clean, freq`
- Buttons, dialog : `x_choose, x_dialog, x_mdialog, getvalue, addmenu`
- Linear systems : `syslin`
- Random numbers : `rand`
- Programming : `function, deff, argn, for, if, end, while, select, warning, error, break, return`
- Comparison symbols : `==`, `>=`, `>`, `=`, `&` (`and`), `|` (`or`)
- Execution of a file : `exec`
- Debugging : `pause, return, abort`
- Spline functions, interpolation : `splin, interp, interpln`
- Character strings : `string, part, evstr, execstr`
- Graphics : `plot, xset, driver, plot2d, xgrid, locate, plot3d, Graphics`
- Ode solvers : `ode, dassl, dassrt, odedc`
- Optimization : `optim, quapro, linpro, lmitool`
- Interconnected dynamic systems : `scicos`
- Adding a C or Fortran routine : `link, fort, addinter, intersci`

4.4 Nonlinear Calculation

Scilab has several powerful non-linear primitives for simulation or optimization.

4.4.1 Nonlinear Primitives

Scilab provides several facilities for nonlinear calculations.

Numerical simulation of systems of differential equations is made by the `ode` primitive. Many solvers are available, mostly from `odepack`, for solving stiff or non-stiff systems. Implicit systems can be solved by `dassl`. It is also possible to solve systems with stopping time : integration is performed until the state is crossing a given surface. See `ode` and `dassrt` commands. There is a number of optional arguments available for solving ode's (tolerance parameters, jacobian, order of approximation, time steps etc). For ode solvers, these parameters are set by the global variable `%ODEOPTIONS`.

Minimizing non linear functions is done the `optim` function. Several algorithms (including non differentiable optimization) are available. Codes are from INRIA's `modulopt` library. Enter `help optim` for more a more detailed description.

4.4.2 Argument functions

Specific Scilab functions or C or Fortran routines can be used as an argument of some high-level primitives (such as `ode`, `optim`, `dassl`...). These fonctions are called argument functions or externals. The calling sequence of this function or routine is imposed by the high-level primitive which sets the argument of this function or routine.

For example the function `costfunc` is an argument of the `optim` primitive. Its calling sequence must be : `[f,g,ind]=costfunc(x,ind)` as imposed by the `optim` primitive. The following non-linear primitives in Scilab need argument functions or subroutines : `ode`, `optim`, `impl`, `dassl`, `intg`, `odedc`, `fsolve`. For problems where computation time is important, it is recommended to use C or Fortran subroutines. Examples of such subroutines are given in the directory `SCIDIR/routines/default`. See the README file there for more details.

When such a subroutine is written it must be linked to Scilab. This link operation can be done dynamically by the `link` command. It is also possible to introduce the code in a more permanent manner by inserting it in a specific interface in `SCIDIR/routines/default` and rebuild a new Scilab by a `make all` command in the Scilab directory.

4.5 XWindow Dialog

It may be convenient to open a specific XWindow window for entering interactively parameters inside a function or for a demo. This facility is possible thanks to e.g. the functions `x_dialog`, `x_choose`, `x_mdialog`, `x_matrix` and `x_message`. The demos which can be executed by clicking on the `demo` button provide simple examples of the use of these functions.

4.6 Tk-Tcl Dialog

An interface between Scilab and Tk-Tcl exists. A Graphic User Interface object can be created by the function `uicontrol`. Basic primitives are `TK_EvalFile`, `TK_EvalStr` and `TK_GetVar, TK_Setvar`. Examples are given by invoking the help of these functions.

Chapitre 5

Graphics

This section introduces graphics in Scilab.

5.1 The Graphics Window

It is possible to use several graphics windows `ScilabGraphicx` x being the number used for the management of the windows, but at any time only one window is active. On the main Scilab window the button `Graphic Window x` is used to manage the windows : x denotes the number of the active window, and we can set (create), raise or delete the window numbered x : in particular we can directly create the graphics window numbered 10. The execution of a plotting command automatically creates a window if necessary.

We will see later that Scilab uses a `graphics environment` defining some parameters of the plot, these parameters have default values and can be changed by the user ; every graphics window has its specific context so the same plotting command van give different results on different windows.

There are 4 buttons on the graphics window :

- `3D Rot.` : for applying a rotation with the mouse to a 3D plot. This button is inhibited for a 2D plot. For the help of manipulations (rotation with specific angles ...) the rotation angles are given at the top of the window.
- `2D Zoom` : zooming on a 2D plot. This command can be recursively invoked. For a 3D plot this button is not inhibited but it has no effect.
- `UnZoom` : return to the initial plot (not to the plot corresponding to the previous zoom in case of multiple zooms).

  These 3 buttons affecting the plot in the window are not always in use ; we will see later that there are different choices for the underlying device and zoom and rotation need the record of the plotting commands which is one of the possible choices (this is the default).
- `File` : this button opens different commands and menus.

  The first one is simple : `Clear` simply rubs out the window (without affecting the graphics context of the window).

  The command `Print...` opens a selection panel for printing. Under Unix, the printers are defined in the main scilab script `SCIDIR/bin/scilab` (obtained by "make all" from the origin file `SCIDIR/bin/scilab.g`).

  The `Export` command opens a panel selection for getting a copy of the plot on a file with a specified format (Postscript, Postscript-Latex, Xfig).

  The `save` command directly saves the plot on a file with a specified name. This file can be loaded later in Scilab for replotting.

The `Close` is the same command than the previous `Delete Graphic Window` of the menu of the main window, but simply applied to its window (the graphic context is, of course deleted).

5.2 The Media

There are different graphics devices in Scilab which can be used to send graphics to windows or paper. The default for the output is `ScilabGraphic0` window .
The different drivers are :

- `X11` : memoryless screen graphics driver
- `Rec` : a screen driver which also records all the graphic commands. This is the default (required for the zoom and rotate).
- `Wdp` : a screen driver without recorded graphics ; the graphics are done on a pixmap and are send to the graphic window with the command `xset("wshow")`. The pixmap is cleared with the command `xset("wwpc")` or with the usual command `xbasc()`
- `Pos` : graphics driver for Postscript printers
- `Fig` : graphics driver for the Xfig system
- `GIF` : graphics driver for the GIF file format

In the 3 first cases the 'implicit' device is a graphics window (existing or created by the plot). For the 2 last cases we will see later how to affect a specific device to the plot : a file where the plot will be recorded in the Postscript or Xfig format.

The basic Scilab graphics commands are :

- `driver` : selects a graphic driver

  The next 3 commands are specific of the screen drivers :
- `xclear` : clears one or more graphic windows ; does not affect the graphics context of these windows.
- `xbasc` : clears a graphic window and erase the recorded graphics ; does not affect the graphics context of the window.
- `xpause` : a pause in milliseconds
- `xselect` : raises the current graphic window (for X-drivers)
- `xclick` : waits for a mouse click
- `xbasr` : redraws the plot of a graphic window
- `xdel` : deletes a graphic window (equivalent to the `Close` button

  The following commands are specific of the Postscript, Xfig and GIF drivers drivers :
- `xinit` : initializes a graphic device (file).
- `xend` : closes a graphic session (and the associated device).

In fact, the regular driver for a common use is `Rec` and there are special commands in order to avoid a change of driver ; in many cases, one can ignore the existence of drivers and use the functions `xbasimp`, `xs2fig` in order to send a graphic to a printer or in a file for the `Xfig` system. For example with :

```
-->driver('Pos')

-->xinit('foo.ps')

-->plot(1:10)

-->xend()
```

```
-->driver('Rec')

-->plot(1:10)

-->xbasimp(0,'foo1.ps')
```

we get two identical Postscript files : `'foo.ps'` and `'foo1.ps.0'` (the appending 0 is the number of the active window where the plot has been done).

The default for plotting is the superposition ; this means that between 2 different plots one of the 2 following command is needed : `xbasc(window-number)` which clears the window and erase the recorded Scilab graphics command associated with the window `window-number` or `xclear`) which simply clears the window.

If you enlarge a graphic window, the command `xbasr(window-number)` is executed by Scilab. This command clears the graphic window `window-number` and replays the graphic commands associated with it. One can call this function manually, in order to verify the associated recorded graphics commands.

Any number of graphics windows can be created with buttons or with the commands `xset` or `xselect`. The environment variable DISPLAY can be used to specify an X11 Display or one can use the `xinit` function in order to open a graphic window on a specific display.

5.3 Global Parameters of a Plot

Graphics Context

Some parameters of the graphics are controlled by a graphic context (for example the line thickness) and others are controlled through graphics arguments of a plotting command. The graphics context has a default definition and can be change by the command `xset` : the command without argument i.e. `xset()` opens the `Scilab Toggles Panel` and the user can changes the parameters by simple mouse clickings. We give here different parameters controlled by this command :

– `xset` : set graphic context values.
(i)-`xset("font",fontid,fontsize)` : fix the current font and its current size.
(ii)-`xset("mark",markid,marksize)` : set the current mark and current mark size.
(iii)-`xset("use color",flag)` : change to color or gray plot according to the values (1 or 0) of `flag`.
(iv)-`xset("colormap",cmap)` : set the colormap as a m x 3 matrix. m is the number of colors. Color number i is given as a 3-uple cmap[i,1],cmap[i,2], cmap[i,3] corresponding respectively to Red, Green and Blue intensity between 0 and 1. Calling again `xset()` shows the colormap with the indices of the colors.
(v)-`xset("window",window-number)` : sets the current window to the window `window-number` and creates the window if it doesn't exist.
(vi)-`xset("wpos",x,y)` : fixes the position of the upper left point of the graphic window.
Many other choices are done by `xset` :
-use of a pixmap : the plot can be directly displayed on the screen or executed on a pixmap and then expose by the command `xset("wshow")` ; this is the usual way for animation effect.
-logical function for drawing : this parameter can be changed for specific effects (superposition or adding or substracting of colors). Looking at the successive plots of the following simple

commands give an example of 2 possible effects of this parameter :

```
xset('default');
plot3d();
plot3d();
xset('alufunction',7);
xset('window',0);
plot3d();
xset('default');
plot3d();
xset('alufunction',6);
xset('window',0);
plot3d();
```

We have seen that some choices exist for the fonts and this choice can be extended by the command :

- `xlfont` : to load a new family of fonts
  There exists the function "reciprocal" to `xset` :
- `xget` : to get informations about the current graphic context.
  All the values of the parameters fixed by `xset` can be obtained by `xget`. An example :

```
-->pos=xget("wpos")
 pos  =

!   105.    121. !
```

`pos` is the position of the upper left point of the graphic window.

Some Manipulations

Coordinates transforms :

- `isoview` : isometric scale without window change
  allows an isometric scale in the window of previous plots without changing the window size :

```
t=(0:0.1:2*%pi)';
plot2d(sin(t),cos(t));
xbasc()
isoview(-1,1,-1,1);
plot2d(sin(t),cos(t));
```

- `square` : isometric scale with resizing the window
  the window is resized according to the parameters of the command.
- `scaling` : scaling on data
- `rotate` : rotation
  `scaling` and `rotate` executes respectively an affine transform and a geometric rotation of a 2-lines-matrix corresponding to the `(x,y)` values of a set of points.
- `xgetech, xsetech` : change of scale inside the graphic window
  The current graphic scale can be fixed by a high level plot command. You may want to get this parameter or to fix it directly : this is the role of `xgetech, xsetech` . `xsetech` is a

simple way to cut the window in differents parts for different plots :

```
t=(0:0.1:2*%pi)';
xsetech(wrect=[0.,0.,0.6,0.3],frect=[-1,1,-1,1]);
plot2d(sin(t),cos(t));
xsetech(wrect=[0.5,0.3,0.4,0.6],frect=[-1,1,-1,1]);
plot2d(sin(t),cos(t));
```

5.4 2D Plotting

5.4.1 Basic 2D Plotting

The simplest 2D plot is `plot(x,y)` or `plot(y)` : this is the plot of `y` as function of `x` where `x` and `y` are 2 vectors ; if `x` is missing, it is replaced by the vector `(1,size(y,'*'))`. If `y` is a matrix, its rows are plotted. There are optional arguments.

A first example is given by the following commands and one of the results is represented on figure 5.1 :

```
t=(0:0.05:1)';
ct=cos(2*%pi*t);
// plot the cosine
plot(t,ct);
// xset() opens the toggle panel and
// some parameters can be changed with mouse clicks
// given by commands for the demo here
xset("font",5,4);xset("thickness",3);
// plot with captions for the axis and a title for the plot
// if a caption is empty the argument ' ' is needed
plot(t,ct,'Time','Cosine','Simple Plot');
// click on a color of the xset toggle panel and do the previous plot again
// to get the title in the chosen color
```

The generic 2D multiple plot is
`plot2di(x,y,<options>)`

- index of `plot2d` : `i=missing,2,3,4`.
  For the different values of `i` we have :
  `i=missing` : piecewise linear/logarithmic plotting
  `i=2` : piecewise constant drawing style
  `i=3` : vertical bars
  `i=4` : arrows style (e.g. ode in a phase space)

```
t=(1:0.1:8)';xset("font",2,3);
subplot(2,2,1)
plot2d([t t],[1.5+0.2*sin(t) 2+cos(t)]);
xtitle('Plot2d-Piecewise linear');
```

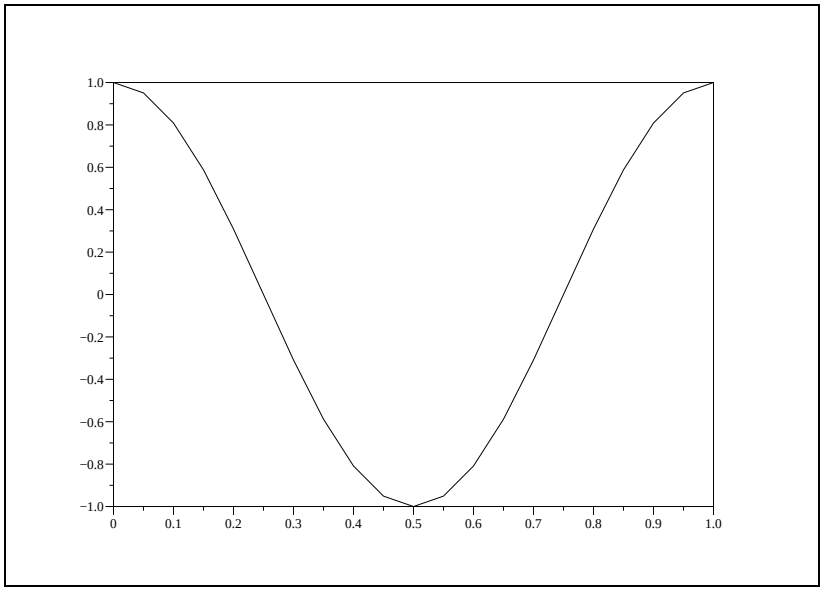


FIG. 5.1 – First example of plotting

```
//
subplot(2,2,2)
plot2d(t,[1.5+0.2*sin(t) 2+cos(t)],logflag='ll');
xtitle('Plot2d1-Logarithmic scales');
//
subplot(2,2,3)
plot2d2(t,[1.5+0.2*sin(t) 2+cos(t)]);
xtitle('Plot2d2-Piecewise constant');
//
subplot(2,2,4)
plot2d3(t,[1.5+0.2*sin(t) 2+cos(t)]);
xtitle('Plot2d3-Vertical bar plot')
```

- Parameters `x,y` : two matrices of the same size `[nl,nc]` (`nc` is the number of curves and `nl` is the number of points of each curve).
  For a single curve the vector can be row or column :
  `plot2d(t',cos(t)') plot2d(t,cos(t))` are equivalent.
- option `style` :it is a real vector of size `(1,nc)` ; the style to use for curve j is defined by `size(j)` (when only one curve is drawn `style` can specify the style and a position to use for the caption).

```
xmax=5.;x=0:0.1:xmax;
u=[-0.8+sin(x);-0.6+sin(x);-0.4+sin(x);-0.2+sin(x);sin(x)];
u=[u;0.2+sin(x);0.4+sin(x);0.6+sin(x);0.8+sin(x)]';
//start trying the symbols (negative values for the style)

plot2d(x,u,style=[-9,-8,-7,-6,-5,-4,-3,-2,-1,0])
```

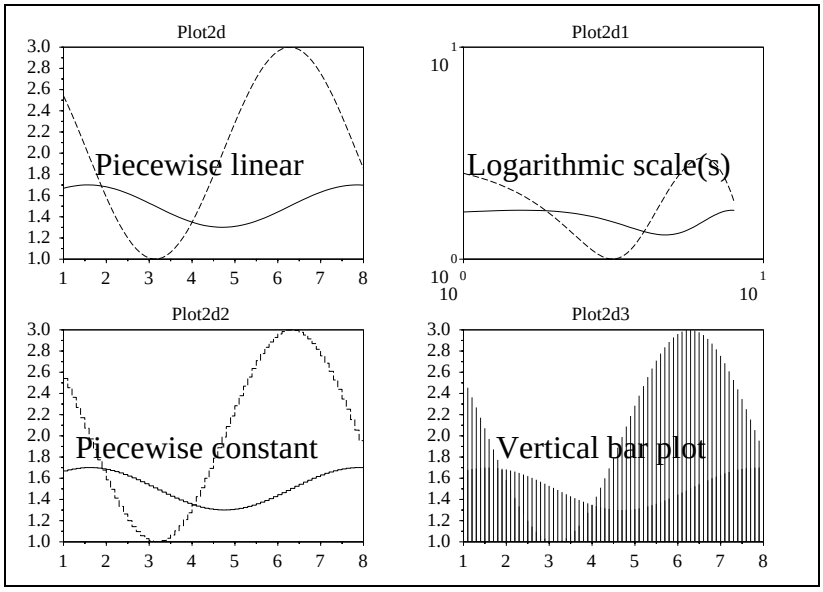


FIG. 5.2 – Different 2D plotting

```
x=0:0.2:xmax;
v=[1.4+sin(x);1.8+sin(x)]';
xset("mark size",5);
plot2d(x,v,style=[-7,-8])
xset('default');
```

– Option `frameflag` : is a scalar corresponding to :

requirements actual	ranges of a previous plot	ranges given by rect arg	ranges computed from x and y
requested one	0	1	2
Computed for isometric view		3	4
Enlarged For pretty axes		5	6
Previous and current plots merged		7	8

– Option `axesflag` controls the display of information on the frame around the plot :
  – 0 : nothing is drawn around the plot.
  – 1 : axes are drawn, the y=axis is displayed on the left.
  – 2 : the plot is surrounded by a box without tics.
  – 3 : axes are drawn, the y=axis is displayed on the right.
  – 4 : axes are drawn centred in the middle of the frame box.
  – 5 : axes are drawn so as to cross at point `(0,0)`. If point `(0,0)` does not lie inside the

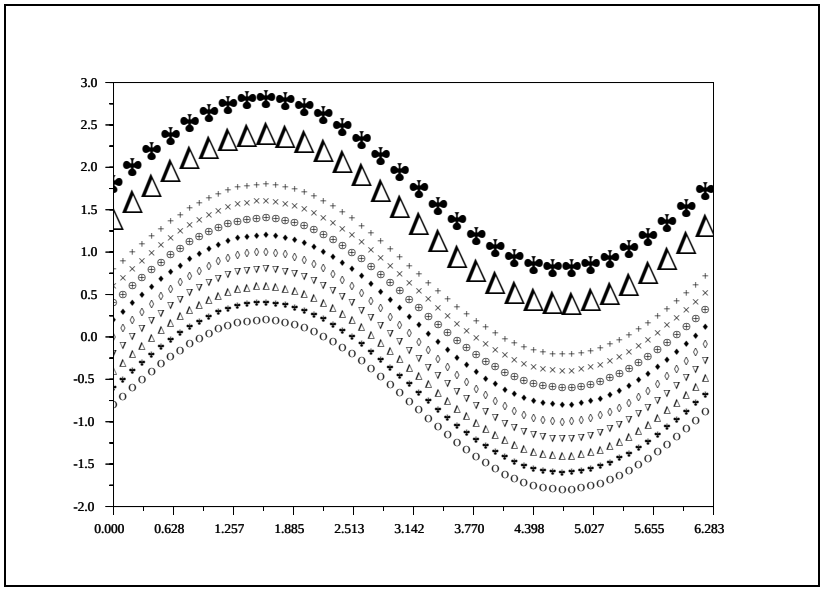


FIG. 5.3 – Black and white plotting styles

frame, axes will not appear on the graph.

- Option `leg` : it is the string of the captions for the different plotted curves . This string is composed of fields separated by the `@` symbol : for example ‘‘`module@phase`’’ (see example below). These strings are displayed under the plot with small segments recalling the styles of the corresponding curves.
- Option `rect` : it is a vector of 4 values specifying the boundaries of the plot `rect=[xmin,ymin,xmax,ymax]`.
- Option `nax` : it is a vector [nx,Nx,ny,Ny] where nx (ny) is the number of subgrads on the x (y) axis and Nx (Ny) is the number of graduations on the x (y) axis.

```
//captions for identifying the curves
//controlling the boundaries of the plot and the tics on axes
x=-5:0.1:5;
y1=sin(x);y2=cos(x);
X=[x;x]; Y=[y1;y2];
plot2d(X',Y',style=[-1 -3]',leg="caption1@caption2",...
rect=[-5,-1,5,1],nax=[2,10,5,5]);
```

For different plots the simple commands without any argument show a demo (e.g `plot2d3()`).

5.4.2 Captions and Presentation

- `xgrid` : adds a grid on a 2D graphic ; the calling parameter is the number of the color.
- `xtitle` : adds title above the plot and axis names on a 2D graphic
- `titlepage` : graphic title page in the middle of the plot

```
//Presentation
x=-%pi:0.1:%pi;
```

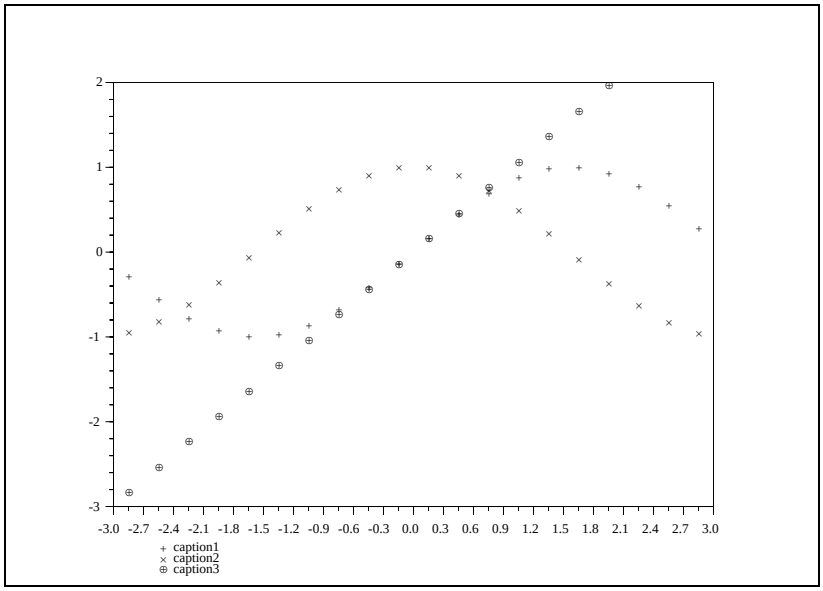


FIG. 5.4 – Box, captions and tics

```
y1=sin(x);y2=cos(x);y3=x;
X=[x;x;x]; Y=[y1;y2;y3];
plot2d(X',Y',style=[-1 -2 -3],leg="caption1@caption2@caption3",...
rect=[-3,-3,3,2],nax=[2,10,2,5]);
xtitle(["General Title";"(with xtitle command)"],...
"x-axis title","y-axis title (with xtitle command)");
xgrid();
xclea(-2.7,1.5,1.5,1.5);
titlepage("Titlepage");
xstring(0.6,.45,"(with titlepage command)");
xstring(0.05,.7,["xstring command after";"xclea command"],0,1);
plot2d(X',Y',style=[-1 -2 -3],leg="caption1@caption2@caption3",...
rect=[-3,-3,3,2],nax=[2,10,2,5]);
```

– `plotframe` : graphic frame with scaling and grid

We have seen that it is possible to control the tics on the axes, choose the size of the rectangle for the plotand add a grid. This operation can be prepared once and then used for a sequence of different plots. One of the most useful aspect is to get graduations by choosing the number of graduations and getting rounded numbers.

```
rect=[-%pi,-1,%pi,1];
tics=[2,10,4,10];
plotframe(rect,tics,[%t,%t],...
['Plot with grids and automatic bounds','angle','velocity']);
```

– `graduate` : a simple tool for computing pretty axis graduations before a plot.

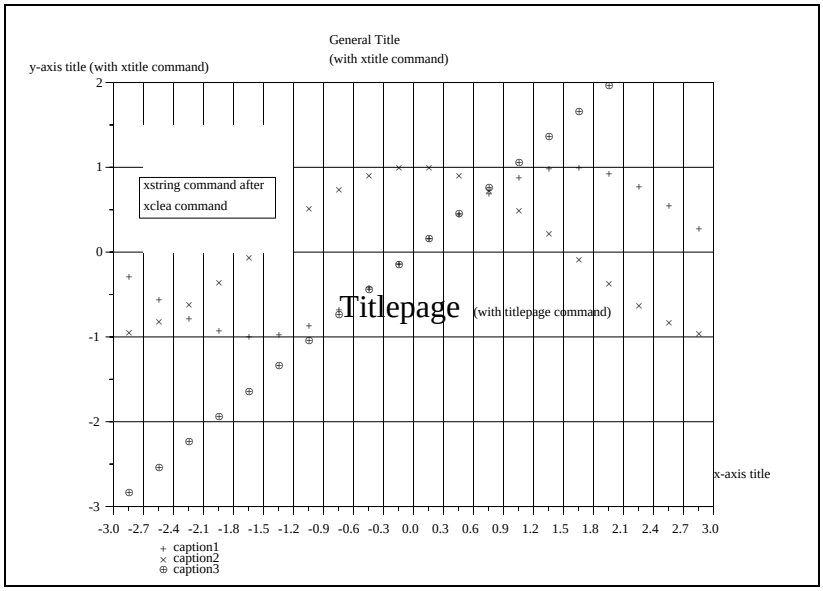


FIG. 5.5 – Grid, Title eraser and comments

5.4.3 Specialized 2D Plottings

- `champ` : vector field in R^2

```
//try champ
x=[-1:0.1:1];y=x;u=ones(x);
fx=x.*.u';fy=u.*.y';
champ(x,y,fx,fy);
xset("font",2,3);
xtitle(['Vector field plot';'(with champ command)']);
//with the color (and a large stacksize)
x=[-1:0.004:1];y=x;u=ones(x);
fx=x.*.u';fy=u.*.y';
champ1(x,y,fx,fy);
```

- `fchamp` : for a vector field in R^2 defined by a function. The same plot than `champ` for a vector field defined for example by a scilab program.
- `fplot2d` : 2D plotting of a curve described by a function. This function plays the same role for `plot2d` than the previous for `champ`.
- `grayplot` : 2D plot of a surface using gray levels ; the surface being defined by the matrix of the values for a grid.
- `fgrayplot` : the same than the previous for a surface defined by a function (scilab program). In fact these 2 functions can be replaced by a usual color plot with an appropriate colormap where the 3 RGB components are the same.

```
R=[1:256]/256;RGB=[R' R' R'];
xset('colormap',RGB);
deff('[z]=surf(x,y)','z=-((abs(x)-1)**2+(abs(y)-1)**2)');
fgrayplot(-1.8:0.02:1.8,-1.8:0.02:1.8,surf,rect=[-2,-2,2,2]);
xset('font',2,3);
xtitle(["Grayplot";"(with fgrayplot command)"]);
```

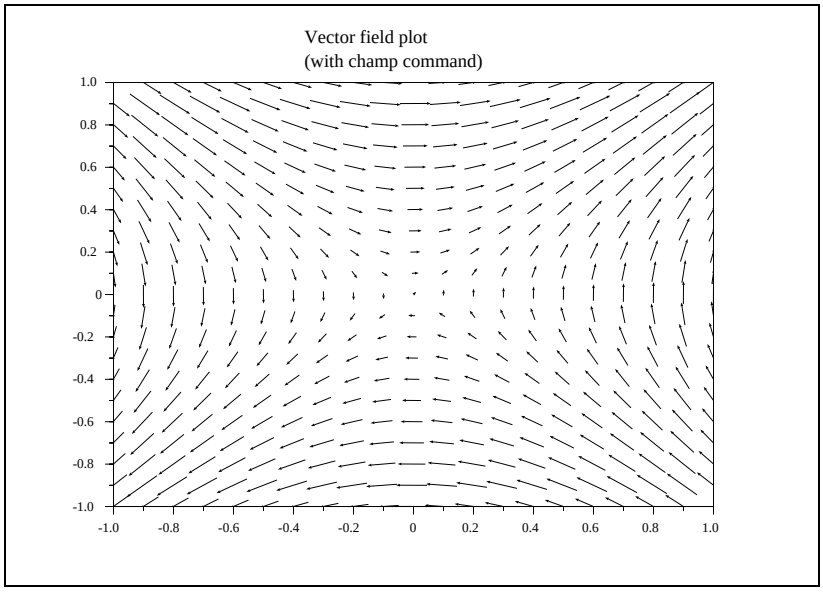


FIG. 5.6 – Vector field in the plane

```
//the same plot can be done with a ‘‘unique’’ given color
R=[1:256]/256;
G=0.1*ones(R);
RGB=[R’ G’ G’];
xset(’colormap’,RGB);
fgrayplot(-1.8:0.02:1.8,-1.8:0.02:1.8,surf,rect=[-2,-2,2,2]);
```

– `errbar` : creates a plot with error bars

5.4.4 Plotting Some Geometric Figures

Polylines Plotting

– `xsegs` : draws a set of unconnected segments
– `xrect` : draws a single rectangle
– `xfrect` : fills a single rectangle
– `xrects` : fills or draws a set of rectangles
– `xpoly` : draws a polyline
– `xpolys` : draws a set of polylines
– `xfpoly` : fills a polygon
– `xfpolys` : fills a set of polygons
– `xarrows` : draws a set of unconnected arrows
– `xfrect` : fills a single rectangle
– `xclea` : erases a rectangle on a graphic window

Curves Plotting

– `xarc` : draws an ellipsis
– `xfarc` : fills an ellipsis
– `xarcs` : fills or draws a set of ellipsis

5.4.5 Writting by Plotting

- `xstring` : draws a string or a matrix of strings
- `xstringl` : computes a rectangle which surrounds a string
- `xstringb` : draws a string in a specified box
- `xnumb` : draws a set of numbers

We give now the sequence of the commands for obtaining the figure **??**.

```
// initialize default environment variables
xset('default');
xset("use color",0);
xset("font",4,3)
xsetech(frect=[1,1,10,10]);
xrect(0,1,3,1)
xfrect(3.1,1,3,1)
xstring(0.5,0.5,"xrect(0,1,3,1)")
xstring(4.,0.5,"xfrect(3.1,1,3,1)")
xset("alufunction",6)
xstring(4.,0.5,"xfrect(3.1,1,3,1)")
xset("alufunction",3)
xv=[0 1 2 3 4]
yv=[2.5 1.5 1.8 1.3 2.5]
xpoly(xv,yv,"lines",1)
xstring(0.5,2.,"xpoly(xv,yv,""lines"",1)")
xa=[5 6 6 7 7 8 8 9 9 5]
ya=[2.5 1.5 1.5 1.8 1.8 1.3 1.3 2.5 2.5 2.5]
xarrows(xa,ya)
xstring(5.5,2.,"xarrows(xa,ya)")
xarc(0.,5.,4.,2.,0.,64*300.)
xstring(0.5,4,"xarc(0.,5.,4.,2.,0.,64*300.)")
xfarc(5.,5.,4.,2.,0.,64*360.)
//xset("alufunction",6)
xclea(5.6,4.4,2.8,0.8);
xstring(5.8,4.,"xfarc  and then  xclea")
//xset("alufunction",3)
xstring(0.,4.5,"WRITING-BY-XSTRING()",-22.5)
xnumb([5.5 6.2 6.9],[5.5 5.5 5.5],[3 14 15],1)
isoview(0,12,0,12)
xarc(-5.,12.,5.,5.,0.,64*360.)
xstring(-4.5,9.25,"isoview + xarc",0.)
A=["  1" "       2" "  3";"  4" "       5" "  6";"68" "  17.2" "  9"];
xstring(7.,10.,A);
rect=xstringl(7,10,A);
xrect(rect(1),rect(2),rect(3),rect(4));
```

e have seen that some parameters of the graphics are controlled by a graphic context (for example the line thickness) and others are controlled through graphics arguments .

- `xset` : to set graphic context values. Some examples of the use of `xset` :

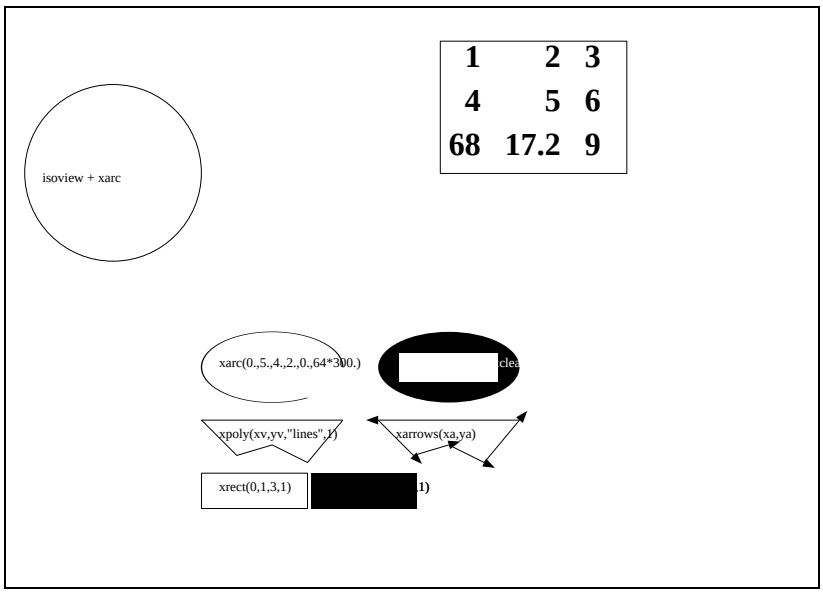


FIG. 5.7 – Geometric Graphics and Comments

geom

(i)-`xset("use color",flag)` changes to color or black and white plot according to the values (1 or 0) of `flag`.
(ii)-`xset("window",window-number)` sets the current window to the window `window-number` and creates the window if it doesn't exist.
(iii)-`xset("wpos",x,y)` fixes the position of the upper left point of the graphic window.
The choice of the font, the width and height of the window, the driver... can be done by `xset`.

- `xget` : to get informations about the current graphic context. All the values of the parameters fixed by `xset` can be obtained by `xget`.
- `xlfont` : to load a new family of fonts

5.4.6 Some Classical Graphics for Automatic Control

- `bode` : plot magnitude and phase of the frequency response of a linear system.
- `gainplot` : same as bode but plots only the magnitude of the frequency response.
- `nyquist` : plot of imaginary part versus real part of the frequency response of a linear system.
- `m_circle` : M-circle plot used with nyquist plot.
- `chart` : plot the Nichols'chart
- `black` : plot the Black's diagram (Nichols'chart) for a linear system.
- `evans` : plot the Evans root locus for a linear system.
- `plzr` : pole-zero plot of the linear system

```
s=poly(0,'s');
h=syslin('c',(s^2+2*0.9*10*s+100)/(s^2+2*0.3*10.1*s+102.01));
h1=h*syslin('c',(s^2+2*0.1*15.1*s+228.01)/(s^2+2*0.9*15*s+225));
//bode
subplot(2,2,1)
gainplot([h1;h],0.01,100);
```

```
//nyquist
subplot(2,2,2)
nyquist([h1;h])

//chart and black
subplot(2,2,3)
black([h1;h],0.01,100,['h1';'h'])
chart([-8 -6 -4],[80 120],list(1,0));
//evans
subplot(2,2,4)
H=syslin('c',352*poly(-5,'s')/poly([0,0,2000,200,25,1],'s','c'));
evans(H,100)
```

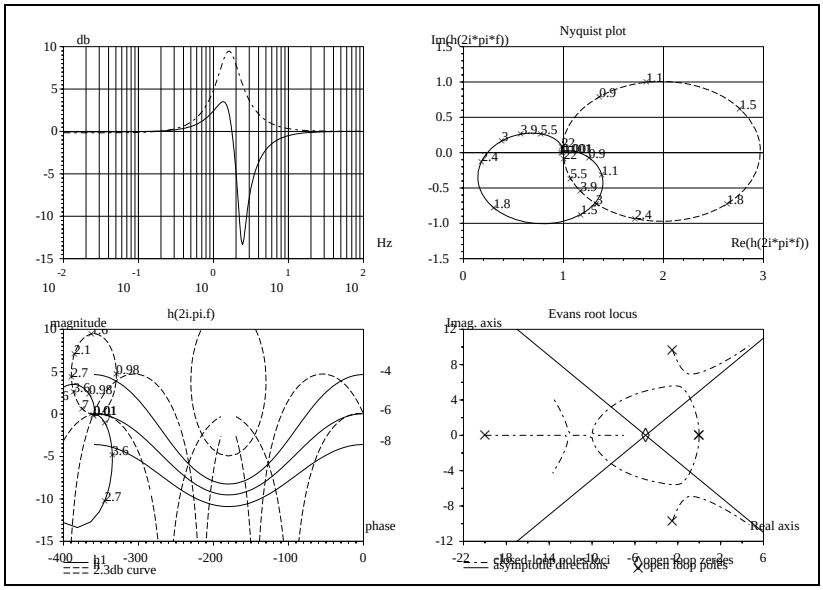


FIG. 5.8 – Some Plots in Automatic Control

ouv8

5.4.7 Miscellaneous

- `edit_curv` : interactive graphic curve editor.
- `gr_menu` : simple interactive graphic editor. It is a Xfig-like simple editor with a flexible use for a nice presentation of graphics : the user can superpose the elements of `gr_menu` and use it with the usual possibilities of `xset`.
- `locate` : to get the coordinates of one or more points selected with the mouse on a graphic window.

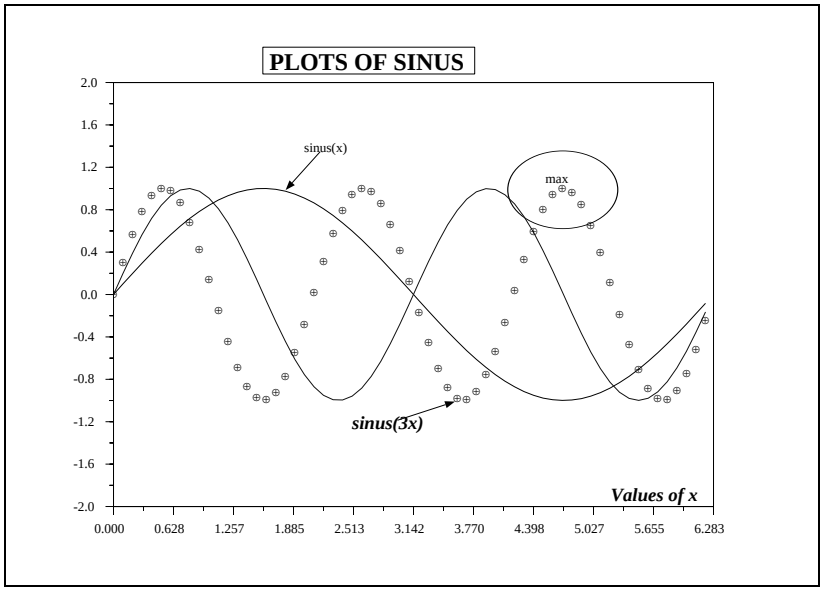


FIG. 5.9 – Presentation of Plots

cifi2

5.5 3D Plotting

5.5.1 Generic 3D Plotting

- `plot3d` : 3D plotting of a matrix of points : plot3d(x,y,z) with x,y,z 3 matrices, z being the values for the points with coordinates x,y. Other arguments are optional
- `plot3d1` : 3d plotting of a matrix of points with gray levels
- `fplot3d` : 3d plotting of a surface described by a function ; z is given by an external z=f(x,y)
- `fplot3d1` : 3d plotting of a surface described by a function with gray levels

5.5.2 Specialized 3D Plotting

- `param3d` : plots parametric curves in 3d space
- `contour` : level curves for a 3d function given by a matrix
- `grayplot10` : gray level on a 2d plot
- `fcontour10` : level curves for a 3d function given by a function
- `hist3d` : 3d histogram
- `secto3d` : conversion of a surface description from sector to plot3d compatible data
- `eval3d` : evaluates a function on a regular grid. (see also feval)

5.5.3 Mixing 2D and 3D graphics

When one uses 3D plotting function, default graphic boundaries are fixed, but in R^3. If one wants to use graphic primitives to add informations on 3D graphics, the `geom3d` function can be used to convert 3D coordinates to 2D-graphics coordinates. The figure 5.10 illustrates this feature.

```
xinit('d7-10.ps');
   r=(%pi):-0.01:0;x=r.*cos(10*r);y=r.*sin(10*r);
```

```
function z=surf(x,y),z=sin(x)*cos(y);endfunction
t=%pi*(-10:10)/10;
fplot3d(t,t,surf,theta=35,alpha=45,leg="X@Y@Z",flag=[-3,2,3]);
z=sin(x).*cos(y);
[x1,y1]=geom3d(x,y,z);
xpoly(x1,y1,"lines");
[x1,y1]=geom3d([0,0],[0,0],[5,0]);
xsegs(x1,y1);
xstring(x1(1),y1(1),' The point (0,0,0)');
```

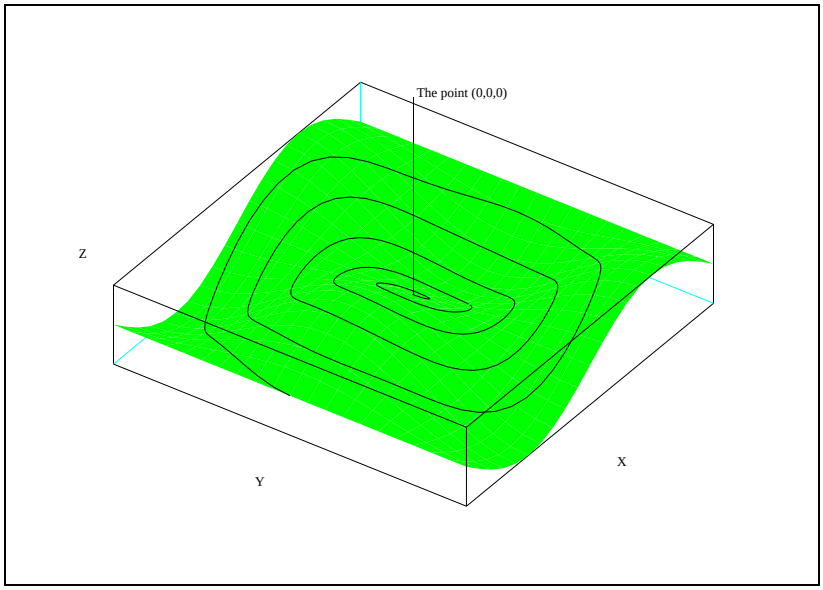


FIG. 5.10 – 2D and 3D plot

5.5.4 Sub-windows

It is also possible to make multiple plotting in the same graphic window (Figure 5.11).

```
xinit('d7-8.ps');
  t=(0:.05:1)';st=sin(2*%pi*t);
  subplot(2,1,1)
  plot2d2(t,st);
  subplot(2,1,2)
  plot2d3(t,st);
  xsetech([0,0,1,1]);
```

5.5.5 A Set of Figures

In this next example we give a brief summary of different plotting functions for 2D or 3D graphics. The figure 5.12 is obtained and inserted in this document with the help of the command `Blatexprs`.

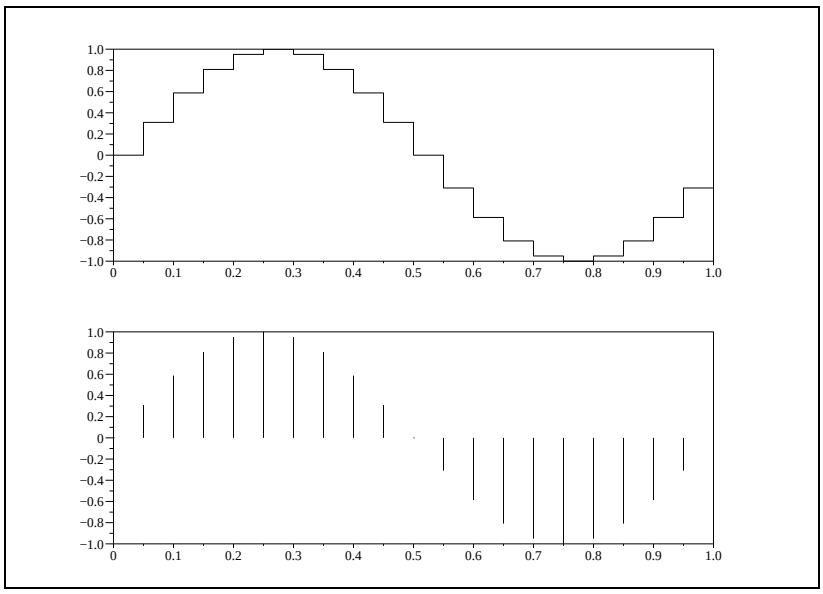


FIG. 5.11 – Use of subplot

```
//some examples
str_l=list();
//
str_l(1)=['plot3d1();';
      'title=[''plot3d1 : z=sin(x)*cos(y)''];';
      'xtitle(title,'' '','' '');'];
//
str_l(2)=['contour();';
      'title=[''contour ''];';
      'xtitle(title,'' '','' '');'];
//
str_l(3)=['champ();';
      'title=[''champ ''];';
      'xtitle(title,'' '','' '');'];
//
str_l(4)=['t=%pi*(-10:10)/10;';
          'deff(''[z]=surf(x,y)'',''z=sin(x)*cos(y)'');';
          'rect=[-%pi,%pi,-%pi,%pi,-5,1];';
          'z=feval(t,t,surf);';
          'contour(t,t,z,10,35,45,''X@Y@Z'',[1,1,0],rect,-5);';
          'plot3d(t,t,z,35,45,''X@Y@Z'',[2,1,3],rect);';
          'title=[''plot3d and contour ''];';
          'xtitle(title,'' '','' '');'];
//
for i=1:4,xinit('d7a11.ps'+string(i)');
          execstr(str_l(i)),xend();end
```

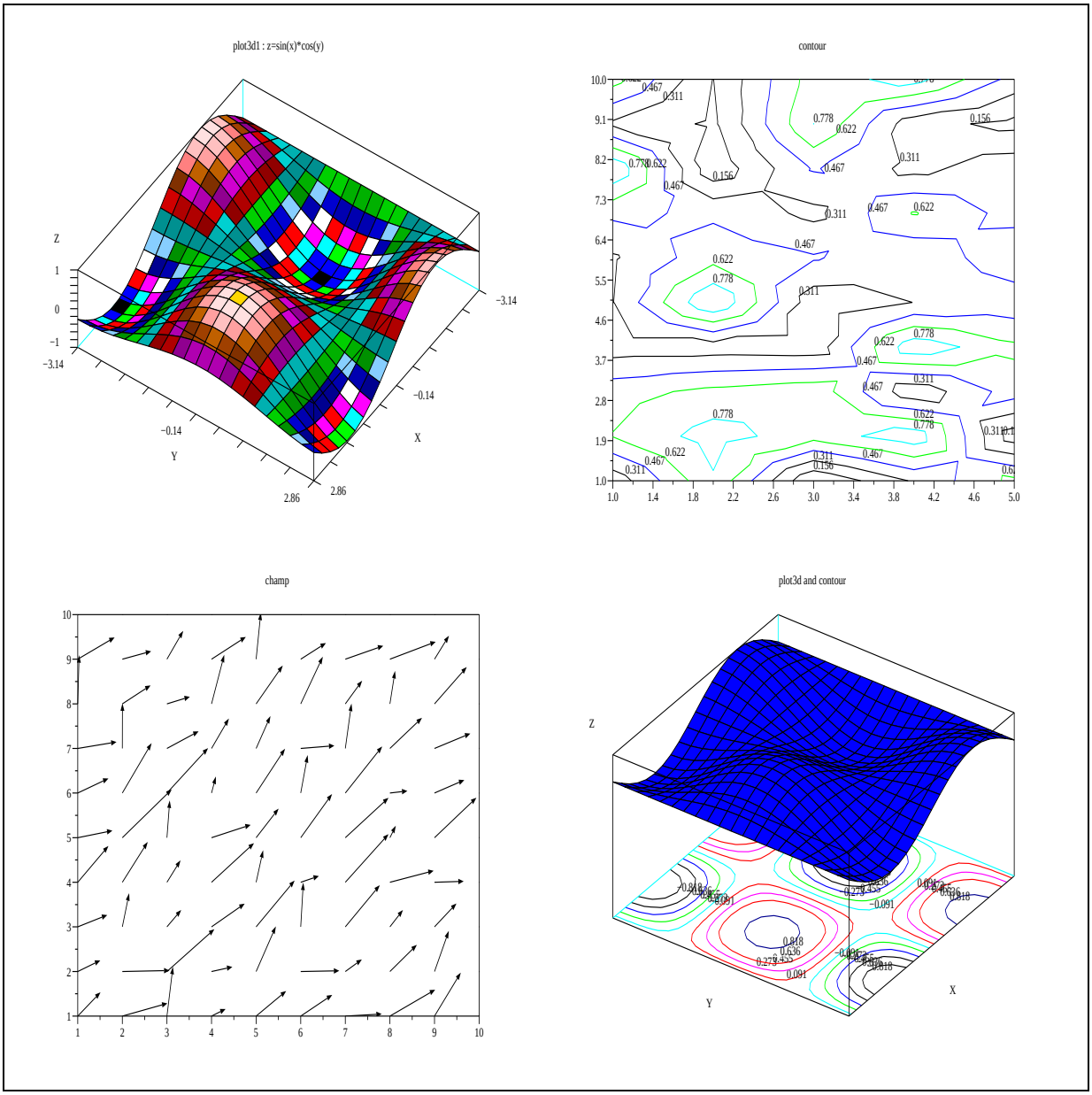


FIG. 5.12 – Group of figures

5.6 Printing and Inserting Scilab Graphics in LaTeX

We describe here the use of programs (Unix shells) for handling Scilab graphics and printing the results. These programs are located in the sub-directory `bin` of Scilab.

5.6.1 Window to Paper

The simplest command to get a paper copy of a plot is to click on the `print` button of the ScilabGraphic window.

5.6.2 Creating a Postscript File

We have seen at the beginning of this chapter that the simplest way to get a Postscript file containing a Scilab plot is :

```
-->driver('Pos')

-->xinit('foo.ps')

-->plot3d1();

-->xend()

-->driver('Rec')

-->plot3d1()

-->xbasimp(0,'foo1.ps')
```

The Postscript files (`foo.ps` or `foo1.ps`) generated by Scilab cannot be directly sent to a Postscript printer, they need a preamble. Therefore, printing is done through the use of Unix scripts or programs which are provided with Scilab. The program `Blpr` is used to print a set of Scilab Graphics on a single sheet of paper and is used as follows :

```
Blpr string-title file1.ps file2.ps > result
```

You can then print the file `result` with the classical Unix command :

```
lpr -Pprinter-name result
```

or use the `ghostview` Postscript interpreter on your Unix workstation to see the result.

You can avoid the file `result` with a pipe, replacing `> result` by the printing command `| lpr` or the previewing command `| ghostview -`.

The best result (best sized figures) is obtained when printing two pictures on a single page.

5.6.3 Including a Postscript File in LaTeX

The `Blatexpr` Unix shell and the programs `Batexpr2` and `Blatexprs` are provided in order to help inserting Scilab graphics in LaTeX.

Taking the previous file `foo.ps` and typing the following statement under a Unix shell :

```
Blatexpr 1.0 1.0 foo.ps
```

creates two files `foo.epsf` and `foo.tex`. The original Postscript file is left unchanged. To include the figure in a LaTeX document you should insert the following LaTeX code in your LaTeX document :

```
\input foo.tex
\dessin{The caption of your picture}{The-label}
```

You can also see your figure by using the Postscript previewer `ghostview`.

The program `Blatexprs` does the same thing : it is used to insert a set of Postscript figures in one LaTeXpicture.

In the following example, we begin by using the Postscript driver `Pos` and then initialize successively 4 Postscript files `fig1.ps, ..., fig4.ps` for 4 different plots and at the end return to the driver `Rec` (X11 driver with record).

```
-->//multiple Postscript files for Latex

-->driver('Pos')


-->t=%pi*(-10:10)/10;



-->plot3d1(t,t,sin(t)'*cos(t),theta=35,alpha=45,flag=[2,2,4]);

-->xend()



-->contour(1:5,1:10,rand(5,10),5);

-->xend()



-->champ(1:10,1:10,rand(10,10),rand(10,10));

-->xend()
```

```
-->t=%pi*(-10:10)/10;

-->function z=surf(x,y),z=sin(x)*cos(y),endfunction

-->rect=[-%pi,%pi,-%pi,%pi,-5,1];

-->z=feval(t,t,surf);

-->contour(t,t,z,10,35,45,'X@Y@Z',[1,1,0],rect,-5);

-->plot3d(t,t,z,theta=35,alpha=45,flag=[2,1,3],ebox=rect);

-->title=['plot3d and contour '];

-->xtitle(title,' ',' ');

-->xend()


-->driver('Rec')
```

Then we execute the command :

```
Blatexprs multi fig1.ps fig2.ps fig3.ps fig4.ps
```

and we get 2 files `multi.tex` and `multi.ps` and you can include the result in a LaTeX source file by :

```
\input multi.tex
\dessin{The caption of your picture}{The-label}
```

Note that the second line `dessin...` is absolutely necessary and you have of course to give the absolute path for the input file if you are working in another directory (see below). The file `multi.tex` is only the definition of the command `dessin` with 2 parameters : the caption and the label ; the command `dessin` can be used with one or two empty arguments `‘‘ ‘‘` if you want to avoid the caption or the label.

The Postscipt files are inserted in LaTeX with the help of the `\special` command and with a syntax that works with the `dvips` program.

The program `Blatexpr2` is used when you want two pictures side by side.

```
Blatexpr2 Fileres file1.ps file2.ps
```

It is sometimes convenient to have a main LaTeX document in a directory and to store all the figures in a subdirectory. The proper way to insert a picture file in the main document, when the picture is stored in the subdirectory `figures`, is the following :

```
\def\Figdir{figures/} % My figures are in the {\tt figures/ } subdirectory.
\input{figures/fig.tex}
\dessin{The caption of you picture}{The-label}
```

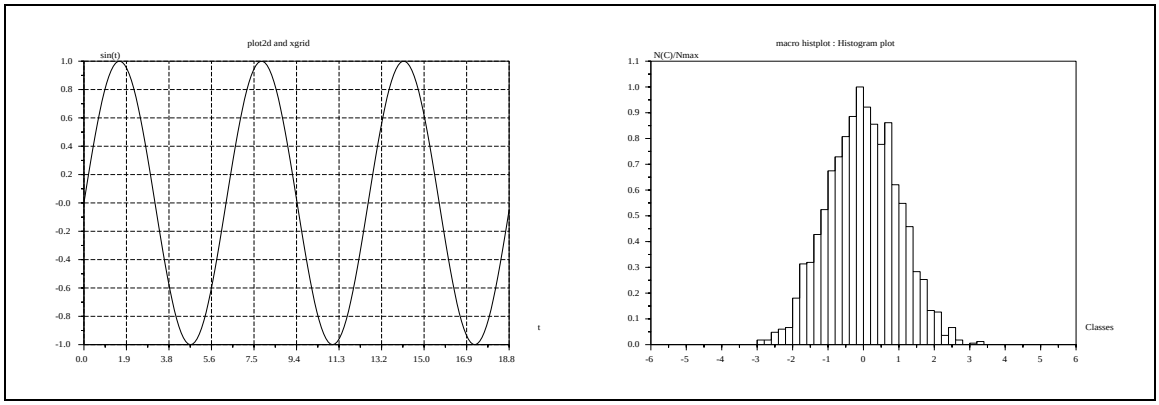


FIG. 5.13 – Blatexp2 Example

The declaration `\def\Figdir{figures/}` is used twice, first to find the file `fig.tex` (when you use `latex`), and second to produce a correct pathname for the `special` LaTeX command found in `fig.tex`. (used at dvips level).

-WARNING : the default driver is `Rec`, i.e. all the graphic commands are recorded, one record corresponding to one window. The `xbasc()` command erases the plot on the active window and all the records corresponding to this window. The `clear` button has the same effect ; the `xclear` command erases the plot but the record is preserved. So you almost never need to use the `xbasc()` or `clear` commands. If you use such a command and if you re-do a plot you may have a surprising result (if you forget that the environment is wiped out) ; the scale only is preserved and so you may have the "window-plot" and the "paper-plot" completely different.

5.6.4 Postscript by Using Xfig

Another useful way to get a Postscript file for a plot is to use Xfig. By the simple command `xs2fig(active-window-number,file-name)` you get a file in Xfig syntax.

This command needs the use of the driver `Rec`.

The window ScilabGraphic0 being active, if you enter :

```
-->t=-%pi:0.3:%pi;

-->plot3d1(t,t,sin(t)'*cos(t),theta=35,alpha=45,flag=[2,2,4]);

-->xs2fig(0,'demo.fig');
```

you get the file `demo.fig` which contains the plot of window 0.

Then you can use Xfig and after the modifications you want, get a Postscript file that you can insert in a LaTeX file. The following figure is the result of Xfig after adding some comments.

5.6.5 Encapsulated Postscript Files

As it was said before, the use of `Blatexpr` creates 2 files : a `.tex` file to be inserted in the LaTeX file and a `.epsf` file.

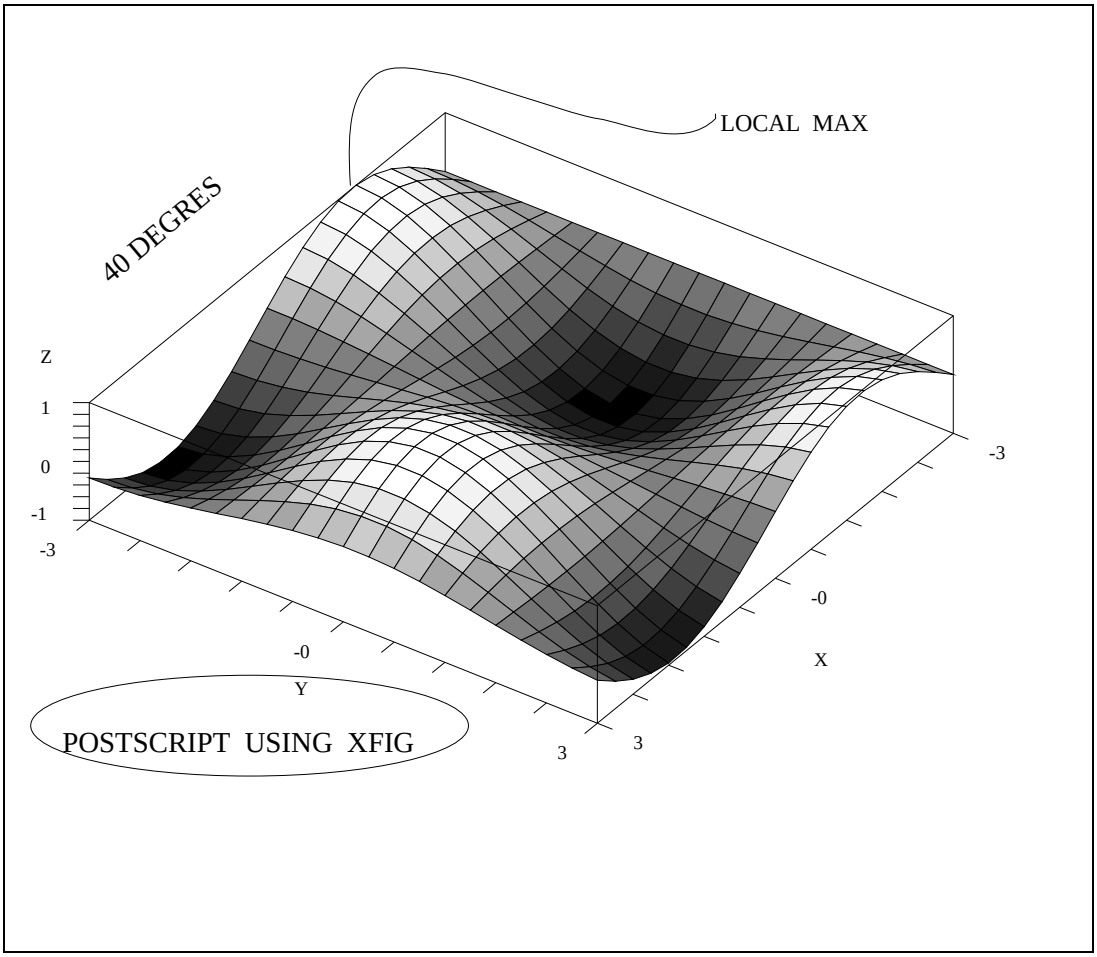


FIG. 5.14 – Encapsulated Postscript by Using Xfig

It is possible to get the encapsulated Postscript file corresponding to a `.ps` file by using the command `BEpsf`.

Notice that the `.epsf` file generated by `Blatexpr` is not an encapsulated Postscript file : it has no bounding box and `BEpsf` generates a `.eps` file which is an encapsulated Postscript file with a bounding box.

Chapitre 6

Interfacing C or Fortran programs with Scilab

Scilab can be easily interfaced with Fortran or C programs. This is useful to have faster code or to use specific numerical code for, e.g., the simulation or optimization of user defined systems, or specific Lapack or `netlib` modules. In fact, interfacing numerical code appears necessary in most nontrivial applications. For interfacing C or Fortran programs, it is of course necessary to link these programs with Scilab. This can be done by a dynamic (incremental) link or by creating a new executable code for Scilab. For executing a C or Fortran program linked with Scilab, its input parameters must be given specific values transferred from Scilab and its output parameters must be transformed into Scilab variables. It is also possible that a linked program is automatically executed by a high-level primitive : for instance the `ode` function can integrate the differential equation $\dot{x} = f(t, x)$ with a rhs function f defined as a C or Fortran program which is dynamically linked to Scilab (see 4.4.2).

The simplest way to call external programs is to use the `link` primitive (which dynamically links the user's program with Scilab) and then to interactively call the linked routine by `call` primitive which transmits Scilab variables (matrices or strings) to the linked program and transforms back the output parameters into Scilab variables. Note that ode/dae solvers and non linear optimization primitives can be directly used with C or Fortran user-defined programs dynamically linked (see 6.1.1). .

An other way to add C or Fortran code to Scilab is by building an interface program. The interface program can be written by the user following the examples given in the following directories `routines/examples/interface-tutorial` and `routines/examples/interface-tour`. Examples of Matlab-like interfaces are given in the directory `routines/examples/mexfiles`.

The interface program can also be generated by `intersci`. `Intersci` builds the interface program from a `.desc` file which describes both the C or Fortran program(s) to be used and the name and parameters of the corresponding Scilab function(s).

Finally it is possible to add a permanent new primitive to Scilab by building an interface program as above and making a new executable code for Scilab. This is done by updating the `fundef` file. In this case, the interface program should be given a specific name (e.g. the default name `matus2`) and a number. The file `default/fundef` should also be updated as done by `intersci`. A new executable code is generated by typing "make all" in the main Scilab directory.

6.1 Using dynamic link

Several simple examples of dynamic link are given in the directory `examples/link-examples`. In this section, we briefly describe how to call a dynamically linked program.

6.1.1 Dynamic link

The command `link('path/pgm.o','pgm',flag)` links the compiled program `pgm` to Scilab. Here `pgm.o` is an object file located in the `path` directory and `pgm` is an entry point (program name) in the file `pgm.o` (An object file can have several entry points : to link them, use a vector of character strings such as `['pgm1','pgm2']`).

`flag` should be set to `'C'` for a C-coded program and to `'F'` for a Fortran subroutine. (`'F'` is the default flag and can be omitted).

If the link operation is OK, scilab returns an integer `n` associated with this linked program. To undo the link enter `ulink(n)`.

The command `c_link('pgm')` returns true if `pgm` is currently linked to Scilab and false if not. Here is a example, with the Fortran BLAS `daxpy` subroutine used in Scilab :

```
-->n=link(SCI+'/routines/blas/daxpy.o','daxpy')
linking files /usr/local/lib/scilab-2.4/routines/calelm/daxpy.o
to create a shared executable.
Linking daxpy (in fact daxpy_)
Link done
 n  =

    0.

-->c_link('daxpy')
 ans  =

  T

-->ulink(n)

-->c_link('daxpy')
 ans  =

  F
```

For more details, enter `help link`.

6.1.2 Calling a dynamically linked program

The `call` function can be used to call a dynamically linked program. Consider for example the `daxpy` Fortran routine. It performs the simple vector operation `y=y+a*x` or, to be more specific,

```
y(1)=y(1)+a*x(1), y(1+incy)=y(1+incy)+a*x(1+incx),...
y(1+n*incy)=y(1+n*incy)+a*x(1+n*incx)
```

where y and x are two real vectors. The calling sequence for daxpy is as follows :

```
      subroutine daxpy(n,a,x,incx,y,incy)
```

To call daxpy from Scilab we must use a syntax as follows :

```
[y1,y2,y3,...]=call('daxpy', inputs description, ...
                    'out', outputs description)
```

Here inputs description is a set of parameters
x1,p1,t1, x2,p2,t2, x3,p3,t3 ...
where xi is the Scilab variable (real vector or matrix) sent to daxpy, pi is the position number of this variable in the calling sequence of daxpy and ti is the type of xi in daxpy (t='i' t='r' t='d' stands for integer, real or double).
outputs description is a set of parameters
[r1,c1],p1,t1, [r2,c2],p2,t2, [r3,c3],p3,t3,..
which describes each output variable. [ri,ci] is the 2 x 1 integer vector giving the number of rows and columns of the ith output variable yi. pi and ti are as for input variables (they can be omitted if a variable is both input and output).

We see that the arguments of call divided into four groups. The first argument 'daxpy' is the name of the called subroutine. The argument 'out' divides the remaining arguments into two groups. The group of arguments between 'daxpy' and 'out' is the list of input arguments, their positions in the call to daxpy, and their data type. The group of arguments to the right of 'out' are the dimensions of the output variables, their positions in the call to daxpy, and their data type. The possible data types are real, integer, and double precision which are indicated, respectively, by the strings 'r', 'i', and 'd'. Here we calculate y=y+a*x by a call to daxpy (assuming that the link command has been done). We have six input variables x1=n, x2=a, x3=x, x4=incx, x5=y, x6=incy. Variables x1, x4 and x6 are integers and variables x2, x3, x5 are double. There is one output variable y1=y at position p1=5. To simplify, we assume here that x and y have the same length and we take incx=incy=1.

```
-->a=3;

-->x=[1,2,3,4];

-->y=[1,1,1,1];

-->incx=1;incy=1;

-->n=size(x,'*');

-->y=call('daxpy',...
        n,1,'i',...
        a,2,'d',...
        x,3,'d',...
        incx,4,'i',...
        y,5,'d',...
        incy,6,'i',...
'out',...
```

```
        [1,n],5,'d');

 y  =

!   4.    7.    10.    13. !
```

(Since y is both input and output parameter, we could also use the simplified syntax `call(...,'out',5)` instead of `call(...,'out'[1,n],5,'d')`).

The same example with the C function `daxpy` (from CBLAS) :

```
int daxpy(int *n, double *da, double *dx, int *incx, double *dy, int *incy)
...
```

```
-->link('daxpy.o','daxpy','C')
linking files daxpy.o  to create a shared executable
Linking daxpy (in fact daxpy)
Link done
 ans  =

    1.

-->y=call('daxpy',...
        n,1,'i',...
        a,2,'d',...
        x,3,'d',...
        incx,4,'i',...
        y,5,'d',...
        incy,6,'i',...
'out',...
        [1,n],5,'d');

-->y
 y  =

!   4.    7.    10.    13. !
```

The routines which are linked to Scilab can also access internal Scilab variables : see the examples in given in the `examples/links` directory.

6.2 Interface programs

6.2.1 Building an interface program

Examples of interface programs are given in the directory `examples/interface-tutorial` and `examples/interface-tour`.

The interface programs use a set of C or Fortran routines which should be used to build the interface program. The simplest way to learn how to build an interface program is to customize the previous skeletons files and to look at the examples provided in this directory. Note that a unique interface program can be used to interface an arbitrary (but less that 99) number of functions.

6.2.2 Example

Let us consider an example given in `examples/interface-tutorial`.

We have the following C function `matmul` which performs a matrix multiplication. Only the calling sequence is important.

```
/*Matrix multiplication C=A*B, (A,B,C stored columnwise) */

#define A(i,k) a[i + k*n]
#define B(k,j) b[k + j*m]
#define C(i,j) c[i + j*n]

void matmul(a,n,m,b,l,c)
double a[],b[],c[];
int n,m,l;
{
int i,j,k; double s;
for( i=0 ; i < n; i++)
   {
     for( j=0; j < l; j++)
        {
        s = 0.;
        for( k=0; k< m; k++)
           {
    s += A(i,k)*B(k,j);
           }
         C(i,j) = s;
         }
   }
}
```

We want to have a new Scilab function (also called `matmul`) which is such that the Scilab command

```
-->C=matmul(A,B)
```

returns in `C` the matrix product `A*B` computed by the above C function. Here `A, B` and `C` are standard numeric Scilab matrices. Thus, the Scilab matrices `A` and `B` should be sent to the C function `matmul` and the matrix `C` should be created, filled, and sent back to Scilab.

To create the Scilab function `matmul`, we have to write the following C gateway function called `intmatmul`. See the file
`SCIDIR/examples/interface-tutorial/intmatmul.c`.

```
#include "stack-c.h"

int intmatmul(fname)
     char *fname;
{
  static int l1, m1, n1, l2, m2, n2, l3;
  static int minlhs=1, maxlhs=1, minrhs=2, maxrhs=2;

  /* Check number of inputs (Rhs=2) and outputs (Lhs=1) */
  CheckRhs(minrhs,maxrhs) ; CheckLhs(minlhs,maxlhs) ;

  /* Get A (#1) and B (#2) as double ("d") */
  GetRhsVar(1, "d", &m1, &n1, &l1);
  GetRhsVar(2, "d", &m2, &n2, &l2);

  /* Check dimensions    */
if (!(n1==m2)) {Scierror(999,"%s: Uncompatible dimensions\r\n",fname);
                return 0;}

  /* Create C (#3) as double ("d") with m1 rows and n1 columns */
```

```
  CreateVar(3, "d", &m1, &n2, &l3);

  /* Call the multiplication function matmul
     inputs:stk(l1)->A, stk(l2)->B  output:stk(l3)->C    */
  matmul(stk(l1), m1, n1, stk(l2), n2, stk(l3));

  /*  Return C (3)  */
  LhsVar(1) = 3;
  return 0;
}
```

Let us now explain each step of the gateway function `intmatmul`. The gateway function must include the file `SCIDIR/routines/stack-c.h`. This is the first line of the file. The name of the routine is `intmatmul` and it admits one input parameter which is `fname`. `fname` must be declared as `char *`. The name of the gateway routine (here `intmatmul`) is arbitrary but the parameter `fname` is compulsory. The gateway routine then includes the declarations of the C variables used. In the gateway function `intmatmul` the Scilab matrices `A`, `B` and `C` are referred to as numbers, respectively `1`, `2` and `3`.

The line

```
CheckRhs(minrhs,maxrhs); CheckLhs(minlhs,maxlhs);
```

is to check that the Scilab function `matmul` is called with a correct number of RHS and LHS parameters. For instance, typing `-->matmul(A)` will give an error message made by `CheckRhs`. The function `CheckRhs` just compares the C variable `Rhs` (transmitted in the include file `stack-c.h`) with the bounds `minrhs` and `maxrhs`.
The next step is to deal with the Scilab variables `A`, `B` and `C`. In a gateway function, all the Scilab variables are referred to as numbers. Here, the Scilab matrices `A`, `B` and `C` are respectively numbered `1`, `2` and `3`. Each input variable of the newly created Scilab function `matmul` (i.e. `A` and `B`) should be processed by a call to `GetRhsVar`. The first two parameters of `GetRhsVar` are inputs and the last three parameters are outputs. The line

```
GetRhsVar(1, "d", &m1, &n1, &l1);
```

means that we process the RHS variable numbered `1` (i.e. `A`). The first parameter of `GetRhsVar` (here `1`) refers to the first parameter (here `A`) of the Scilab function `matmul`. This variable is a Scilab numeric matrix which should be seen (”d”) as a `double` C array, since the C routine `matmul` is expecting a `double` array. The second parameter of `GetRhsVar` (here `"d"`) refers to the type (double, int, char etc) of the variable. From the call to `GetRhsVar` we know that `A` has `m1` rows and `n1` columns.
The line

```
if (n1 !=m2 )
     {Scierror(999,"%s: Uncompatible dimensions\r\n",fname);
                 return 0;}
```

is to make a return to Scilab if the matrices `A` and `B` passed to `matmul` have uncompatible dimensions. The number of columns of `A` should be equal to the number of rows of `B`.
The next step is to create the output variable `C`. This is done by

```
CreateVar(3, "d", &m1, &n2, &l3);
```

Here we create a variable numbered 3 (1 was for A and 2 was for B). It is an array of double ("d"). It has m1 rows and n2 columns. The calling sequence of CreateVar is the same as the calling sequence of GetRhsVar, but the four first parameters of CreateVar are inputs.

The next step is the call to matmul. Remember the calling sequence :

```
void matmul(a,n,m,b,l,c)
double a[],b[],c[]; int n,m,l;
```

We must send to this function (double) pointers to the numeric data in A, B and C. This is done by :

```
matmul(stk(l1), m1, n1, stk(l2), n2, stk(l3));
```

Here stk(l1) is a double pointer to the content of the A matrix. The entries of the A matrix are stored columnwise in stk(l1)[0], stk(l1)[1] etc. Similarly, after the call to the C function matmul the (double) numbers stk(l3)[0], stk(l3)[1] are the values of the matrix product A*B stored columnwise as computed by matmul. The last parameter of the functions GetRhsVar and CreateVar is an output parameter which allow to access the data through a pointer (here the double pointers stk(l1), stk(l2) and stk(l3).

The final step is to return the result, i.e. the C matrix to Scilab. This is done by

```
LhsVar(1) = 3;
```

This statement means that the first LHS variable of the Scilab function matmul is the variable numbered 3.

Once the gateway routine is written, it should be compiled, linked with Scilab and a script file should be executed in Scilab for loading the new function.

It is possible to build a static or a dynamic library. The static library corresponding the the example just described here is built in the directory SCIDIR/examples/interface-tutorial and the dynamic library is built into the directory SCIDIR/examples/interface-tutorial-so.

Static library

In the directory SCIDIR/examples/interface-tutorial just enter the make command in an Unix platform or in the Windows environment with the Visual C++ environment enter nmake /f Makefile.mak. This command produces the following file tutorial_gateway.c which is a C function produced by the Makefile :

```
#include "mex.h"
extern Gatefunc intview;
extern Gatefunc intmatmul;

static GenericTable Tab[]={
{(Myinterfun)sci_gateway, intview,"error msg"},
{(Myinterfun)sci_gateway, intmatmul,"error msg"},
         };

int C2F(tutorial_gateway)()
{  Rhs = Max(0, Rhs);
(*(Tab[Fin-1].f))(Tab[Fin-1].name,Tab[Fin-1].F);
  return 0;
}
```

This function is essentially the table of C functions which are dynamically linked wih Scilab.

The following file tutorial.sce is also produced by the Makefile :

```
scilab_functions =[...
"view";
"matmul";
          ];
auxiliary="";
files=G_make(["tutorial_gateway.o","tutorial.a", auxiliary],"void(Win)");
addinter(files,"tutorial_gateway",scilab_functions);
```

The Scilab function `addinter` makes the correspondance between the C gateway functions (such as `intmatmul`) and their names as Scilab functions.

To load the newly created function `matmul`, one has to execute this script and then the function `matmul` can be called into Scilab

```
-->exec tutorial.sce
```

```
-->A=rand(2,3);B=rand(3,3);C=matmul(A,B);    //C=A*B
```

Summing up, to build an static interface, the user has to write a gateway function such as `intmatmul`. Then he has to edit the Makefile in `SCIDIR/examples/interface-tutorial` (or a copy of it) and to put there the name of his gateway function(s) (e.g. `intmatmul.o`) in the target `CINTERFACE` and the name of the corresponding Scilab function (e.g. `matmul`) in the target `CFUNCTIONS` with the same ordering. Typing `make` produces the static library and a script file (here `tutorial.sce`) which should be executed each time the newly created function(s) are needed. Of course, it is possible to perform this operation automatically when Scilab is launched by creating a startup file `.scilab` identical to `tutorial.sce`.

Dynamic library

The directory `SCIDIR/examples/interface-tutorial-so` contains the material necessary to create a dynamic library (or a dll in the Windows environment) that can be dynamically linked with Scilab. This directory contains the following file called `builder.sce` :

```
// This is the builder.sce
// must be run from this directory

ilib_name  = "libtutorial"           // interface library name
files = ["intview.o","intmatmul.o"]  // objects files
                                     //
libs  = []                           // other libs needed for linking
table = [ "view", "intview";         // table of (scilab_name,interface-name)
          "matmul","intmatmul"];     //

// do not modify below
// -------------------------------------------
ilib_build(ilib_name,table,files,libs)
```

The user should edit this file, which is a Scilab script, and in particular the variables `files` (a row vector of strings) anf `table` a two column matrix of strings. `files` should contain the names of all the object files (gateway functions and C functions called). Each row of `table` is a pair of two strings : the first is the name of the Scilab function, and the second the name of the gateway function. Here we have two functions `view` which has `intview` as gateway and `matmul` which has `intmatmul` as gateway. This is the example given above. After the file `builder.sce` has been edited, it should be executed in Scilab by the command

```
-->exec builder.sce
```

Scilab then generates the file `loader.sce` :

```
// generated by builder.sce
libtutorial_path=get_file_path('loader.sce');
functions=[ 'view';
            'matmul';
];
addinter(libtutorial_path+'/libtutorial.so','libtutorial',functions);
```

This file should be executed in Scilab to load the newly created function `matmul`

```
-->exec loader.sce

-->A=rand(2,3);B=rand(3,3);C=matmul(A,B);   //C=A*B
```

Summing up, to build a dynamic interface the user has to write a gateway function (such as `intmatmul`), then he has to edit the file `builder.sce` (or a copy of it) to enter the name of the Scilab function and the necessary C functions, then he has to execute the script `builder.sce`. This produce the dynamic library and the script `loader.sce`. Then each time he needs the newly created function(s), he has to execute the script `loader.sce`.

6.2.3 Functions used for building an interface

The functions used to build an interface are Fortran subroutines when the interface is written in Fortran and are coded as C macros (defined in `stack-c.h`) when the interface is coded in C. The main functions are as follows :

- `CheckRhs(minrhs, maxrhs)`
  `CheckLhs(minlhs, maxlhs)`
  Function `CheckRhs` is used to check that the Scilab function is called with `minrhs <= Rhs <= maxrhs`. Function `CheckLhs` is used to check that the expected return values are in the range `minlhs <= Lhs <= maxlhs`. (Usually one has `minlhs=1` since a Scilab function can be always be called with less lhs arguments than expected).
- `GetRhsVar(k,ct,&mk,&nk,&lk)`
  Note that `k` (integer) and `ct` (string) are inputs and `mk,nk` and `lk` (integers) are outputs of `GetRhsVar`. This function defines the type (`ct`) of input variable numbered `k`, i.e. the `k`th input variable in the calling sequence of the Scilab function. The pair `mk,nk` gives the dimensions (number of rows and columns) of variable numbered `k` if it is a matrix. If it is a chain `mk*nk` is its length. `lk` is the adress of variable numbered `k` in Scilab internal stack. The type of variable number `k`, `ct`, should be set to `"d", "r", "i" ,"z"` or `"c"` which stands for double, float (real), integer, double complex or character respectively. The interface should call function `GetRhsVar` for each of the rhs variables of the Scilab function with `k=1, k=2,..., k=Rhs`. Note that if the Scilab argument doesn't match the requested type then Scilab enters an error function and returns from the interface function.
- `CreateVar(k,ct,&mk,&nk,&lk)`
  Here `k,ct,&mk,&nk` are inputs of `CreateVar` and `lk` is an output of `CreateVar`. The parameters are as above. Variable numbered `k` is created in Scilab internal stack at adress `lk`. When calling `CreateVar`, `k` must be greater than `Rhs` i.e. `k=Rhs+1, k=Rhs+2, ....` If due to memory lack, the argument can't be created, then a Scilab error function is called and the interface function returns.
- `CreateVarFromPtr(k,ct,&mk,&nk,&lk)`

Here `k,ct,&mk,&nk,&lk` are all inputs of `CreateVarFromPtr` and `lk` is pointer created by a call to a C function. This function is used when a C object was created inside the interfaced function and a Scilab object is to be created using a pointer to this C object.

Once the variables have been processed by `GetRhsVar` or created by `CreateVar`, they are given values by calling one or several numerical routine. The call to the numerical routine is done in such a way that each argument of the routine points to the corresponding Scilab variable. Character, integer, real, double and double complex type variables are respectively in the `cstk`, `istk`, `sstk`, `stk`, `zstk` Scilab internal stack at the adresses `lk`'s returned by `GetRhsVar` or `CreateVar`.

Then they are returned to Scilab as lhs variables. The interface should define how the lhs (output) variables are numbered. This is done by the global variable `LhsVar`. For instance

```
LhsVar(1) = 5;
LhsVar(2) = 3;
LhsVar(3) = 1;
LhsVar(4) = 2;
```

means that the Scilab function has at most 4 output parameters which are variables numbered `k= 5, k=3, k=1, k=2` respectively.

The functions `sciprint(amessage)` and `Error(k)` are used for managing messages and errors.

Other useful functions which can be used are the following.

- `GetMatrixptr("Aname", &m, &n, &lp);`
  This function reads a matrix in Scilab internal stack. `Aname` is a character string, name of a Scilab matrix. Outputs are integers `m,n` and `lp`, the entries of the matrix are ordered *columnwise*.
- `ReadString("Aname",&n,str)`
  This function reads a string in Scilab internal stack. `n` is the length of the string.

The Fortran functions have the same syntax and return logical values.

6.2.4 Examples

There are many examples of external functions interfaced with Scilab in the directories `SCIDIR/examples/inter` and
`SCIDIR/examples/interface-tour-so`. Examples are given in C and Fortran. The best way to build an interface is to copy one of the examples given there and to adapt the code to particular needs.

6.2.5 The `addinter` command

Once the interface program is written, it must be compiled to produce an object file. It is then linked to Scilab by the addinter command.

The syntax of addinter is the following :

```
addinter([‘interface.o’, ’userfiles.o’],’entrypt’,[’scifcts’])
```

Here `interface.o` is the object file of the interface, `userfiles.o` is the set of user's routines to be linked, `entrypt` is the entry point of the interface routine and 'scifcts' is the set of Scilab functions to be interfaced.

6.3 Intersci

The directory `SCIDIR/examples/intersci-examples-so` contains several examples for using `intersci` which is a tool for producing gateway routines from a descriptor file. Let us describe a simple example, `ex01`. We want to build an interface for the following C function :

```
int ext1c(n, a, b, c)
     int *n;
     double *a, *b, *c;
{
  int k;
  for (k = 0; k < *n; ++k)
      c[k] = a[k] + b[k];
  return(0);
}
```

This function just adds the two real vectors `a` and `b` with `n` entries and returns the result in `c`. We want to have in Scilab a function `c=ext1c(a,b)` which performs this operation by calling `ext1c`. For that, we provide a `.desc` file, `ex01fi.desc` :

```
ext1c a b
a  vector m
b  vector m
c vector  m

ext1c   m a b c
m integer
a double
b double
c double

out sequence c
**********************
```

This file in divided into three parts separated by a blank line. The upper part (four first lines) describes the Scilab function `c=ext1c(a,b)`. Then (next five lines) the C function is described. The last line of `ex01fi.desc` gives the name of output variables. To run `intersci` with this file as input we enter the command :

```
SCIDIR/bin/intersci-n ex01fi
```

Two files are created : `ex01fi.c` and `ex01fi_builder.sce`. The file `ex01fi.c` is the C gateway function needed for interfacing `ext1c` with Scilab. It is a gateway file built as explained above (see 6.2.2) :

```
#include "stack-c.h"

int intsext1c(fname)
   char *fname;
{
 int m1,n1,l1,mn1,m2,n2,l2,mn2,un=1,mn3,l3;
 CheckRhs(2,2);
```

```
 CheckLhs(1,1);
 /*  checking variable a */
 GetRhsVar(1,"d",&m1,&n1,&l1);
 CheckVector(1,m1,n1);
 mn1=m1*n1;
 /*  checking variable b */
 GetRhsVar(2,"d",&m2,&n2,&l2);
 CheckVector(2,m2,n2);
 mn2=m2*n2;
 /* cross variable size checking */
 CheckDimProp(1,2,m1*n1 != m2*n2);
 CreateVar(3,"d",(un=1,&un),(mn3=mn1,&mn3),&l3);/* named: c */
 C2F(ext1c)(&mn1,stk(l1),stk(l2),stk(l3));
 LhsVar(1)= 3;
 return 0;
}
```

The file `ex01fi_builder.sce` is the following :

```
// generated with intersci
ilib_name = 'libex01fi'// interface library name

table =["ext1c","intsext1c"];
ilib_build(ilib_name,table,files,libs);
```

This builder file is to be executed by Scilab after the variables `files` and `libs` have been set :

```
-->files = ['ex01fi.o' , 'ex01c.o'];
-->libs  = [] ;
-->exec ex01fi_builder.sce
```

A dynamic library is then created as well as a file `loader.sce`. Executing `loader.sce` loads the library into Scilab and executes the `addinter` command to link the library and associate the name of the function `ext1c` to it. We can then call the new function ;

```
-->exec loader.sce
-->a=[1,2,3];b=[4,5,6]; c=ext1c(a,b);
```

To use `intersci` one has to construct a `.desc` file. The keywords which describe the Scilab function and the function to be called can be found in the examples given.

6.4 Argument functions

Some built-in nonlinear solvers, such as `ode` or `optim`, require a specific function as argument. For instance in the Scilab command `ode(x0,t0,t,fydot)`, `fydot` is the specific argument function for the `ode` primitive. This function can be a either Scilab function or an external function written in C or Fortran. In both cases, the argument function must obey a specific syntax. In the following we will consider, as running example, using the `ode` primitive with a rhs function written in Fortran. The same steps should be followed for all primitives which require a function as argument.

If the argument function is written in C or Fortran, there are two ways to call it :

– -Use dynamic link

```
-->link('myfydot.o','myfydot')
//or -->link('myfydot.o','myfydot','C')
-->ode(x0,t0,t,'myfydot')
```

– -Use the `Ex-ode.f` interface in the `routines/default` directory (and `make all` in Scilab directory). The call to the `ode` function is as above :
`-->ode(x0,t0,t,'myfydot')`

In this latter case, to add a new function, two files should be updated :

– The `Flist` file : Flist is list of entry points. Just add the name of your function at in the appropriate list of functions.
`ode_list= ... myfydot`
– The `Ex-ode.f` (or `Ex-ode-more.f`) file : this file contains the source code for argument functions. Add your function here.

Many exemples are provided in the `default` directory. More complex examples are also given. For instance it is shown how to use Scilab variables as optional parameters of `fydot`.

6.5 Mexfiles

The directories under `SCIDIR/examples/mexfiles` contain some examples of Matlab mexfiles which can be used as interfaces in the Scilab environment. The Scilab `mexlib` library emulates the most commonly used Matlab `mxfunctions` such as `mxGetM`, `mxGetPr`, `mxGetIr` etc. Not all the `mxfunctions` are available but standard mexfiles which make use of matrices (possibly sparse), character strings and n-dimensional arrays can be used without any modification in the Scilab environment.

6.6 Maple to Scilab Interface

To combine symbolic computation of the computer algebra system Maple with the numerical facilities of Scilab, Maple objects can be transformed into Scilab functions. To assure efficient numerical evaluation this is done through numerical evaluation in Fortran. The whole process is done by a Maple procedure called `maple2scilab`.

6.7 Maple2scilab

The procedure `maple2scilab` converts a Maple object, either a scalar function or a matrix into a Fortran subroutine and writes the associated Scilab function. The code of `maple2scilab` is in the directory `SCIDIR/maple`.

The calling sequence of `maple2scilab` is as follows :
`maple2scilab(function-name,object,args)`

– The first argument, `function-name` is a name indicating the function-name in Scilab.
– The second argument `object` is the Maple name of the expression to be transferred to Scilab.
– The third argument is a list of arguments containing the formal parameters of the Maple-object `object`.

When `maple2scilab` is invoked in Maple, two files are generated, one which contains the Fortran code and another which contains the associated Scilab function. Aside their existence, the user has not to know about their contents.

The Fortran routine which is generated has the following calling sequence :
`<Scilab-name>(x1,x2,...,xn,matrix)`
and this subroutine computes matrix(i,j) as a function of the arguments `x1,x2,...,xn`. Each argument can be a Maple scalar or array which should be in the argument list. The Fortran subroutine is

put into a file named <Scilab-name>.f, the Scilab-function into a file named <Scilab-name>.sci. For numerical evaluation in Scilab the user has to compile the Fortran subroutine, to link it with Scilab (e.g. Menu-bar option 'link') and to load the associated function (Menu-bar option 'getf'). Information about link operation is given in Scilab's manual : Fortran routines can be incorporated into Scilab by dynamic link or through the Ex-fort.f file in the default directory. Of course, this two-step procedure can be automatized using a shell-script (or using unix in Scilab). Maple2scilab uses the "Macrofort" library which is in the share library of Maple.

6.7.1 Simple Scalar Example

Maple-Session

```
> read(‘maple2scilab.maple‘):
> f:=b+a*sin(x);
```

$$f := b + a \sin(x)$$

```
> maple2scilab('f_m',f,[x,a,b]);
```

Here the Maple variable f is a scalar expression but it could be also a Maple vector or matrix. 'f_m' will be the name of f in Scilab (note that the Scilab name is restricted to contain at most 6 characters). The procedure maple2scilab creates two files : f_m.f and f_m.sci in the directory where Maple is started. To specify another directory just define in Maple the path : rpath:=‘/work/‘; then all files are written in the sub-directory work. The file f_m.f contains the source code of a stand alone Fortran routine which is dynamically linked to Scilab by the function f_m in defined in the file f_m.sci.

Scilab Session

```
-->unix('make f_m.o');

-->link('f_m.o','f_m');

linking  _f_m_ defined in f_m.o

-->getf('f_m.sci','c')

-->f_m(%pi,1,2)
 ans       =

    2.
```

6.7.2 Matrix Example

This is an example of transferring a Maple matrix into Scilab.

Maple Session

```
> with(linalg):read(‘maple2scilab.maple‘):

> x:=vector(2):par:=vector(2):

> mat:=matrix(2,2,[x[1]^2+par[1],x[1]*x[2],par[2],x[2]]);

                          [     2                     ]
                          [ x[1]  + par[1]  x[1] x[2] ]
                 mat := [                             ]
                          [     par[2]         x[2]   ]

> maple2scilab('mat',mat,[x,par]);
```

Scilab Session

```
-->unix('make mat.o');

-->link('mat.o','mat')

linking  _mat_ defined in mat.o

-->getf('mat.sci','c')

-->par=[50;60];x=[1;2];

-->mat(x,par)
 ans       =

!   51.    2. !
!   60.    2. !
```

Generated code Below is the code (Fortran subroutines and Scilab functions) which is automatically generated by `maple2scilab` in the two preceding examples.

Fortran routines

```
c
c     SUBROUTINE f_m
c
      subroutine f_m(x,a,b,fmat)
      doubleprecision x,a,b
      implicit doubleprecision (t)
      doubleprecision fmat(1,1)
         fmat(1,1) = b+a*sin(x)
      end

c
c     SUBROUTINE mat
```

```
c
      subroutine mat(x,par,fmat)
      doubleprecision x,par(2)
      implicit doubleprecision (t)
      doubleprecision fmat(2,2)
         t2 = x(1)**2
         fmat(2,2) = x(2)
         fmat(2,1) = par(2)
         fmat(1,2) = x(1)*x(2)
         fmat(1,1) = t2+par(1)
      end
```

Scilab functions

```
function [var]=f_m(x,a,b)
var=call('f_m',x,1,'d',a,2,'d',b,3,'d','out',[1,1],4,'d')

function [var]=fmat(x,par)
var=call('fmat',x,1,'d',par,2,'d','out',[2,2],3,'d')
```

Table des figures

Index