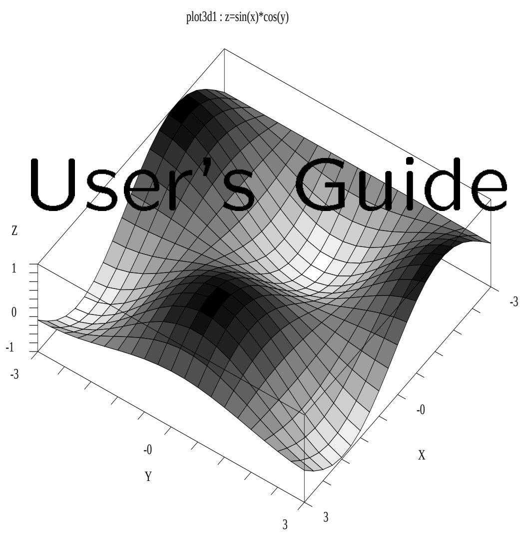
Introduction

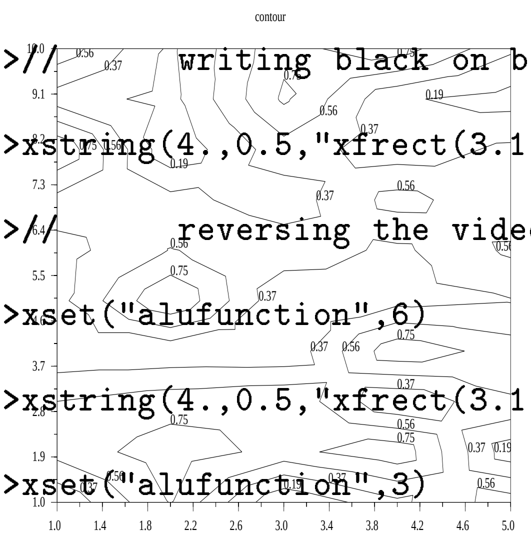
To

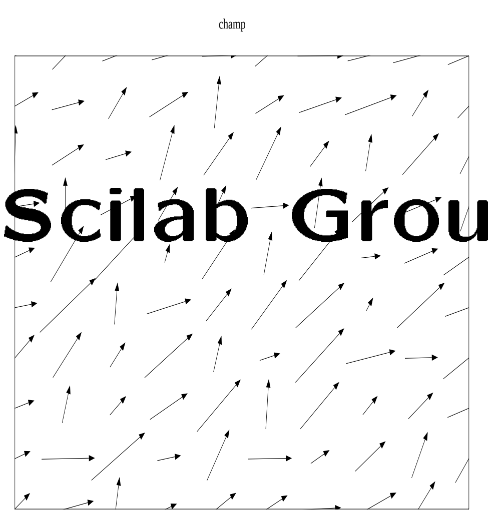
Scilab

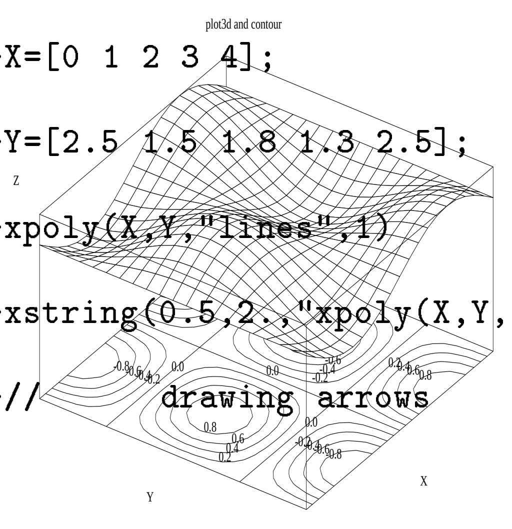
User's Guide

```
-->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
```









Scilab Group

INTRODUCTION
TO
SCILAB

Scilab Group
INRIA Meta2 Project/ENPC Cergrene

INRIA - Unité de recherche de Rocquencourt - Projet Meta2
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)
E-mail : scilab@inria.fr
Home page : http://www-rocq.inria.fr/scilab

Contents

Chapter 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 file `notice.tex`).

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 2.6).

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 files `scilab.star` (startup file), the copyright file `notice.tex`, and the `configure` file (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. The file `alldems.dem` allows to add a new demo which can be run by clicking the "Demos" button. This directory contains the codes corresponding to various demos. They are often useful for inspiring new users. 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`. See also the manual (on-line `help`) in the directory `SCIDIR/man`
- `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 `items` 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 editing the file `SCIDIR/man/Chapters`. See the README file.
- `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).
- `intersci` contains the program provided for building interface programs necessary to add 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.
- `tmp` : some examples written by users for courses, etc have been added in this directory.
- `util` contains some utility functions for calling Scilab as a sub-routine or for making the documentation

- `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 Manual. Both are distributed as Postscript files and generated by a set of Makefiles. In addition, there are 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-2.4  ( 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 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

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=1000000.000000*(a*sin(A))^2+...
-->     2000000.000000*a*b*sin(A)*cos(A)+...
-->     1000000.000000*(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     MSDOS      home       PWD
 TMPDIR     percentlib            fraclablib            soundlib   xdesslib
 utillib    tdcslib    siglib     s2flib     roblib     optlib     metalib
 elemlib    commlib    polylib    autolib    armalib    alglib     mtlblib
 SCI        %F         %T         %z         %s         %nan       %inf
 old        newstacksize          $          %t         %f         %eps
 %io        %i         %e         %pi
 using        3978 elements  out of    1000000.
          and         43 variables out of       1023
```

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)',[-1],'011',' ',[0,0,3.5,0.7],[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.    0. !

       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.

....

```
-->deff('[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)' ])
```

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.       !
! - 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 `deff` 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 `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')
Tue Oct 27 14:28:31 MET 1998
```

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;}  !

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

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

-->link('foo.o','foo','C');

-->deff('[c]=myplus(a,b)',...
-->   'c=fort(''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`.

. . . .

```
-->deff('[ydot]=f(t,y)','ydot=[a-y(2)*y(2) -1;1 0]*y')

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

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

-->plot2d(y(1,:)',y(2,:)',[-1],'011',' ',[-3,-3,3,3],[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

-->deff('[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,'])

-->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 (note that the double quote is necessary because the body of the `deff` contains quotes inside of quotes). 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.

....

```
-->deff('[y]=calcul(x,method)','z=method(x),y=poly(z,''x'')')

-->deff('[z]=meth1(x)','z=x')

-->deff('[z]=meth2(x)','z=2*x')

-->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.

. . . .

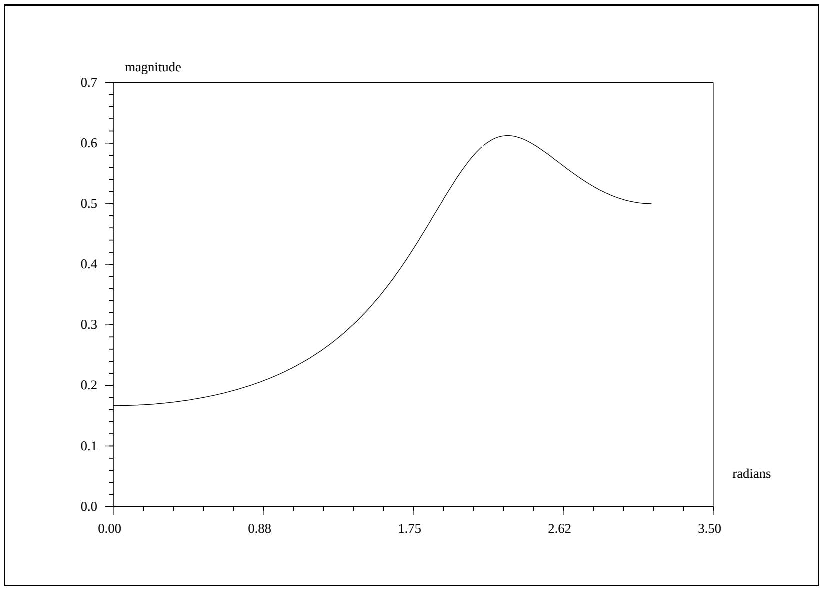


Figure 1.1: A Simple Response

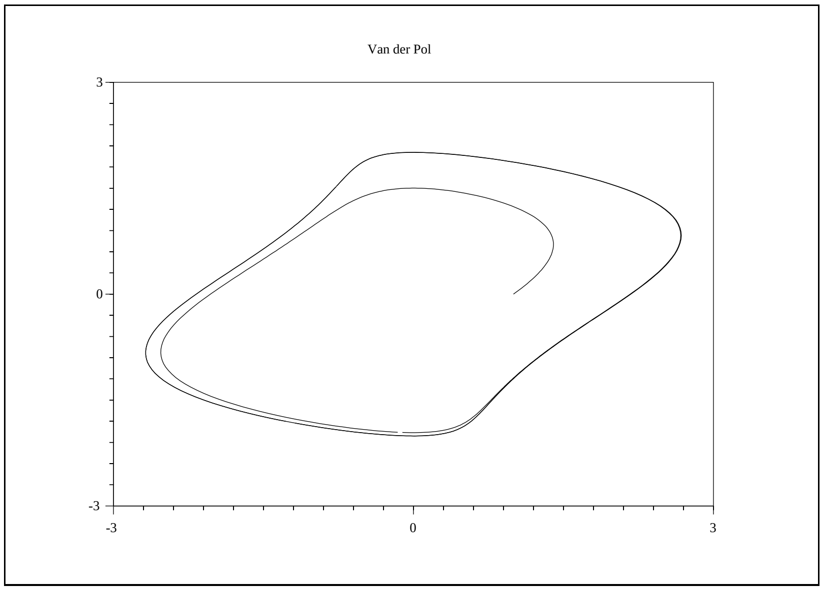


Figure 1.2: Phase Plot

Chapter 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 2.6). 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 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.

2.7 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
   -----
  - 1 + s

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

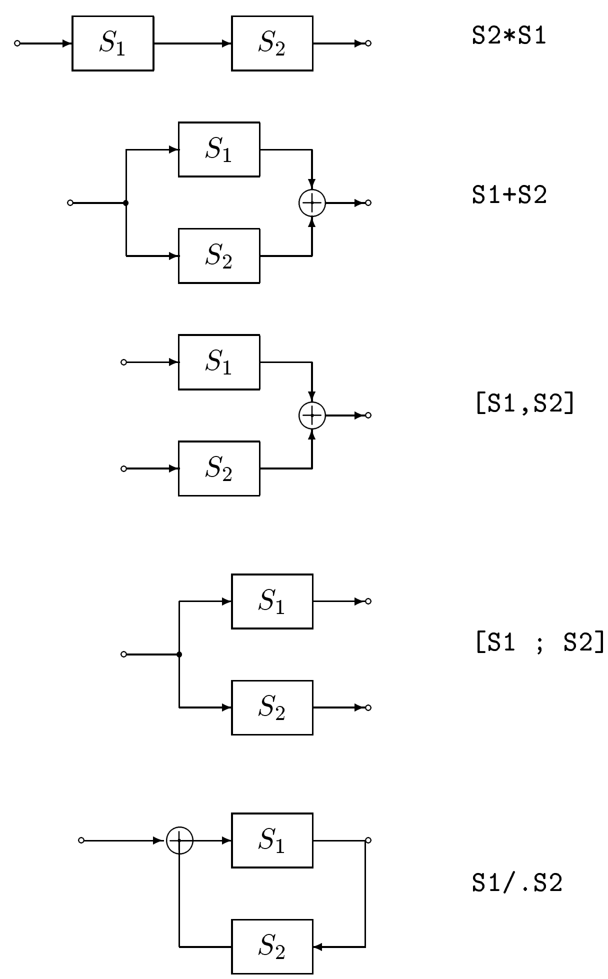


Figure 2.1: Inter-Connection of Linear Systems

```
 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.

2.8 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 `deff` primitive.

```
--> 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 `getf('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.

2.9 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.

2.10 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.
- `state-space (or rational)` for syslin lists.
- `library` for library definition.

2.11 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

2.12 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`.

2.12.1 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  !
```

2.12.2 Indexing in lists

The following session illustrates how to create lists and insert/extract entries in `lists` and `tlists`.

```
-->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  =


       l(1)

    1.

       l(2)

 qwerw
```

```
       l(3)

    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.
```

Chapter 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.

`==` or `=`	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,
```

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

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

    0.

--> getf('../macros/fact.sci')

--> 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 getf 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). 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 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


-->[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



-->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
```

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.

```
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);
```

```
--> 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. 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`	`;` (row separator)
`c`	`[ ]` (matrix definition)
`d`	`./`
`e`	() extraction: `m=a(k)`
`i`	() insertion: `a(k)=m`
`k`	`.*.`
`l`	`\` left division
`m`	`*`
`p`	`^` exponent
`q`	`.\`
`r`	`/` right division
`s`	`-`
`t`	`'` (transpose)
`u`	`*.`
`v`	`/.`
`w`	`\.`
`x`	`.*`
`y`	`./.`
`z`	`.\.`

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
`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),
```

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.

Chapter 4

Basic Primitives

This chapter briefly describes some basic primitives of Scilab. More detailed information is given in the manual (see the directory `SCIDIR/man/LaTex-doc`).

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.

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.

Functions which exist in files can be seen by using `disp` and loaded by using `getf`.

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 command `getf` 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 commands `read` and `write`. These two commands 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.

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. This facility is equivalent to the Unix `whatis` command. 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 file `SCIDIR/man/Chapters` (see also the README file there). You may use the standard format of the scilab manual (see the `SCIDIR/man/subdirectories`). The Scilab LaTeX manual is automatically obtained from the manual items by a `Makefile`. See the directory `SCIDIR/man/Latex-doc`. Note that the command `manedit` opens an help file with an editor (default editor is `emacs`).

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 ôde 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.

Chapter 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. 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` : graphics driver for the X11 window system
- `Rec` : an X Window driver (X11) which also records all the graphic commands. This is the default (required for the zoom and rotate).
- `Wdp` : an X11 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

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 X-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 and Xfig drivers :

- `xinit`: initializes a graphic device simply opens a graphics window for the X-drivers this command is necessary for Postscript and Xfig drivers.
- `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 from the XWindow Manager

  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),-1,'001');
```

- `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([0.,0.,0.6,0.3],[-1,1,-1,1]);
plot2d(sin(t),cos(t));
xsetech([0.5,0.3,0.4,0.6],[-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
```

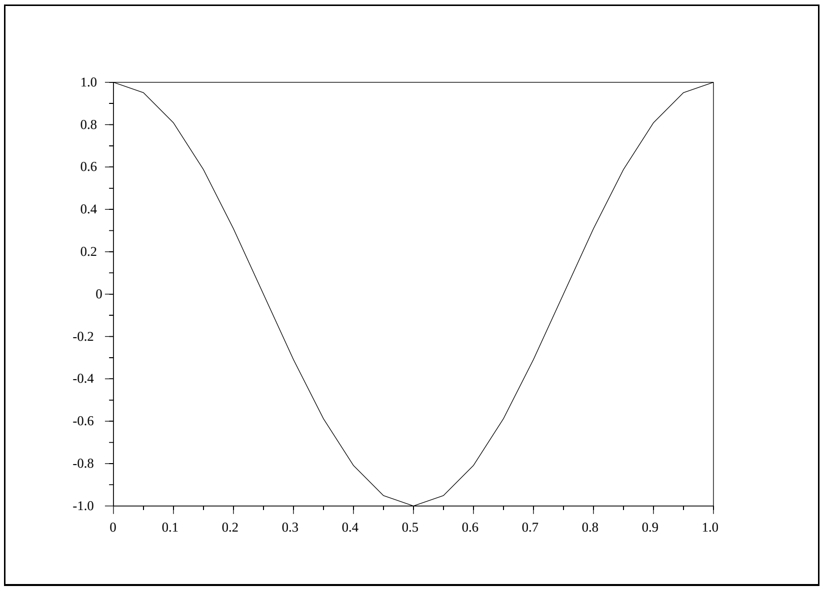


Figure 5.1: First example of plotting

The generic 2D multiple plot is
`plot2di(str,x,y,[style,strf,leg,rect,nax])`

- index of `plot2d` : `i=missing,1,2,3,4`.

For the different values of `i` we have:

`i=missing` : piecewise linear plotting

`i=1` : as previous with possible logarithmic scales

`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);
xsetech([0.,0.,0.5,0.5],[-1,1,-1,1]);
plot2d([t t],[1.5+0.2*sin(t) 2+cos(t)]);
xtitle('Plot2d');
titlepage('Piecewise linear');
//
xsetech([0.5,0.,0.5,0.5],[-1,1,-1,1]);
plot2d1('oll',t,[1.5+0.2*sin(t) 2+cos(t)]);
xtitle('Plot2d1');
titlepage('Logarithmic scale(s)');
//
xsetech([0.,0.5,0.5,0.5],[-1,1,-1,1]);
plot2d2('onn',t,[1.5+0.2*sin(t) 2+cos(t)]);
xtitle('Plot2d2');
titlepage('Piecewise constant');
//
xsetech([0.5,0.5,0.5,0.5],[-1,1,-1,1]);
plot2d3('onn',t,[1.5+0.2*sin(t) 2+cos(t)]);
xtitle('Plot2d3')
titlepage('Vertical bar plot')
xset('default')
```

- Parameter `str` : it is the string `"abc"` :

  `str` is empty if `i` is missing.

  `a=e` : means empty; the values of `x` are not used; (The user must give a dummy value to `x`).

  `a=o` : means one; the x-values are the same for all the curves

  `a=g` : means general.

  `b=l` : a logarithmic scale is used on the X-axis

  `c=l` : a logarithmic scale is used on the Y-axis

  -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.

- Parameter `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).

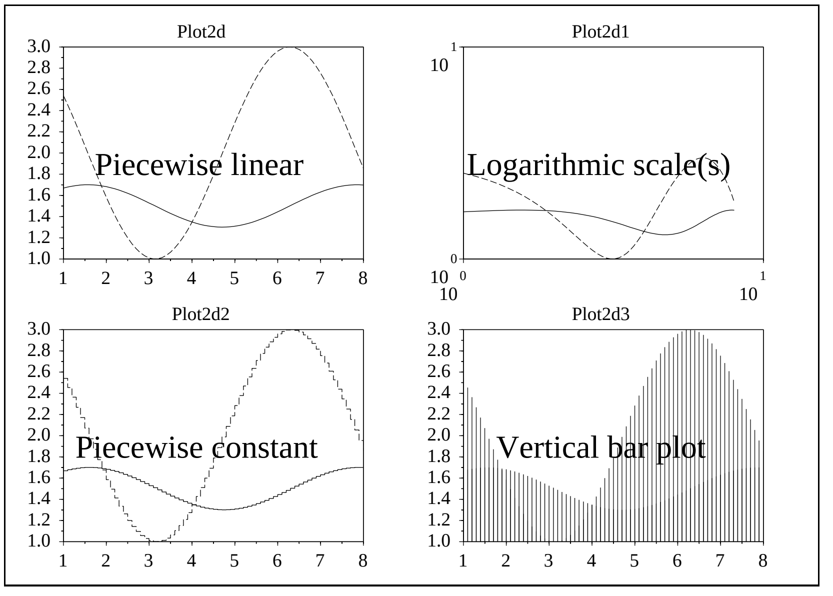


Figure 5.2: Different 2D plotting

```
x=0:0.1:2*%pi;
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 color with the 2 following lines
//sty=[-9,-8,-7,-6,-5,-4,-3,-2,-1,0];
//plot2d1('onn',x',u,sty,"111"," ",[0,-2,2*%pi,3],[2,10,2,10]);
plot2d1('onn',x',u,...
  [9,8,7,6,5,4,3,2,1,0],"011"," ",[0,-2,2*%pi,3],[2,10,2,10]);
x=0:0.2:2*%pi;
v=[1.4+sin(x);1.8+sin(x)]';
xset("mark",1,5);
plot2d1('onn',x',v,[7,8],"011"," ",[0,-2,2*%pi,3],[2,10,2,10]);
xset('default');
```

- Parameter `strf` : it is a string of length 3 `"xyz"` corresponding to :

  `x=1` : captions displayed

  `y=1` : the argument `rect` is used to specify the boundaries of the plot. `rect=[xmin,ymin,xmax,ymax]`

  `y=2` : the boundaries of the plot are computed

  `y=0` : the current boundaries

  `z=1` : an axis is drawn and the number of tics can be specified by the `nax` argument

  `z=2` : the plot is only surrounded by a box

- Parameter `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.

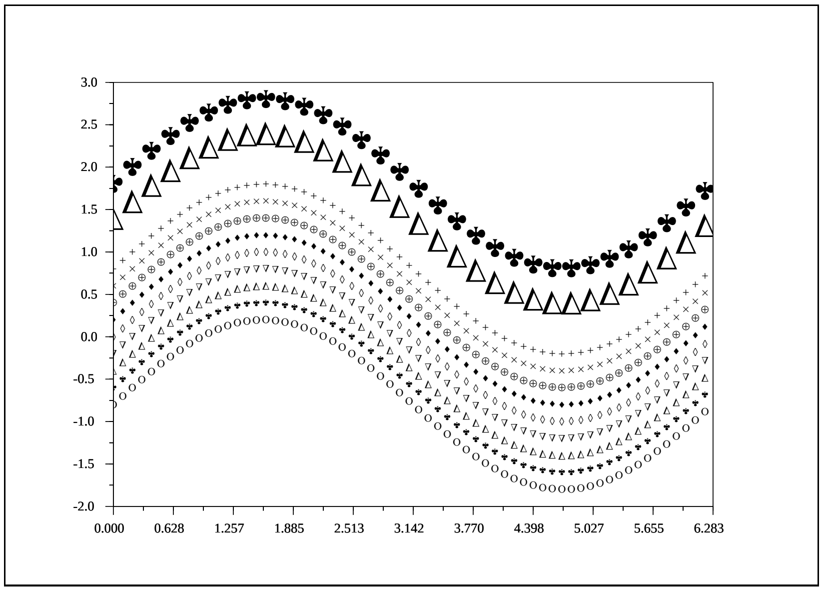


Figure 5.3: Black and white plotting styles

- Parameter `rect` : it is a vector of 4 values specifying the boundaries of the plot `rect=[xmin,ymin,xmax,ymax]`.
- Parameter `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=-%pi:0.3:%pi;
y1=sin(x);y2=cos(x);y3=x;
X=[x;x;x]; Y=[y1;y2;y3];
plot2d1("gnn",X',Y',[1 2 3]',"111","caption1@caption2@caption3",...
[-3,-3,3,2],[2,20,5,5]);
```

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
```

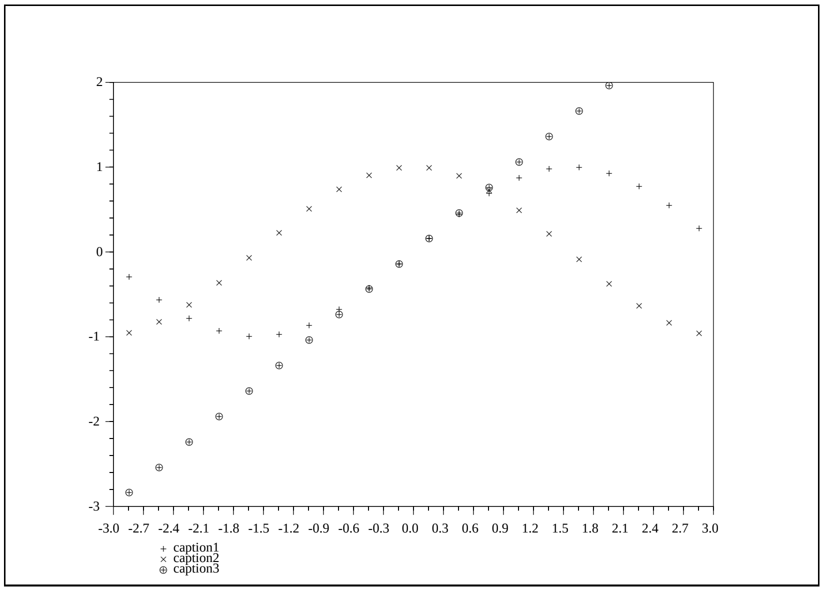


Figure 5.4: Box, captions and tics

```
x=-%pi:0.3:%pi;
y1=sin(x);y2=cos(x);y3=x;
X=[x;x;x]; Y=[y1;y2;y3];
plot2d1("gnn",X',Y',[1 2 3]',"111","caption1@caption2@caption3",...
[-3,-3,3,2],[2,20,2,5]);
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);
```

- `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.

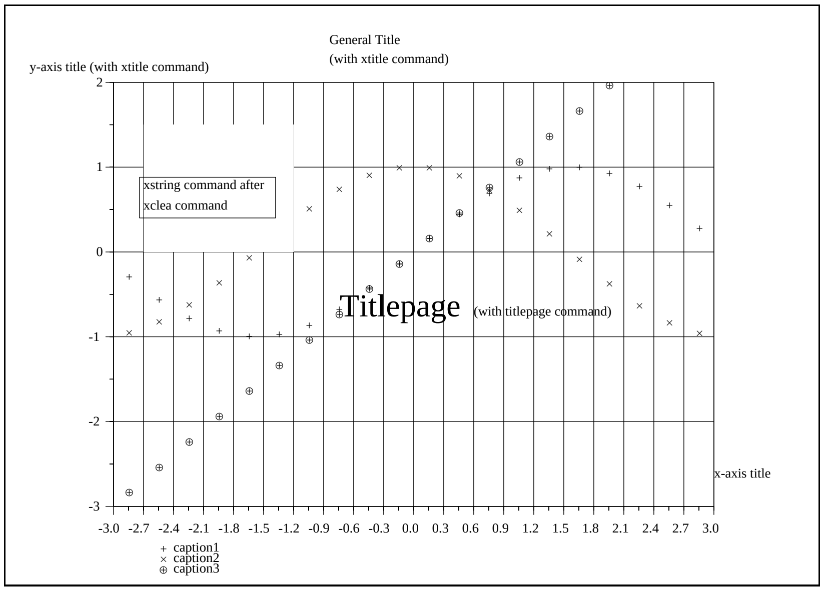


Figure 5.5: Grid, Title eraser and comments

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.

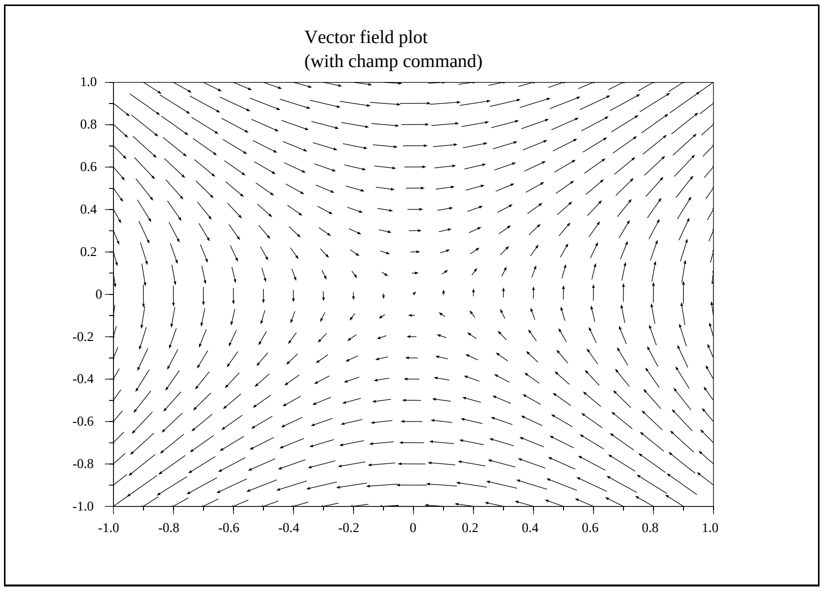


Figure 5.6: Vector field in the plane

```
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,"111",[-2,-2,2,2]);
xset('font',2,3);
xtitle(["Grayplot";"(with fgrayplot command)"]);
//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,"111",[-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

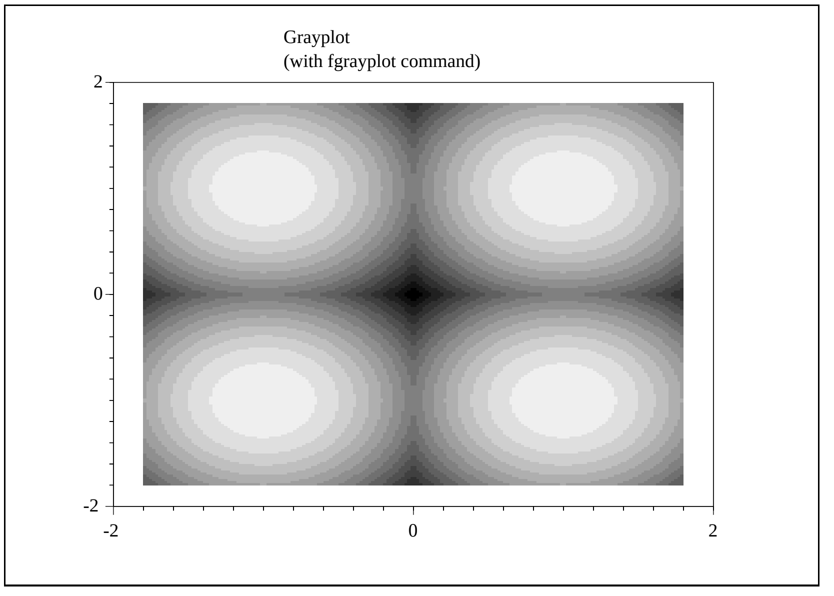


Figure 5.7: Gray plot with a gray colormap

- `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 5.8.

```
// initialize default environment variables
xset('default');
```

```
xset("use color",0);
plot([1:10]);
xbasc()
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.)
xset("font",4,5)
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` :

  (i)-`xset("use color",flag)` changes to color or gray 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`.

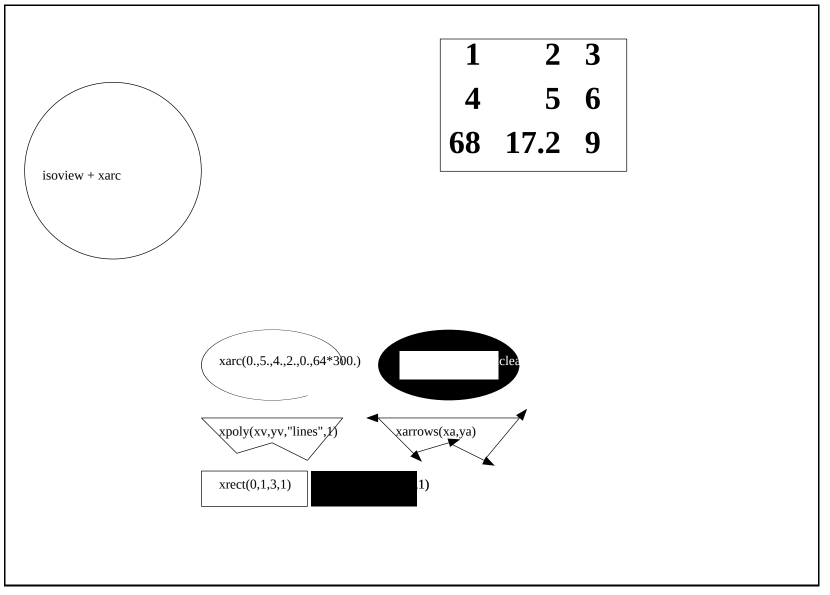


Figure 5.8: Geometric Graphics and Comments

- `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 from the XWindow Manager

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
xsetech([0.,0.,0.5,0.5],[-1,1,-1,1]);
gainplot([h1;h],0.01,100);
//nyquist
xsetech([0.5,0.,0.5,0.5],[-1,1,-1,1]);
```

```
nyquist([h1;h])
//chart and black
xsetech([0.,0.5,0.5,0.5],[-1,1,-1,1]);
black([h1;h],0.01,100,['h1';'h'])
chart([-8 -6 -4],[80 120],list(1,0));
//evans
xsetech([0.5,0.5,0.5,0.5],[-1,1,-1,1]);
H=syslin('c',352*poly(-5,'s')/poly([0,0,2000,200,25,1],'s','c'));
evans(H,100)
```

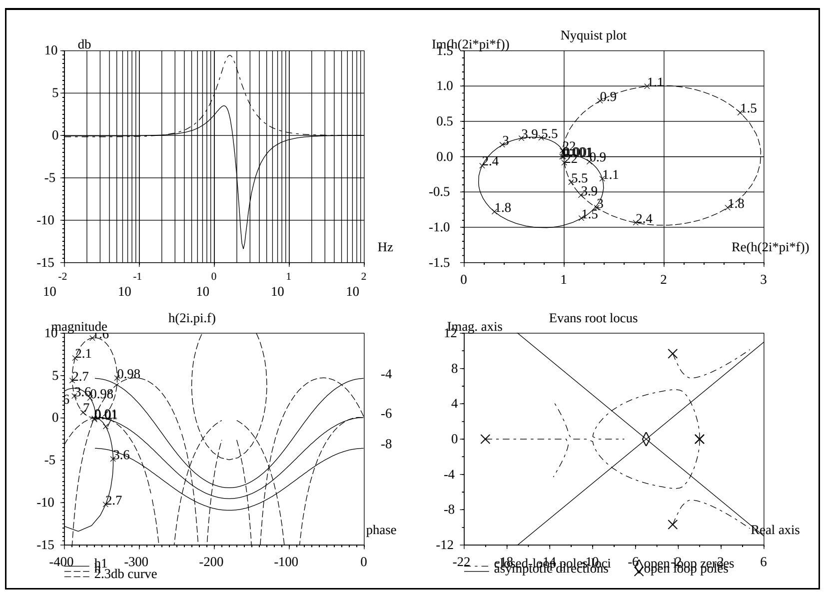


Figure 5.9: Some Plots in Automatic Control

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.

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

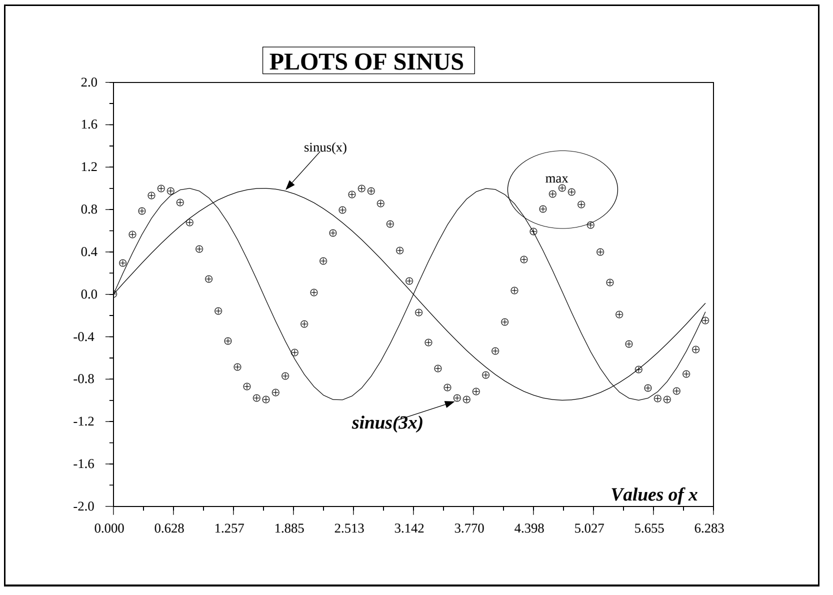


Figure 5.10: Presentation of Plots

- `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 5.11 illustrates this feature.

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

```
deff("[z]=surf(x,y)","z=sin(x)*cos(y)");
t=%pi*(-10:10)/10;
fplot3d(t,t,surf,35,45,"X@Y@Z",[-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)');
```

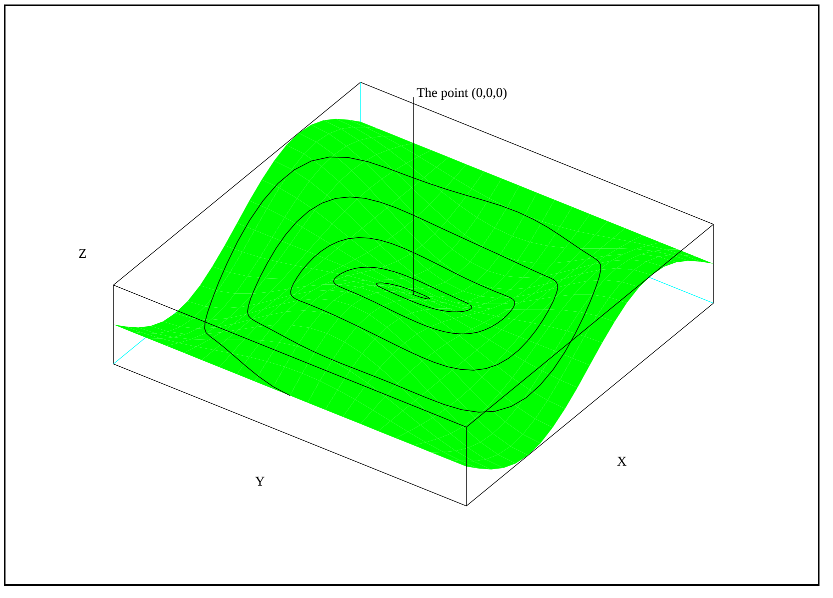


Figure 5.11: 2D and 3D plot

5.5.4 Sub-windows

It is also possible to make multiple plotting in the same graphic window (Figure 5.12).

```
xinit('d7-8.ps');
  t=(0:.05:1)';st=sin(2*%pi*t);
  xsetech([0,0,1,0.5]);
  plot2d2("onn",t,st);
  xsetech([0,0.5,1,0.5]);
  plot2d3("onn",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 5.13 is obtained and inserted in this document with the help of the command `Blatexprs`.

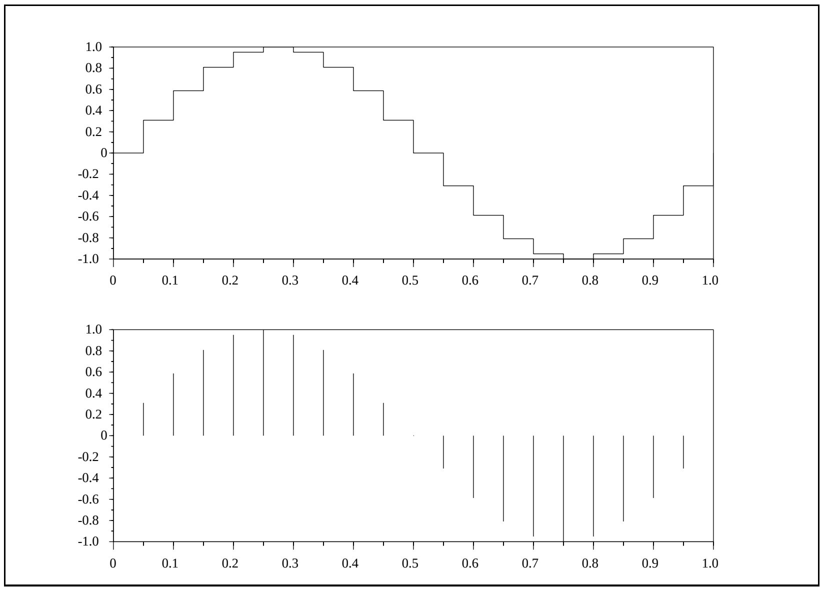


Figure 5.12: Use of xsetech

```
//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
```

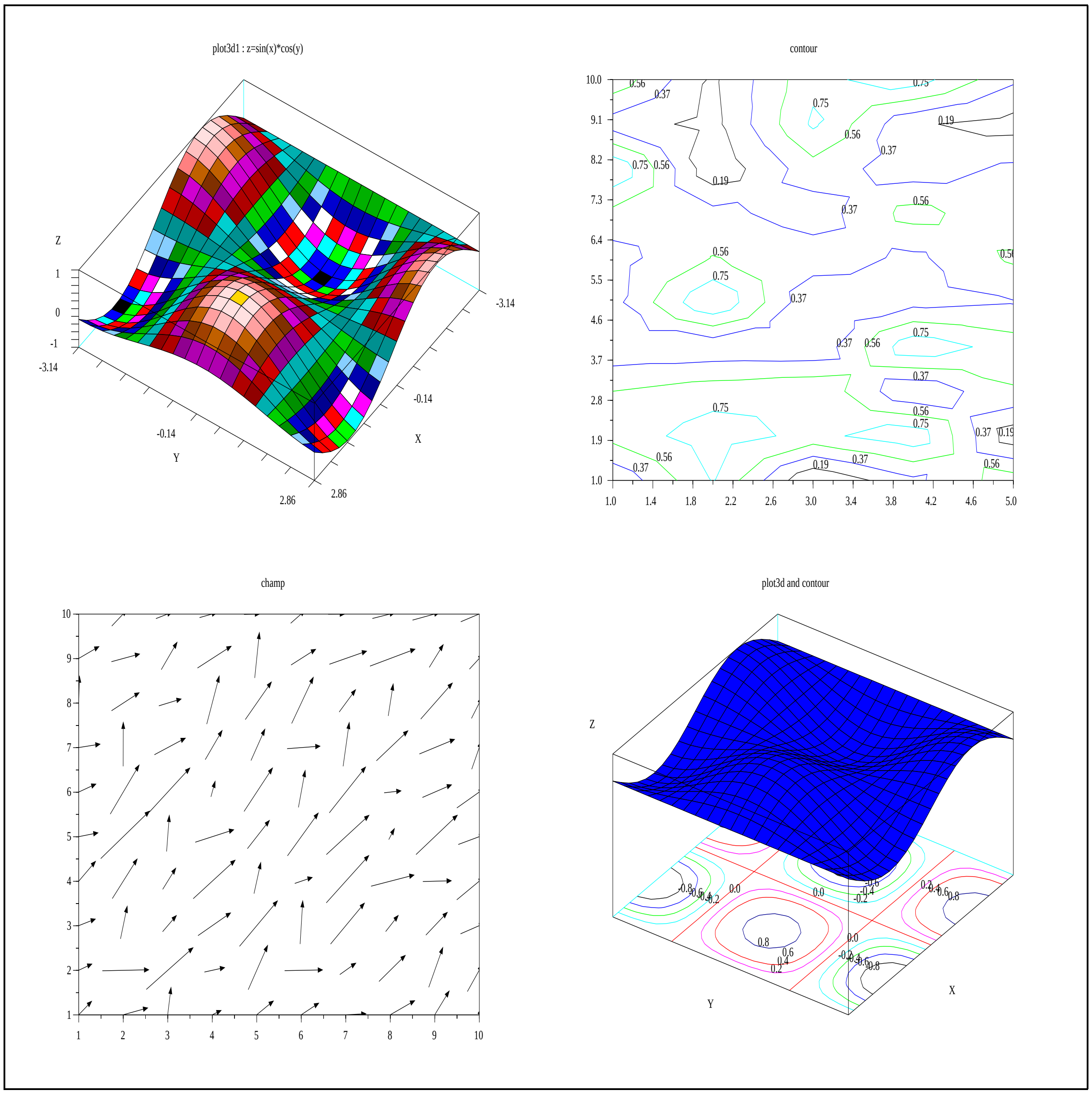


Figure 5.13: Group of figures

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),35,45,'X@Y@Z',[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;

-->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,' ',' ');

-->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}
```

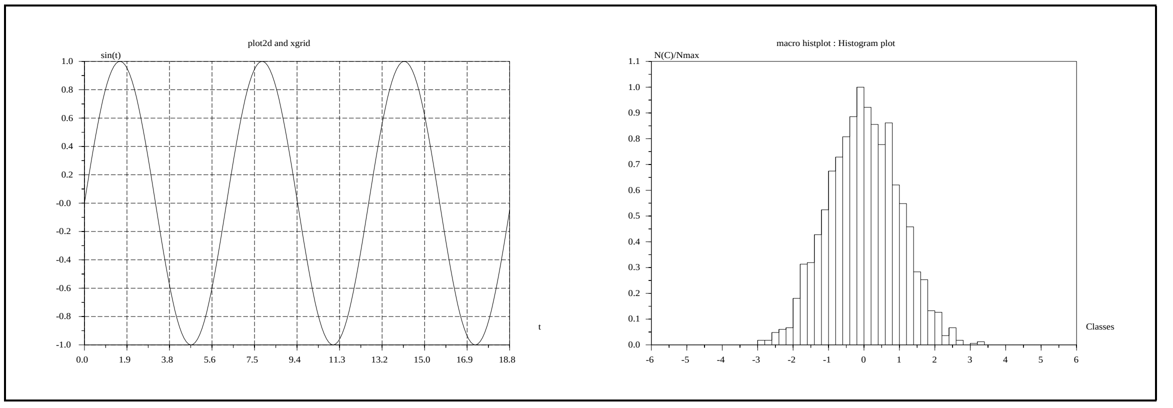


Figure 5.14: Blatexp2 Example

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),35,45,'X@Y@Z',[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.

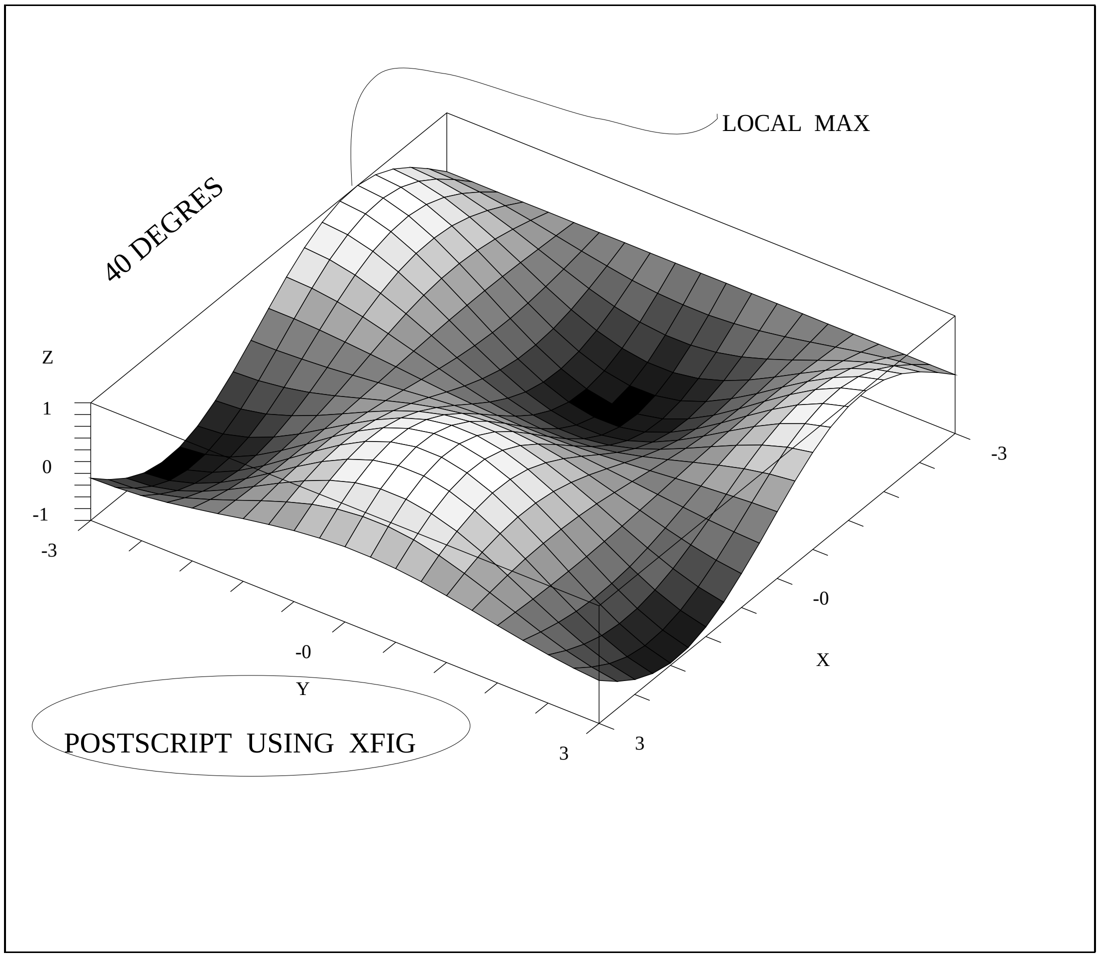


Figure 5.15: Encapsulated Postscript by Using Xfig

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.

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.

Chapter 6

Interfacing C or Fortran programs

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 Scilab `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 `fort` 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 `routines/examples` directory. This interface program is dynamically linked to Scilab by the `addinter` command.

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 made by `intersci` 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/calelm/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 `fort` 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,...]=fort('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 `fort` 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=fort('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 `fort(...,'out',5)` instead of `fort(...,'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=fort('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/addinter-examples`. The two files `template.c` and `template.f` are skeletons of interface programs.

- The file `Examplc.c` is a C interface for the function `foubare2c` which is defined in the file `src/foubare2c.c`. This interface can be tested with the Scilab script `Examplc.sce`.
- The file `Examplf.f` is a Fortran interface for the function `foubare2f` which is defined in the file `src/foubare2f.f`. This interface can be tested with the Scilab script `Examplc.sce`.

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. An interface program defines a set of Scilab functions and the calls to the corresponding numerical programs. Note that a unique interface program can be used to interface an arbitrary (but less that 99) number of functions.
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'` or `'c'` which stands for double, float (real), integer 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 satck 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 (see function `intfce2c` in file `examples/addinter-examples/Testc.c`). The function `FreePtr` should be used to free the pointer.

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 (see example below). Character, integer, real, double type variables are in the `cstk` (resp. `istk, sstk, stk`) Scilab internal stack at the adresses `lk`'s returned by `GetRhsVar` or `CreateVar`.

Then they are returned to Scilab as lhs variables (this is done by function `PutLhsVar`). 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;
PutLhsVar();
```

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.

- `ReadMatrix(aname,m,n,w)`

  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 `w`, the entries of the matrix ordered *columnwise*. `w` is a copy of the Scilab variable called `aname`.

- `ReadString(aname,n,w)`

  This function reads a string in Scilab internal stack. `n` is the length of the string.

- `GetMatrixptr(aname,m,n,l)`

  This function returns the dimensions `m, n` and the address `l` of Scilab variable `aname`.

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

6.2.2 Example

The following interface is taken from the examples in the `examples/addinter-examples` directory. The function to be interfaced has the following calling sequence:

```
int foubare2c (char *ch, int *a, int *ia, float *b, int *ib,
               double *c, int *mc, int *nc, double *d, double *w,
               int *err));
```

The associated Scilab function is:

```
function [y1,y2,y3,y4,y5]=foobar(x1,x2,x3,x4)
```

where `x1` is a character string, and `x2, x3, x4` are matrices which, in the called C function, `foubare2c` are respectively integer, real and double arrays.

The interface program is the following:

```
int intsfoubare(fname)
     char *fname;
{
  int i1, i2;
  static int ierr;
  static int l1, m1, n1, m2, n2, l2, m3, n3, l3, m4, n4, l4, l5, l6;
  static int minlhs=1, minrhs=4, maxlhs=5, maxrhs=4;

  Nbvars = 0;

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

  GetRhsVar(1, "c", &m1, &n1, &l1);
  GetRhsVar(2, "i", &m2, &n2, &l2);
  GetRhsVar(3, "r", &m3, &n3, &l3);
  GetRhsVar(4, "d", &m4, &n4, &l4);

  CreateVar(5, "d", &m4, &n4, &l5);
  CreateVar(6, "d", &m4, &n4, &l6);

  i1 = n2 * m2;
  i2 = n3 * m3;

  foubare2c(cstk(l1), istk(l2), &i1, sstk(l3), &i2, stk(l4),
     &m4, &n4, stk(l5),stk(l6), &ierr);

  if (ierr > 0)
    {
      sciprint("Internal Error");
      Error(999);
      return 0;
    }

  LhsVar(1) = 5;
  LhsVar(2) = 4;
  LhsVar(3) = 3;
  LhsVar(4) = 2;
  LhsVar(5) = 1;
  PutLhsVar();
  return 0;
}

static TabF Tab[]={
  {intsfoubare, "foobar"}
} ;

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

Note that the last part of the interface program should contain in the table `TabF` the pair = (name of the interface program, name of the associated Scilab function). If several functions are interfaced in the interface a pair of names should be given for each function. The entrypoint `foobar` is used by the dynamic link command `addinter`.

6.2.3 `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’])`

<*Scilab function name*> <*function arguments*>
<*Scilab variable*> <*Scilab type*> <*possible arguments*>
⋮ ⋮ ⋮ ⋮ ⋮
<*Fortran subroutine name*> <*subroutine arguments*>
<*Fortran argument*> <*Fortran type*>
⋮ ⋮ ⋮ ⋮
out <*type*> <*formal output names*>
<*formal output name*> <*variable*>
⋮ ⋮ ⋮ ⋮
****

Table 6.1: Description of a pair of Scilab function and Fortran subroutine

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.

In the previous example `addinter` can be called as follows:

```
addinter(['Examplc.o','foubare2c.o'],'foobar','foubare');
```

6.3 Intersci

Intersci is a program for building an interface file between Scilab and Fortran subroutines or C functions. This interface describes both the routine called and the associated Scilab function. The interface is automatically generated from a description file with `.desc` suffix.

6.3.1 Using Intersci

In the following, we will only consider Fortran subroutine interfacing. The process is nearly the same for C functions (see 6.3.1).

To use Intersci execute the command:
`intersci` <*interface name*>
where <*interface name*>`.desc` is the file describing the interface.

The `intersci` script file is located in the directory SCIDIR/bin.

Then the interface file <*interface name*>`.f` is created. A Scilab script file `.sce` is also created. This file, with appropriate changes, can be used to link the interface with Scilab.

The file <*interface name*>`.desc` is a sequence of descriptions of pairs formed by the Scilab function and the corresponding Fortran subroutine (see table 6.1).

Each description is made of three parts:

- description of Scilab function and its arguments
- description of Fortran subroutine and its arguments
- description of the output of Scilab function.

Description of Scilab function The first line of the description is composed by the name of the Scilab function followed by its input arguments.

The next lines describe Scilab variables: the input arguments and the outputs of the Scilab function, together with the arguments of the Fortran subprogram with type `work` (for which memory must be allocated). It is an error not to describe such arguments.

The description of a Scilab variable begins by its name, then its type followed by possible informations depending on the type.

Types of Scilab variables are:

any any type: only used for an input argument of Scilab function.

column column vector: must be followed by its dimension.

list list: must be followed by the name of the list, <*list name*>. This name must correspond to a file <*list name*>`.list` which describes the structure of the list (see 6.3.1).

matrix matrix: must be followed by its two dimensions.

polynom polynomial: must be followed by its dimension (size) and the name of the unknown.

row row vector: must be followed by its dimension.

scalar scalar.

string character string: must be followed by its dimension (length).

vector row or column vector: must be followed by its dimension.

work working array: must be followed by its dimension. It must not correspond to an input argument or to the output of the Scilab function.

A blank line and only one ends this description.

Optional input arguments Optional arguments are defined as follows:

- `[c val]` . This means that `c` is an optional argument with default value `val`. `val` can be a scalar: e.g. `[c 10]`, an array: e.g. `[c (4)/1,2,3,4/]` or a chain: e.g. `[c pipo]`
- `{b xx}`. This means that `b` is an optional argument. If not found, one looks for `xx` in current existing Scialb variables.

Description of Fortran subroutine The first line of the description is composed by the name of the Fortran subroutine followed by its arguments.

The next lines describe Fortran variables: the arguments of the Fortran subroutine.

The description of a Fortran variable is made of its name and its type. Most Fortran variables correspond to Scilab variables (except for dimensions, see 6.3.1) and must have the same name as the corresponding Scilab variable.

Types of Fortran variables are:

char character array.

double double precision variable.

int integer variable.

real real variable.

Other types types also exist, that are called "external" types see 6.3.1.

A blank line and only one ends this description.

Description of the output of Scilab function The first line of this description must begin by the word `out` followed by the type of Scilab output.

Types of output are:

empty the Scilab function returns nothing.

list a Scilab list: must be followed by the names of Scilab variables which form the list.

sequence a Scilab sequence: must be followed by the names of Scilab variables elements of the sequence. This is the usual case.

This first line must be followed by other lines corresponding to output type conversion. This is the case when an output variable is also an input variable with different Scilab type: for instance an input column vector becomes an output row vector. The line which describes this conversion begins by the name of Scilab output variable followed by the name of the corresponding Scilab input variable. See 6.3.1 as an example.

A line beginning with a star "`*`" ends the description of a pair of Scilab function and Fortran subroutine. This line is compulsory even if it is the end of the file. Do not forget to end the file by a carriage return.

Dimensions of non scalar variables When defining non scalar Scilab variables (vectors, matrices, polynomials and character strings) dimensions must be given. There are a few ways to do that:

- It is possible to give the dimension as an integer (see 6.3.1).
- The dimension can be the dimension of an input argument of Scilab function. This dimension is then denoted by a formal name (see 6.3.1).
- The dimension can be defined as an output of the Fortran subroutine. This means that the memory for the corresponding variable is allocated by the Fortran subroutine. The corresponding Fortran variable must necessary have an external type (see 6.3.1 and 6.3.1).

Intersci is not able to treat the case where the dimension is an algebraic expression of other dimensions. A Scilab variable corresponding to this value must defined.

Fortran variables with external type External types are used when the dimension of the Fortran variable is unknown when calling the Fortran subroutine and when its memory size is allocated in this subroutine. This dimension must be an output of the Fortran subroutine. In fact, this will typically happen when we want to interface a C function in which memory is dynamically allocated.

Existing external types:

cchar character string allocated by a C function to be copied into the corresponding Scilab variable.

ccharf the same as `cchar` but the C character string is freed after the copy.

cdouble C double array allocated by a C function to be copied into the corresponding Scilab variable.

cdoublef the same as `cdouble` but the C double array is freed after the copy.

cint C integer array allocated by a C function to be copied into the corresponding Scilab variable.

cintf the same as `cint` but the C integer array is freed after the copy.

In fact, the name of an external type corresponds to the name of a C function. This C function has three arguments: the dimension of the variable, an input pointer and an output pointer.

For instance, below is the code for external type `cintf`:

```
#include "../machine.h"

/* ip is a pointer to a Fortran variable coming from SCILAB
which is itself a pointer to an array of n integers typically
coming from a C function
   cintf converts this integer array into a double array in op
   moreover, pointer ip is freed */

void C2F(cintf)(n,ip,op)
int *n;
int *ip[];
double *op;
{
  int i;
  for (i = 0; i < *n; i++)
    op[i]=(double)(*ip)[i];
  free((char *)(*ip));
}
```

For the meaning of `#include "../machine.h"` and `C2F` see 6.3.1.

Then, the user can create its own external types by creating its own C functions with the same arguments. Intersci will generate the call of the function.

<comment on the variable element of the list>
<name of the variable element of list> <type> <possible arguments>
****

Table 6.2: Description of a variable element of a list

Using lists as input Scilab variables An input argument of the Scilab function can be a Scilab list. If *<list name>* is the name of this variable, a file called *<list name>*`.list` must describe the structure of the list. This file permits to associate a Scilab variable to each element of the list by defining its name and its Scilab type. The variables are described in order into the file as described by table 6.2.

Then, such a variable element of the list, in the file *<interface name>*`.desc` is referred to as its name followed by the name of the corresponding list in parenthesis. For instance, `la1(g)` denotes the variable named `la1` element of the list named `g`.

An example is shown in 6.3.1.

C functions interfacing

The C function must be considered as a procedure i.e. its type must be `void` or the returned value must not be used.

The arguments of the C function must be considered as Fortran arguments i.e. they must be only pointers.

Moreover, the name of the C function must be recognized by Fortran. For that, the include file `machine.h` located in the directory *<Scilab directory>*`/routines` should be included in C functions and the macro `C2F` should be used.

Writing compatible code

Messages and Error Messages To write messages in the Scilab main window, user must call the `out` Fortran routine or `cout` C procedure with the character string of the desired message as input argument.

To return an error flag of an interfaced routine user must call the `erro` Fortran routine or `cerro` C procedure with the character string of the desired message as input argument. This call will produce the edition of the message in the Scilab main window and the error exit of Scilab associated function.

Input and output

To open files in Fortran, it is highly recommended to use the Scilabroutine `clunit`. If the interfaced routine uses the Fortran `open` instruction, logical units must in any case be greater than 40.

```
call clunit( lunit, file, mode)
```

with:

- `file` the file name `character string`
- `mode` a two integer vector defining the opening mode `mode(2)` defines the record length for a direct access file if positive. `mode(1)` is an integer formed with three digits `f`, `a` and `s`

- `f` defines if file is formatted (`0`) or not (`1`)
- `a` defines if file has sequential (`0`) or direct access (`1`)
- `s` defines if file status must be new (`0`), old (`1`), scratch (`2`) or unknown (`3`)

Files opened by a call to `clunit` must be close by

```
call clunit( -lunit, file, mode)
```

In this case the `file` and `mode` arguments are not referenced.

Examples

Example 1 The Scilab function is `a=calc(str)`. Its input is a string and its output is a scalar.

The corresponding Fortran subroutine is `subroutine fcalc(str,a)`. Its arguments are a string `str` (used as input) and an integer `a` (used as output).

We reserve a fixed dimension of 10 for the string.

The description file is the following:

```
calc    str
str     string  10
a       scalar

fcalc   str     a
str     char
a       integer

out     a
***********************
```

Example 2 The name of the Scilab function is `c=som(a,b)`. Its two inputs are row vectors and its output is a column vector.

The corresponding Fortran subroutine is `subroutine fsom(a,n,b,m,c)`. Its arguments are a real array with dimension n (used as input), another real array with dimension m (used as input) and a real array (used as output). These dimensions `m` and `n` are determined at the calling of the Scilab function and do not need to appear as Scilab variables.

Intersci will do the job to make the necessary conversions to transform the double precision (default in Scilab) row vector `a` into a real array and to transform the real array `c` into a double precision row vector.

The description file is the following:

```
som     a       b
a       row     m
b       row     n
c       column  n

fsom    a       n       b       m       c
a       real
n       integer
```

```
b       real
m       integer
c       real

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

Example 3 The Scilab function is `[o,b]=ext(a)`. Its input is a matrix and its outputs are a matrix and a column vector.

The corresponding Fortran subroutine is `fext(a,m,n,b,p)` and its arguments are an integer array (used as input and output), its dimensions m,n (used as input) and another integer array and its dimension p (used as outputs).

The dimension `p` of the output `b` is computed by the Fortran subroutine and the memory for this variable is also allocated by the Fortran subroutine (perhaps by to a call to another C function). So the type of the variable is external and we choose `cintf`.

Moreover, the output `a` of the Scilab function is the same as the input but its type changes from a $m \times n$ matrix to a $n \times m$ matrix. This conversion is made my introducing the Scilab variable `o`

The description file is the following:

```
ext     a
a       matrix  m       n
b       column  p
o       matrix  n       m

fext    a       m       n       b       p
a       integer
m       integer
n       integer
b       cintf
p       integer

out     sequence        o       b
o       a
***************************
```

Example 4 The name of the Scilab function is `contr`. Its input is a list representing a linear system given by its state representation and a tolerance. Its return is a scalar (for instance the dimension of the controllable subspace).

The name of the corresponding Fortran subroutine is `contr` and its arguments are the dimension of the state of the system (used as input), the number of inputs of the system (used as input), the state matrix of the system (used as input), the input matrix of the system (used as input), an integer giving the dimension of the controllable subspace (used as output), and the tolerance (used as input).

The description file is the following:

```
contr   sys     tol
tol     scalar
sys     list    lss
```

```
icontr  scalar

contr   nstate(sys)     nin(sys)     a(sys)  b(sys)  icontr  tol
a(sys)  double
b(sys)  double
tol     double
nstate(sys)     integer
nin(sys)        integer
icontr  integer

out     sequence        icontr
******************************
```

The type of the list is `lss` and a file describing the list `lss.list` is needed. It is shown below:

```
1 type
type    string  3
******************************
2 state matrix
a       matrix  nstate  nstate
******************************
3 input matrix
b       matrix  nstate  nin
******************************
4 output matrix
c       matrix  nout    nstate
******************************
5 direct tranfer matrix
d       matrix  nout    nin
******************************
6 initial state
x0      column  nstate
******************************
7 time domain
t       any
******************************
```

The number of the elements is not compulsory in the comment describing the elements of the list but is useful.

Adding a new primitive

It is possible to add a set a new built-in functions to Scilab by a permanent link the interface program. For that, it is necessary to update the files `default/fundef` and `routines/callinter.h`.

When `intersci` is invoked as follows:

`intersci` *<interface name>* *<interface number>*

intersci then builds a .fundef file which is used to update the default/fundef file.

To add a new interface the user needs also to update the routines/callinter.h file with a particular value of fun Fortran variable corresponding to the new interface number.

Two unused empty interface routines called by default (matusr.f and matus2.f) are predefined and may be replaced by the interface program. Their interface numbers 14 and 24 respectively. They can be used as default interface programs. The executable code of Scilab is then made by typing "make all" or "make bin/scilex" in Scilab directory.

6.4 The routines/default directory

The SCIDIR/routines/default directory contains a set of C and Fortran routines which can be customized by the user. When customizing a routine in this directory a new executable code for Scilab is made by typing make all in the main Scilab directory. It is possible to add new primitives by modifying the default files given in this directory. The file Ex-fort.f contains a example of a subroutine (bidon2) which can be interactively called by the Scilab fort command. Thus, it is possible to call a C or Fortran routine by modifying the Ex-fort.f file, re-making Scilab and then using the fort function. The link operation now made outside Scilab by the make all command which creates a full new executable code for Scilab (SCIDIR/bin/scilex).

Let us consider again the example of the daxpy function. We want to call it from Scilab by the following function

```
function y=scilabdaxpy(a,x,incx,y,incy)
y=fort('daxpy1',a,x,incx,y,incy)
```

which performs the following:

y(1:incy:n*incy)=y(1:incy:n*incy)+a*x(1:incx:n*incx)

The fort function looks for the called program (here daxpy1) in the interface file default/Ex-fort.f: for that, it is necessary that the name daxpy1 appear in the file default/Flist. (We do not use the link command here).

Note that the fort function just sends the Scilab variables a,x,incx,y,incy to the interface program Ex-fort.f. These variables are associated with the numbers 1,2,3,4,5,6, respectively in the interface program Ex-fort.f. For our scilabdaxpy function, we perform the following steps:

- Add the name daxpy1 to the appropriate list of functions in the file Flist in the routines/default directory:

  ```
  interf_list= ... daxpy1
  ```

- Edit the file routines/default/Ex-fort.f and insert the following code:

  ```
   subroutine daxpy1()
   include '../stack.h'

   n=msize(2,mx,nx)

   call alloc(1,1,1,1,'d')
   call alloc(2,n,mx,nx,'d')
  ```

```
      call alloc(3,1,1,1,'i')
      call alloc(4,n,mx,nx,'d')
      call alloc(5,1,1,1,'i')

      call daxpy(n,stk(ladr(1)),stk(ladr(2)),stk(ladr(3)),
     +             stk(ladr(4)),stk(ladr(5)))
      call back(4)
      return
      end
```

The interface is done using the functions `msize`, `alloc` and `back`. When the command `fort('daxpy1',a,x,incx,y,incy)` is issued, each variable `a,x,incx,y,incy` is automatically assigned a number in `Ex-fort`, in increasing order. Here `a` is assigned number 1, `x` is assigned number 2, etc. Variable # `n` is located in Scilab internal stack `stk` at adress `ladr(n)`. For instance, `x`, (the third variable in `daxpy` calling sequence), is associated with the pointer `ladr(2)` in `stk` since `x` is variable # 2.

The statement `n=msize(2,mx,nx)` retrieves the dimensions `mx, nx` of variable # 2 i.e. `x` and `n=mx*mx` i.e. n=number of rows $\times$ number of columns.

This function allows to know the dimensions of all the variables passed to `fort`. At this stage, the user can test that the dimensions of the variables are correct; the corresponding error message can be done as follows:

```
buf='error message'
call error(9999)
return
```

The function `alloc` defines the type of a variable (integer, real, double), sets its dimensions and allocate memory for it. For instance `call alloc(4,mx*nx,mx,nx,'d')` is used to define the fourth variable (`y`) as a matrix with `mx` rows and `nx` columns of type "double". The last parameter of `alloc` should be `'i'` for integer, `'r'` for real or `'d'` for double.

When `alloc` is called with a number `n` (as first parameter) which does not correspond to a input of `fort`, a valid new adress (pointer) `ladr(n)` is automatically set. For instance the statement `call alloc(6,12,4,3,'i')` will return in `ladr(6)` a pointer for a 6th matrix variable (not in the parameters of `fort`) with dimensions 3×4 and integer type.

Note that the default type for variables is `'d'` i.e. double. For such variables, the call to `alloc` can be omitted: in our example, only the statements `call alloc(3,...,'i')` and `call alloc(5,...,'i')` which convert `incx` and `incy` to integers are necessary. However, the call to `alloc` is always necessary for defining a variable which does not appear in the `fort` parameters.

After the call to `daxpy`, the function `back(i)` returns variable number `i` to Scilab. This variable has the dimensions set by the previous call to `alloc` and is converted into a Scilab matrix.

6.4.1 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 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`.

6.6 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.

6.6.1 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.
```

6.6.2 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=fort('f_m',x,1,'d',a,2,'d',b,3,'d','out',[1,1],4,'d')

function [var]=fmat(x,par)
var=fort('fmat',x,1,'d',par,2,'d','out',[2,2],3,'d')
```

Appendix A

Demo session

We give here the Scilab session corresponding to the first demo.

```
-->//               SCILAB OBJECTS

-->//               1. SCALARS

-->a=1               //real constant
 a  =

    1.

-->1==1              //boolean
 ans  =

  T

-->'string'          //character string
 ans  =

 string

-->z=poly(0,'z')     // polynomial with variable 'z' and with one root at zero
 z  =

    z

-->p=1+3*z+4.5*z^2   //polynomial
 p  =

                 2
    1 + 3z + 4.5z

-->r=z/p             //rational
 r  =
```

```
          z
    -------------
                 2
    1 + 3z + 4.5z


-->//               2. MATRICES

-->A=[a+1 2 3
-->     0 0 atan(1)
-->     5 9 -1]      //3 x 3 constant matrix
 A  =

!   2.    2.    3.        !
!   0.    0.    0.7853982 !
!   5.    9.  - 1.        !


-->b=[%t,%f]         //1 x 2 boolean matrix
 b  =

! T F !


-->Mc=['this','is';
-->    'a' ,'matrix']   //2 x 2 matrix of strings
 Mc  =

!this  is      !
!              !
!a     matrix  !


-->Mp=[p,1-z;
-->    1,z*p]        //2 x 2 polynomial matrix
 Mp  =

!                2                  !
!   1 + 3z + 4.5z     1 - z         !
!                                   !
!                           2      3 !
!   1                 z + 3z + 4.5z  !


-->F=Mp/poly([1+%i 1-%i 1],'z')   //rational matrix
 F  =

!                 2                      !
```

```
!    1 + 3z + 4.5z        - 1            !
!   ---------------     ---------        !
!              2   3              2      !
! - 2 + 4z - 3z + z     2 - 2z + z       !
!                                        !
!                              2      3  !
!          1             z + 3z + 4.5z   !
!   ---------------     ---------------  !
!              2   3               2   3 !
! - 2 + 4z - 3z + z   - 2 + 4z - 3z + z  !


-->Sp=sparse([1,2;4,5;3,10],[1,2,3])   //sparse matrix
 Sp  =

(    4,   10) sparse matrix

(    1,    2)        1.
(    3,   10)        3.
(    4,    5)        2.


-->Sp(1,10)==Sp(1,1)                   //boolean sparse matrix
 ans  =

(    1,    1) sparse matrix

(    1,    1)    T


-->//                 3. LISTS

-->L=list(a,-(1:5), Mp,['this','is';'a','list'])   //list
 L  =


       L(1)

    1.

       L(2)

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

       L(3)

!                2                   !
!   1 + 3z + 4.5z     1 - z          !
```

```
!                                    !
!                           2      3 !
!   1                 z + 3z + 4.5z  !

       L(4)

!this  is    !
!            !
!a     list  !

-->L(2)(3)     //sub-entry in list
 ans  =

  - 3.

-->Lt=tlist(['mylist','color','position','weight'],'blue',[0,1],10)  //typed-list
 Lt  =


       Lt(1)

!mylist  color  position  weight  !

       Lt(2)

 blue

       Lt(3)

!   0.    1. !

       Lt(4)

    10.

-->Lt('color')      //extracting
 ans  =

 blue

-->Lt('weight')     //extracting
 ans  =

    10.

-->A=diag([2,3,4]);B=[1 0;0 1;0 0];C=[1 -1 0];D=0*C*B;x0=[0;0;0];

-->Sl=syslin('c',A,B,C,D,x0)    //Standard state-space linear system
```

```
 Sl  =


       Sl(1)   (state-space system:)

!lss  A  B  C  D  X0  dt  !

       Sl(2) = A matrix =

!   2.    0.    0. !
!   0.    3.    0. !
!   0.    0.    4. !

       Sl(3) = B matrix =

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

       Sl(4) = C matrix =

!   1.  - 1.    0. !

       Sl(5) = D matrix =

!   0.    0. !

       Sl(6) = X0 (initial state) =

!   0. !
!   0. !
!   0. !

       Sl(7) = Time domain =

 c

-->Sl("A"), Sl("C")             //Retrieving elements of a typed list
 ans  =

!   2.    0.    0. !
!   0.    3.    0. !
!   0.    0.    4. !
 ans  =

!   1.  - 1.    0. !

-->Slt=ss2tf(Sl)                // Transfer matrix
```

```
 Slt  =

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

-->Slt('num'), Slt('den')
 ans  =

!   1   - 1  !
 ans  =

! - 2 + s   - 3 + s  !

-->//                  OPERATIONS

-->v=1:5;W=v'*v                 //constant matrix multiplication
 W  =

!   1.    2.     3.     4.     5.  !
!   2.    4.     6.     8.     10. !
!   3.    6.     9.     12.    15. !
!   4.    8.     12.    16.    20. !
!   5.    10.    15.    20.    25. !

-->W(1,:)                       //extracting first row
 ans  =

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

-->W(:,$)                       //extracting last column
 ans  =

!   5.  !
!   10. !
!   15. !
!   20. !
!   25. !

-->Mp'*Mp+eye()                 //polynomial matrix
 ans  =


         column 1

!               2      3        4 !
!   3 + 6z + 18z + 27z + 20.25z  !
!                                !
```

```
!                2               !
!   1 + 3z + 4.5z                !

         column 2

!                2                         !
!   1 + 3z + 4.5z                          !
!                                          !
!              2    3     4     5        6 !
!   2 - 2z + 2z + 6z + 18z + 27z + 20.25z  !

-->Mp1=Mp(1,1)+4.5*%i           //complex
 Mp1  =

real part



                2
    1 + 3z + 4.5z
imaginary part



    4.5

-->Fi=C*(z*eye()-A)^(-1)*B;     //transfer function evaluation

-->F(:,1)*Fi                    //operations with rationals
 ans  =

!                    2                          2      !
!       1 + 3z + 4.5z            - 1 - 3z - 4.5z       !
!   ---------------------     ---------------------    !
!                2    3   4                 2    3   4 !
!   4 - 10z + 10z - 5z + z     6 - 14z + 13z - 6z + z  !
!                                                      !
!             1                       - 1              !
!   ---------------------     ---------------------    !
!                2    3   4                 2    3   4 !
!   4 - 10z + 10z - 5z + z     6 - 14z + 13z - 6z + z  !

-->M=[Mp -Mp; Mp' Mp+eye()]     //concatenation of polynomial matrices
 M  =


         column 1 to 3

!                2                               2 !
!   1 + 3z + 4.5z     1 - z           - 1 - 3z - 4.5z  !
```

```
!                                                        !
!                            2      3                    !
!   1                  z + 3z + 4.5z   - 1               !
!                                                        !
!                2                                     2 !
!   1 + 3z + 4.5z     1                  2 + 3z + 4.5z   !
!                                                        !
!                            2      3                    !
!   1 - z              z + 3z + 4.5z     1               !

         column 4

! - 1 + z               !
!                       !
!          2      3     !
! - z - 3z - 4.5z       !
!                       !
!   1 - z               !
!                       !
!              2      3 !
!   1 + z + 3z + 4.5z   !

-->[Fi, Fi(:,1)]                // ... or rationals
 ans  =

!     1       - 1         1    !
!   -----     -----     -----  !
! - 2 + z   - 3 + z   - 2 + z  !

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

-->Num=F('num');Den=F('den');           //operation on transfer matrix


-->//                  SOME NUMERICAL PRIMITIVES

-->inv(A)                       //Inverse
 ans  =

!   0.5    0.           0.   !
!   0.     0.3333333    0.   !
!   0.     0.           0.25 !

-->inv(Mp)                      //Inverse
 ans  =


         column 1
```

```
!                  2      3           !
!            z + 3z + 4.5z            !
!   ------------------------------    !
!              2     3     4        5 !
! - 1 + 2z + 6z + 18z + 27z + 20.25z  !
!                                     !
!                - 1                  !
!   ------------------------------    !
!              2     3     4        5 !
! - 1 + 2z + 6z + 18z + 27z + 20.25z  !

         column 2

!              - 1 + z                !
!   ------------------------------    !
!              2     3     4        5 !
! - 1 + 2z + 6z + 18z + 27z + 20.25z  !
!                                     !
!                         2           !
!            1 + 3z + 4.5z            !
!   ------------------------------    !
!              2     3     4        5 !
! - 1 + 2z + 6z + 18z + 27z + 20.25z  !

-->inv(Sl*Sl')                  //Product of two linear systems and inverse
 ans  =


       ans(1)   (state-space system:)

!lss  A  B  C  D  X0  dt  !

       ans(2) = A matrix =

!   2.8641369  - 0.9304438    0.    0. !
!   0.4111970    2.1358631    0.    0. !
!   0.         - 9.339D-16    4.    0. !
!   0.           0.           0.    4. !

       ans(3) = B matrix =

!   0.7027925 !
!   0.3620534 !
!   1.665D-16 !
!   0.        !

       ans(4) = C matrix =
```

```
! - 0.3238304    0.6285968    1.890D-15    0. !

       ans(5) = D matrix =

                      2
    2.75 - 2.5s + 0.5s

       ans(6) = X0 (initial state) =

!   0. !
!   0. !
!   0. !
!   0. !

       ans(7) = Time domain =

 c

-->w=ss2tf(ans)                 //Transfer function representation
 w  =

                    2    3      4
    18 - 30s + 18.5s - 5s + 0.5s
    ----------------------------
                        2
            6.5 - 5s + s

-->w1=inv(ss2tf(Sl)*ss2tf(Sl'))    //Product of two transfer functions and inverse
 w1  =

                  2     3   4
    36 - 60s + 37s - 10s + s
    ------------------------
                      2
         13 - 10s + 2s

-->clean(w-w1)
 ans  =

    1.730D-09 - 6.605D-10s
    ----------------------
                     2
         6.5 - 5s + s

-->A=rand(3,3);;B=rand(3,1);n=contr(A,B)                 //Controllability
 n  =
```

```
    3.

-->K=ppol(A,B,[-1-%i -1+%i -1])        //Pole placement
 K  =

!   7.1638394    7.2295307    0.3176982 !

-->poly(A-B*K,'z')-poly([-1-%i -1+%i -1],'z')    //Check...
 ans  =

                                      2
  - 8.882D-16 + 1.776D-15z - 1.332D-15z


-->s=sin(0:0.1:5*%pi);

-->ss=fft(s(1:128),-1);        //FFT

-->xbasc();

-->plot2d3("enn",1,abs(ss)');      //simple plot



-->//              ON LINE DEFINITION OF FUNCTION

-->deff('[x]=fact(n)','if n=0 then x=1,else x=n*fact(n-1),end')
 Warning: obsolete use of = instead of ==
if n=0 then x=1,else x=n*fact(n-1),end
      !
at line       2 of function fact                     called by :
deff('[x]=fact(n)','if n=0 then x=1,else x=n*fact(n-1),end')



-->10+fact(5)
 ans  =

    130.

-->//                    OPTIMIZATION

-->deff('[f,g,ind]=rosenbro(x,ind)', 'a=x(2)-x(1)^2 , b=1-x(2) ,...
-->f=100.*a^2 + b^2 , g(1)=-400.*x(1)*a , g(2)=200.*a -2.*b ');

-->[f,x,g]=optim(rosenbro,[2;2],'qn')
 g  =
```

```
!   0. !
!   0. !
 x  =

!   1. !
!   1. !
 f  =

    0.


-->//                   SIMULATION

-->a=rand(3,3)
 a  =

!   0.7263507    0.2320748    0.8833888 !
!   0.1985144    0.2312237    0.6525135 !
!   0.5442573    0.2164633    0.3076091 !

-->e=expm(a)
 e  =

!   2.6034702    0.5788017    1.7895052 !
!   0.6508072    1.4242213    1.1108378 !
!   1.0616116    0.4238856    1.8909166 !

-->deff('[ydot]=f(t,y)','ydot=a*y');

-->e(:,1)-ode([1;0;0],0,1,f)
 ans  =

! - 1.425D-07 !
! - 7.368D-08 !
! - 8.683D-08 !


-->//                  SYSTEM DEFINITION

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

-->h=[1/s,1/(s+1);1/s/(s+1),1/(s+2)/(s+2)]
 h  =

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

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

-->w=tf2ss(h);

-->ss2tf(w)
 ans  =

!        1                 1            !
!   -------------         -----         !
! - 4.710D-16 + s         1 + s         !
!                                       !
!    1 + 6.935D-16s       1 + 2.448D-16s  !
!   ----------------      --------------  !
!                   2                2    !
! - 1.610D-15 + s + s       4 + 4s + s    !

-->h1=clean(ans)
 h1  =

!   1          1         !
!   -         -----      !
!   s         1 + s      !
!                        !
!     1           1      !
!   -----     ---------  !
!       2               2 !
!   s + s     4 + 4s + s  !


-->//             EXAMPLE: SECOND ORDER SYSTEM ANALYSIS

-->sl=syslin('c',1/(s*s+0.2*s+1))
 sl  =

         1
    -----------
               2
    1 + 0.2s + s

-->instants=0:0.05:20;

-->//             step response:

-->y=csim('step',instants,sl);
```

```
-->xbasc();plot2d(instants',y')

-->//             Delayed step response

-->deff('[in]=u(t)','if t<3 then in=0;else in=1;end');

-->y1=csim(u,instants,sl);plot2d(instants',y1');


-->//             Impulse response;

-->yi=csim('imp',instants,sl);xbasc();plot2d(instants',yi');

-->yi1=csim('step',instants,s*sl);plot2d(instants',yi1');


-->//              Discretization

-->dt=0.05;

-->sld=dscr(tf2ss(sl),0.05);


-->//               Step response

-->u=ones(instants);
Warning :redefining function: u


-->yyy=flts(u,sld);

-->xbasc();plot(instants,yyy)


-->//              Impulse response

-->u=0*ones(instants);u(1)=1/dt;

-->yy=flts(u,sld);

-->xbasc();plot(instants,yy)


-->//            system interconnexion

-->w1=[w,w];

-->clean(ss2tf(w1))
```

```
 ans  =

!   1          1            1          1         !
!   -        -----          -        -----       !
!   s        1 + s          s        1 + s       !
!                                                !
!     1          1            1          1       !
!   -----    ---------      -----    ---------   !
!        2            2          2            2  !
!   s + s    4 + 4s + s     s + s    4 + 4s + s  !

-->w2=[w;w];

-->clean(ss2tf(w2))
 ans  =

!   1          1         !
!   -        -----       !
!   s        1 + s       !
!                        !
!     1          1       !
!   -----    ---------   !
!        2            2  !
!   s + s    4 + 4s + s  !
!                        !
!   1          1         !
!   -        -----       !
!   s        1 + s       !
!                        !
!     1          1       !
!   -----    ---------   !
!        2            2  !
!   s + s    4 + 4s + s  !


-->//              change of variable

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

-->horner(h,(1-z)/(1+z))  //bilinear transform
 ans  =

!   1 + z          1 + z       !
!   -----          -----       !
!   1 - z            2         !
!                              !
!            2              2  !
!   1 + 2z + z     1 + 2z + z  !
```

```
!   ----------     ----------  !
!                           2 !
!     2 - 2z       9 + 6z + z  !


-->//                 PRIMITIVES

-->H=[1.    1.    1.    0.;
-->   2.   -1.    0.    1;
-->   1.    0.    1.    1.;
-->   0.    1.    2.   -1];


-->ww=spec(H)
 ww  =

!   2.7320508 !
! - 2.7320508 !
!   0.7320508 !
! - 0.7320508 !


-->//             STABLE SUBSPACES

-->[X,d]=schur(H,'cont');

-->X'*H*X
 ans  =

! - 2.7320508  - 1.110D-15    0.           1.          !
!   0.         - 0.7320508  - 1.         - 7.772D-16 !
!   7.216D-16    0.           2.7320508    0.          !
!   0.         - 6.106D-16    0.           0.7320508 !


-->[X,d]=schur(H,'disc');

-->X'*H*X
 ans  =

!   0.7320508    0.           7.772D-16    1.          !
!   0.         - 0.7320508  - 1.           8.604D-16 !
!   0.           0.           2.7320508  - 1.166D-15 !
!   7.772D-16    1.110D-15  - 1.277D-15  - 2.7320508 !


-->//Selection of user-defined eigenvalues (# 3 and 4 here);
```

```
-->deff('[flg]=sel(x)',...
-->'flg=0,ev=x(2)/x(3),...
--> if abs(ev-ww(3))<0.0001|abs(ev-ww(4))<0.00001 then flg=1,end')

-->[X,d]=schur(H,sel)
 d  =

    2.
 X  =

! - 0.5705632  - 0.2430494  - 0.6640233  - 0.4176813 !
! - 0.4176813    0.6640233  - 0.2430494    0.5705632 !
!   0.5705632  - 0.2430494  - 0.6640233    0.4176813 !
!   0.4176813    0.6640233  - 0.2430494  - 0.5705632 !


-->X'*H*X
 ans  =

!   0.7320508    0.           7.772D-16    1.         !
!   0.         - 0.7320508  - 1.           8.604D-16 !
!   0.           0.           2.7320508  - 1.166D-15 !
!   7.772D-16    1.110D-15  - 1.277D-15  - 2.7320508 !


-->//               With matrix pencil

-->[X,d]=gschur(H,eye(H),sel)
 d  =

    2.
 X  =

!   0.5705632    0.2430494    0.6640233    0.4176813 !
!   0.4176813  - 0.6640233    0.2430494  - 0.5705632 !
! - 0.5705632    0.2430494    0.6640233  - 0.4176813 !
! - 0.4176813  - 0.6640233    0.2430494    0.5705632 !

-->X'*H*X
 ans  =

!   0.7320508    0.           9.576D-16    1.         !
!   0.         - 0.7320508  - 1.           0.         !
!   8.882D-16    0.           2.7320508    0.         !
!   0.           0.           0.         - 2.7320508 !


-->//            block diagonalization
```

```
-->[ab,x,bs]=bdiag(H);


-->inv(x)*H*x
 ans  =

!   2.7320508    1.610D-15    0.           0.        !
! - 3.664D-15  - 2.7320508    0.           6.661D-16 !
!   0.           0.           0.7320508  - 7.910D-16 !
!   0.           0.           0.         - 0.7320508 !


-->//                     Matrix pencils

-->E=rand(3,2)*rand(2,3);

-->A=rand(3,2)*rand(2,3);

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


-->w=det(s*D-A)   //determinant
 w  =

                           2
  - 0.0149837s + 0.0004193s

-->[al,be]=gspec(A,E);

-->al./(be+%eps*ones(be))
 ans  =

!   9.202D+14 !
!   35.734043 !
! - 1.170D-16 !

-->roots(w)
 ans  =

!   0         !
!   35.734043 !

-->[Ns,d]=coffg(s*D-A);    //inverse of polynomial matrix;

-->clean(Ns/d*(s*D-A))
 ans  =
```

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

-->[Q,M,i1]=pencan(E,A);   // Canonical form;
    rank A^k    rcond
        2.      0.169D-01
    rank A^k    rcond
        2.      0.847D+00

-->clean(M*E*Q)
 ans  =

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

-->clean(M*A*Q)
 ans  =

!   35.774235  - 3.0560234    0. !
!   0.4704929  - 0.0401920    0. !
!   0.           0.           1. !


-->//           PAUSD-RESUME

-->write(%io(2),'pause command...');
pause command...

-->write(%io(2),'TO CONTINUE...');
TO CONTINUE...

-->write(%io(2),'ENTER ''resume (or return) or click on resume!!''');
ENTER 'resume (or return) or click on resume!!'

-->//pause;


-->//           CALLING EXTERNAL ROUTINE
```

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

-->unix_s('rm -f foo.c')

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

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

-->link('foo.o','foo','C')    //Linking to Scilab
 ans  =

    0.

-->//5+7 by C function

-->fort('foo',5,1,'d',7,2,'d','out',[1,1],3,'d')
 ans  =

    12.
```

Appendix B

System interconnexion

The purpose of this appendix is to illustrate some of the more sophisticated aspects of Scilab by the way of an example. The example shows how Scilab can be used to symbolically represent the inter-connection of multiple systems which in turn can then be used to numerically evaluate the performance of the inter-connected systems. The symbolic representation of the inter-connected systems is done with a function called `bloc2exp` and the evaluation of the resulting system is done with `evstr`.

The example illustrates the symbolic inter-connection of the systems shown in Figure B.1. Figure B.1 illustrates the classic regulator problem where the block labeled `Proc` is to be controlled using feedback from the `Sensor` block and `Reg` block. The `Reg` block compares the output from the `Model` block to the output from the `Sensor` block to decide how to regulate the `Proc` block. There is also a feed-forward block which filters the input signal `U` to the `Proc` block. The outputs of the system are `Y` and `UP`.

The system illustrated in Figure B.1 can be represented in Scilab by using the function `bloc2exp`. The use of `bloc2exp` is illustrated in the following Scilab session. There a two kinds of objects: "transfer" and "links". The example considered here admits 5 transfers and 7 links. First the transfer are defined in a symbolic manner. Then links are defined and an "interconnected system" is defined as a specific list. The function `bloc2exp` evaluates symbolically the global transfer and `evstr` evaluates numerically the global transfer function once the systems are given "values", i.e. are defined as Scilab linear systems.

```
-->model=2;reg=3;proc=4;sensor=5;ff=6;somm=7;

-->tm=list('transfer','model');tr=list('transfer',['reg(:,1)','reg(:,2)']);

-->tp=list('transfer','proc');ts=list('transfer','s
ensor');

-->tf=list('transfer','ff');tsum=list('transfer',['1','1']);

-->lum=list('link','input',[-1],[model,1],[ff,1]);

-->lmr=list('link','model output',[model,1],[reg,1]);

-->lrs=list('link','regulator output',[reg,1],[somm,1]);
```

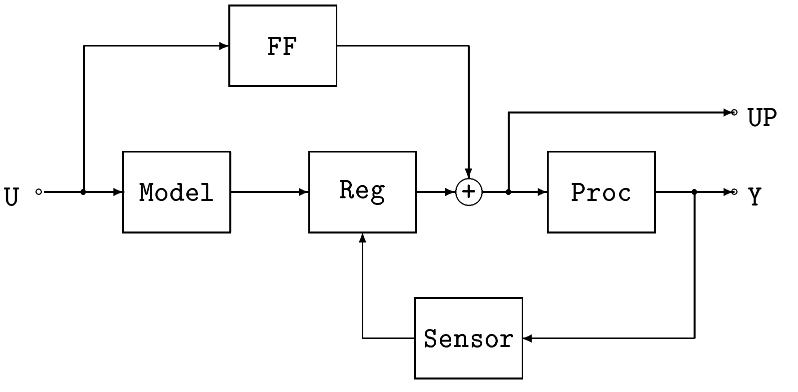


Figure B.1: Inter-Connected Systems

```
-->lfs=list('link','feed-forward output',[ff,1],[somm,2]);

-->lsp=list('link','proc input',[somm,1],[proc,1],[-2]);

-->lpy=list('link','proc output',[proc,1],[sensor,1],[-1]);

-->lsup=list('link','sensor output',[sensor,1],[reg,2]);

-->syst=...
 list('blocd',tm,tr,tp,ts,tf,tsum,lum,lmr,lrs,lfs,lsp,lpy,lsup);

-->[sysf,names]=bloc2exp(syst)
 names     =


      names>1

 input

      names>2

!proc output  !
!             !
!proc input   !
 sysf      =

!proc*((eye()-reg(:,2)*sensor*proc)\(-(-ff-reg(:,1)*model)))  !
!                                                             !
```

```
!(eye()-reg(:,2)*sensor*proc)\(-(-ff-reg(:,1)*model))          !
```

Note that the argument to `bloc2exp` is a list of lists. The first element of the list `syst` is (actually) the character string `'blocd'` which indicates that the list represents a block-diagram inter-connection of systems. Each list entry in the list `syst` represents a block or an inter-connection in Figure B.1. The form of a list which represents a block begins with a character string `'transfer'` followed by a matrix of character strings which gives the symbolic name of the block. If the block is multi-input multi-output the matrix of character strings must be represented as is illustrated by the block `Reg`.

The inter-connections between blocks is also represented by lists. The first element of the list is the character string `'link'`. The second element of the inter-connection is its symbolic name. The third element of the inter-connection is the input to the connection. The remaining elements are all the outputs of the connection. Each input and output to an inter-connection is a vector which contains as its first element the block number (for instance the `model` block is assigned the number 2). The second element of the vector is the port number for the block (for the case of multi-input multi-output blocks). If an inter-connection is not attached to anything the value of the block number is negative (as for example the inter-connection labeled `'input'` or is omitted.

The result of the `bloc2exp` function is a list of names which give the unassigned inputs and outputs of the system and the symbolic transfer function of the system given by `sysf`. The symbolic names in `sysf` can be associated to polynomials and evaluated using the function `evstr`. This is illustrated in the following Scilab session.

```
-->s=poly(0,'s');ff=1;sensor=1;model=1;proc=s/(s^2+3*s+2);

-->reg=[1/s 1/s];sys=evstr(sysf)
 sys       =

!     1 + s          !
!   ----------       !
!             2      !
!   1 + 3s + s       !
!                    !
!              2   3 !
!   2 + 5s + 4s + s  !
!   ---------------- !
!           2   3    !
!     s + 3s + s     !
```

The resulting polynomial transfer function links the input of the block system to the two outputs. Note that the output of `evstr` is the rational polynomial matrix `sys` whereas the output of `bloc2exp` is a matrix of character strings.

The symbolic evaluation which is given here is not very efficient with large interconnected systems. The function `bloc2ss` performs the previous calculation in state-space format. Each system is given now in state-space as a `syslin` list or simply as a gain (constant matrix). Note `bloc2ss` performs the necessary conversions if this is not done by the user. Each system must be given a value before bloc2ss is called. All the calculations are made in state-space representation even if the linear systems are given in transfer form.

Appendix C

Converting Scilab functions to Fortran

C.1 Converting Scilab Functions to Fortran Routines

Scilab provides a compiler (under development) to transform some Scilab functions into Fortran subroutines. The routines which are thus obtained make use of the routines which are in the Fortran libraries. All the basic matrix operations are available.

Let us consider the following Scilab function:

```
function [x]=macr(a,b,n)
z=n+m+n,
c(1,1)=z,
c(2,1)=z+1,
c(1,2)=2,
c(2,2)=0,
if n=1 then,
 x=a+b+a,
else,
x=a+b-a'+b,
end,
y=a(3,z+1)-x(z,5),
x=2*x*x*2.21,
sel=1:5,
t=a*b,
for k=1:n,
 z1=z*a(k+1,k)+3,
end,
t(sel,5)=a(2:4,7),
x=[a b;-b' a']
```

which can be translated into Fortran by using the function `mac2for`. Each input parameter of the subroutine is described by a list which contains its type and its dimensions. Here, we have three input variables `a,b,c` which are, say, `double precision`, `double precision, integer` with dimensions `(m,m), (m,m), (1,1)`. This information is gathered in the following list:

```
l=list();
l(1)=list('1','m','m');
l(2)=list('1','m','m');
l(3)=list('0','1','1');
```

The call to `mac2for` is made as follows:

```
comp(macr);
mac2for(macr2lst(macr),l)
```

The output of this command is a string containing a stand-alone Fortran subroutine.

```
      subroutine macr(a,b,n,x,m,work,iwork)
c!
c  automatic translation
.
.
.
      double precision a(m,m),b(m,m),x(m+m,m+m),y,z1,24(m,m),work(*)
      integer n,m,z,c(2,2),sel(5),k,iwork(*)
.
.
.
      call dmcopy(b,m,x(1,m+1),m+m,m,m)
      call dmcopy(work(iw1),m,x(m+1,1),m+m,m,m)
      call dmcopy(work(iw1),m,x(m+1,m+1),m+m,m,m)
      return
c
      end
```

This routine can be linked to Scilab and interactively called.

List of Figures

Index