Stephen L. Campbell, Jean-Philippe Chancelier and Ramine Nikoukhah

Modeling and Simulation in Scilab/Scicos

Mathematics Subject Classification (2000): 01-01, 04-01, 11 Axx, 26-01

Library of Congress Control Number: 2005930797

ISBN-10: 0-387-27802-8
ISBN-13: 978-0387278025

Printed on acid-free paper.

© 2006 Springer Science + Business Media, Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, Inc., 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaption, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden. The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America. (BPR/HAM)

9 8 7 6 5 4 3 2 1

springeronline.com

Preface

Scilab (`http://www.scilab.org`) is a free open-source software package for scientific computation. It includes hundreds of general purpose and specialized functions for numerical computation, organized in libraries called toolboxes that cover such areas as simulation, optimization, systems and control, and signal processing. These functions reduce considerably the burden of programming for scientific applications.

One important Scilab toolbox is Scicos. Scicos (`http://www.scicos.org`) provides a block-diagram graphical editor for the construction and simulation of dynamical systems. Scilab/Scicos is the only open-source alternative to commercial packages for dynamical system modeling and simulation packages such as MATLAB/Simulink and MATRIXx/SystemBuild. Widely used at universities and engineering schools, Scilab/Scicos is gaining ground in industrial environments. This is due in part to the creation of an international consortium in 2003 that a number of large and small companies have joined. The consortium is responsible for providing well-tested and documented releases for various platforms and coordinates the development of the core product and toolboxes, which is done by research groups, in particular, at INRIA[1] and ENPC.[2] Currently there are over 10,000 monthly Scilab downloads from `http://www.scilab.org` alone.

Scilab includes a full user's manual, which is available with search capabilities in a help window. All commands, their syntax, and simple illustrative examples are given. While very useful in finding out the details of a particular command, this manual does not provide a tutorial on the philosophy of either Scilab or Scicos. Nor does it address how to use several of these commands together in the solution of a technical problem.

The objective of this book is to provide a tutorial for the use of Scilab/Scicos with a special emphasis on modeling and simulation tools. While it will provide useful information to experienced users, it is designed to be accessible to beginning users from a variety of disciplines. Students [33] and academic and industrial scientists and engineers should find it useful. The discussion includes some information on modeling and simulation in order to assist the reader in deciding which simulation tools might be most useful to them. Every software environment has its special features, some would say quirks, that experienced users automatically take into account but often prove confusing to beginning users. We have tried to point these out where appropriate.

The book is divided into two parts. The first part concerns Scilab and includes a tutorial covering the language features, the data structures and specialized functions for doing graphics, importing and exporting data, interfacing with external routines, etc. It

[1] Institut National de Recherche en Informatique et en Automatique

[2] Ecole Nationale des Ponts et des Chaussées

also covers in detail Scilab numerical solvers for ODEs (ordinary differential equations) and DAE's (differential-algebraic equations). Even though the emphasis is placed on modeling and simulation applications, this part provides a global view of the product.

The second part is dedicated to modeling and simulation of dynamical systems in Scicos. Scicos provides a block-diagram editor for constructing models. This type of modeling tool is widely used in industry because it provides a means for constructing modular and reusable models. This part contains a detailed description of the editor and its usage, which is illustrated through numerous examples. It also covers advanced subjects such as constructing new blocks and batch simulation. Code generation and debugging are other topics covered. Finally, a new extension of Scicos is discussed. This extension allows the use of components described by the Modelica (`http://www.modelica.org`) language.

There have been several previous books concerning Scilab. Most of these have been in French [18, 3, 2, 1, 25] and dealt with earlier versions of Scilab, as in [16]. This book is unique in a number of ways. It is the first to deal with the new Scilab 3.1 version. This book is also the first to focus on simulation and modeling. It is also the first to put a major emphasis on Scicos and discuss Scicos in depth.

The source of all the examples presented in this book can be downloaded from `http://www.scicos.org`.

Finally, a large number of people have supported us in many ways. We would especially like to thank our wives, Gail Campbell and Homa Nikoukhah, and our parents, Aline and René Chancelier, for their support in this and everything else we do.

Steve Campbell
Jean-Philippe Chancelier
Ramine Nikoukhah

Contents

Part I Scilab

1 General Information . . . 3
- 1.1 What Is Scilab? . . . 3
- 1.2 How to Start? . . . 4
  - 1.2.1 Installation . . . 4
  - 1.2.2 First Steps . . . 4
  - 1.2.3 Line Editor . . . 5
  - 1.2.4 Documentation . . . 6
- 1.3 Typical Usage . . . 6
- 1.4 Scilab on the Web . . . 7

2 Introduction to Scilab . . . 9
- 2.1 Scilab Objects . . . 11
  - 2.1.1 Matrix Construction and Manipulation . . . 12
  - 2.1.2 Strings . . . 17
  - 2.1.3 Boolean Matrices . . . 19
  - 2.1.4 Polynomial Matrices . . . 20
  - 2.1.5 Sparse Matrices . . . 21
  - 2.1.6 Lists . . . 22
  - 2.1.7 Functions . . . 26
- 2.2 Scilab Programming . . . 27
  - 2.2.1 Branching . . . 28
  - 2.2.2 Iterations . . . 29
  - 2.2.3 Scilab Functions . . . 31
  - 2.2.4 Debugging Programs . . . 35
- 2.3 Input and Output Functions . . . 37
  - 2.3.1 Display of Variables . . . 37
  - 2.3.2 Formatted Input and Output . . . 38
  - 2.3.3 Input Output in Binary Mode . . . 40
  - 2.3.4 Accessing the Host System . . . 42
  - 2.3.5 Graphical User Interface . . . 43
- 2.4 Scilab Graphics . . . 48
  - 2.4.1 Basic Graphing . . . 48
  - 2.4.2 Graphic Tour . . . 49
  - 2.4.3 Graphics Objects . . . 53

2.4.4 Scilab Graphics and LaTeX ... 56
2.4.5 Old Graphics Style ... 60
2.5 Interfacing ... 62
2.5.1 Linking Code ... 63
2.5.2 Writing an Interface ... 66
2.5.3 Dynamic Loading ... 69

3 Modeling and Simulation in Scilab ... 73
3.1 Types of Models ... 73
3.1.1 Ordinary Differential Equations ... 73
3.1.2 Boundary Value Problems ... 74
3.1.3 Difference Equations ... 75
3.1.4 Differential Algebraic Equations ... 76
3.1.5 Hybrid Systems ... 77
3.2 Simulation Tools ... 78
3.2.1 Ordinary Differential Equations ... 78
3.2.2 Boundary Value Problems ... 90
3.2.3 Difference Equations ... 95
3.2.4 Differential Algebraic Equations ... 98
3.2.5 Hybrid Systems ... 100

4 Optimization ... 107
4.1 Comments on Optimization and Solving Nonlinear Equations ... 107
4.2 General Optimization ... 108
4.3 Solving Nonlinear Equations ... 112
4.4 Nonlinear Least Squares ... 113
4.5 Parameter Fitting ... 117
4.6 Linear and Quadratic Programming ... 119
4.6.1 Linear Programs ... 119
4.6.2 Quadratic Programs ... 120
4.6.3 Semidefinite Programs ... 120
4.7 Differentiation Utilities ... 120
4.7.1 Higher Derivatives ... 122

5 Examples ... 125
5.1 Modeling and Simulation of an N-Link Pendulum ... 125
5.1.1 Equations of Motion of the N-Link Pendulum ... 126
5.1.2 Generated Code and Simulation ... 130
5.1.3 Maple Code ... 133
5.2 Modeling and Simulation of a Car ... 135
5.2.1 Basic Model ... 135
5.2.2 Equations of Motion ... 136
5.2.3 Simulation Model ... 138
5.2.4 Scilab Implementation ... 139
5.2.5 Simulation Result ... 141
5.3 Open-Loop Control to Swing Up a Pendulum ... 142
5.3.1 Model ... 142
5.3.2 Control Problem Formulation ... 142
5.3.3 Optimization Problem ... 143
5.3.4 Implementation in Scilab ... 145

5.4 Parameter Fitting and Implicit Models ... 147
5.4.1 Mathematical Model ... 148
5.4.2 Scilab Implementation ... 148

Part II Scicos

6 Introduction ... 159

7 Getting Started ... 163
7.1 Construction of a Simple Diagram ... 163
7.1.1 Running Scicos ... 163
7.1.2 Editing a Model ... 163
7.1.3 Diagram Simulation ... 165
7.1.4 Changing Block Parameters ... 166
7.2 Symbolic Parameters and Context ... 169
7.3 Hierarchy ... 173
7.3.1 Placing a Super Block in a Diagram ... 173
7.3.2 Editing a Super Block ... 174
7.4 Save and Load ... 175
7.4.1 Scicos File Formats ... 175
7.4.2 Super Block and Palette ... 176
7.5 Synchronism and Special Blocks ... 176

8 Scicos Formalism ... 179
8.1 Activation Signal ... 179
8.1.1 Block Activation ... 179
8.1.2 Activation Generation ... 181
8.2 Inheritance ... 182
8.3 Always Active Blocks ... 183
8.4 Constant Blocks ... 184
8.5 Conditional Blocks ... 184

9 Scicos Blocks ... 189
9.1 Block Behavior ... 189
9.1.1 External Activation ... 189
9.1.2 Always Activation ... 191
9.1.3 Internal Zero-Crossing ... 192
9.2 Blocks Inside Palettes ... 192
9.3 Modifying Block Parameters ... 193
9.4 Super Block and `Scifunc` ... 193
9.4.1 Super Blocks ... 193
9.4.2 `Scifunc` ... 194
9.5 Constructing New Basic Blocks ... 194
9.5.1 Interfacing Function ... 195
9.5.2 Computational Function ... 197
9.5.3 Saving New Blocks ... 207
9.6 Constructing and Loading a New Palette ... 207

10 Examples ... 209
10.1 Predator Prey Model ... 209
10.2 Control Application ... 210
10.3 Signal Processing Application ... 213
10.4 Queuing Systems ... 216
10.5 Neuroscience Application ... 218
10.6 A Fluid Model of TCP-Like Behavior ... 220
10.7 Interactive GUI ... 221

11 Batch Processing in Scilab ... 227
11.1 Piloting Scicos via Scilab Commands ... 227
11.1.1 Function `scicosim` ... 228
11.1.2 Function `scicos_simulate` ... 232
11.2 Data Sharing ... 233
11.2.1 Context Variables ... 234
11.2.2 Input/Output Files ... 234
11.2.3 Global Variables ... 236
11.3 Examples ... 237
11.4 Steady-State Solution and Linearization ... 243
11.4.1 Scilab Function `steadycos` ... 247
11.4.2 Scilab Function `lincos` ... 248

12 Code Generation ... 253
12.1 Code Generation Procedure ... 253
12.2 Limitations ... 257
12.2.1 Continuous-Time Activation ... 257
12.2.2 Synchronicism ... 258
12.3 A Look Inside ... 258
12.4 Some Pitfalls ... 260
12.5 Applications ... 263

13 Debugging ... 267
13.1 Error Messages ... 267
13.1.1 Block Errors ... 267
13.1.2 Errors During Numerical Integration ... 268
13.1.3 Other Errors ... 269
13.2 Debugging Tools ... 269
13.3 Examples ... 270
13.3.1 Log File ... 271
13.3.2 Animation ... 271

14 Implicit Scicos and Modelica ... 273
14.1 Introduction ... 273
14.2 Internally Implicit Blocks ... 275
14.3 Implicit Blocks ... 275
14.3.1 Scicos Editor ... 276
14.3.2 Scicos Compiler ... 276
14.3.3 Block Construction ... 276
14.4 Example ... 277

A Inside Scicos . . . 281
A.1 Scicos Editor . . . 281
A.1.1 Main Editor Function . . . 281
A.1.2 Structure of `scs_m` . . . 283
A.2 Scicos Complier . . . 286
A.2.1 First Compilation Stage . . . 286
A.2.2 Second Compilation Stage . . . 287
A.2.3 Structure of `%cpr` . . . 287
A.2.4 Partial Compilation . . . 290
A.3 Scicos Simulator . . . 291

B Scicos Blocks of Type 5 . . . 293
B.1 Type 5 Block for the Bouncing Ball Example . . . 293
B.2 Animation Block for the Cart Pendulum Example . . . 294

C Animation Program for the Car Example . . . 299

D Extraction Program for the LaTeX Graphic Example . . . 301

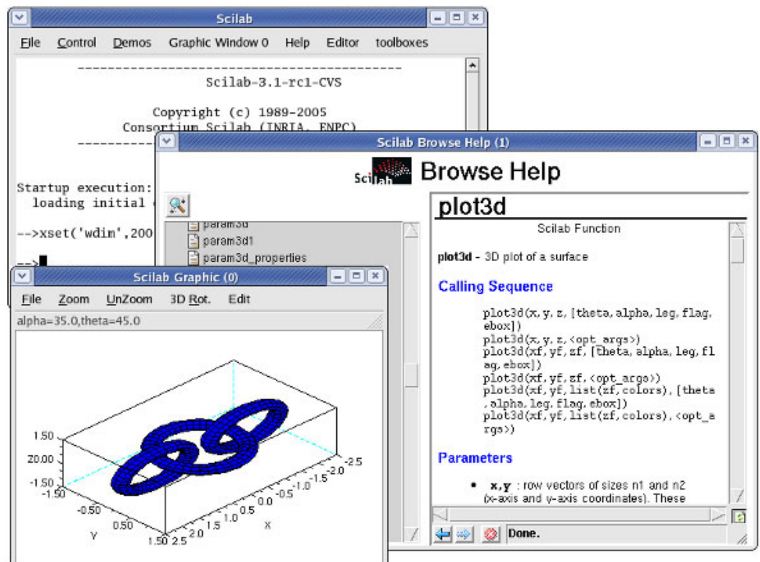
E Maple Code Used for Modeling the *N*-Link Pendulum . . . 303

References . . . 307

Index . . . 309

Part I

Scilab



1

General Information

1.1 What Is Scilab?

There exist two categories of general scientific software: computer algebra systems that perform symbolic computations, and general purpose numerical systems performing numerical computations and designed specifically for scientific applications. The best-known examples in the first category are Maple, Mathematica, Maxima, Axiom, and MuPad. The second category represents a larger market dominated by MATLAB. Scilab, which is free open-source software, belongs to this second category.

Scilab is an interpreted language with dynamically typed objects. Scilab runs, and is available in binary format, for the main available platforms: Unix/Linux workstations (the main software development is performed on Linux workstations), Windows, and MacOSX. MacOSX users can also install Scilab using fink. Compiling Scilab from the source code is also possible and is fairly straightforward.

Scilab was originally named Basile and was developed at INRIA as part of the Meta2 project. Development continued under the name of Scilab by the Scilab group, which was a team of researchers from INRIA Metalau and ENPC. Since 2004, Scilab development has been coordinated by a consortium.

Scilab can be used as a scripting language to test algorithms or to perform numerical computations. But it is also a programming language, and the standard Scilab library contains around 2000 Scilab coded functions. The Scilab syntax is simple, and the use of matrices, which are the fundamental object of scientific calculus, is facilitated through specific functions and operators. These matrices can be of different types including real, complex, string, polynomial, and rational. Scilab programs are thus quite compact and most of the time are smaller than their equivalents in C, C++, or Java.

Scilab is mainly dedicated to scientific computing, and it provides easy access to large numerical libraries from such areas as linear algebra, numerical integration, and optimization. It is also simple to extend the Scilab environment. One can easily import new functionalities from external libraries into Scilab by using static or dynamic links. It is also possible to define new data types using Scilab structures and to overload standard operators for new data types. Numerous toolboxes that add specialized functions to Scilab are available on the official site.

Scilab also provides many visualization functionalities including 2D, 3D, contour and parametric plots, and animation. Graphics can be exported in various formats such as Gif, Postscript, Postscript-Latex, and Xfig. In addition to Scilab's user interface functions, the Scilab Tcl/Tk interface can be used to develop sophisticated GUI's (Graphical user interfaces).

Scilab is a large software package containing approximately 13,000 files, more than 400,000 lines of source code (in C and Fortran), 70,000 lines of Scilab code (specialized libraries), 80,000 lines of online help, and 18,000 lines of configuration files. These files include

- Elementary functions of scientific calculation;
- Linear algebra, sparse matrices;
- Polynomials and rational functions;
- Classic and robust control, LMI optimization;
- Nonlinear methods (optimization, ODE and DAE solvers, Scicos, which is a hybrid dynamic systems modeler and simulator);
- Signal processing;
- Random sampling and statistics;
- Graphs (algorithms, visualization);
- Graphics, animation;
- Parallelism using PVM;
- MATLAB-to-Scilab translator;
- A large number of contributions for various areas.

1.2 How to Start?

1.2.1 Installation

Scilab is available for downloading at `http://www.scilab.org`. The procedure for installing Scilab depends on the operating system (Windows, MacOSX, Linux, or Unix), and information can be found on the website. The user has the choice between installing the binary version (if one is available for the host system) and compiling the source version. To compile the source version, the host system must be equipped with appropriate C and Fortran compilers. For the Windows operating system, a Visual C++ compiler suffices because an `f2c` (Fortran-to-C) translator is included in the source code. Note that C and Fortran compilers are already installed on most Linux platforms, and for other Unix systems, if native compilers are not available, freely available GNU compilers can be used. On computers running MacOSX there is also the option of installing Scilab using fink.

In this book, `SCI` designates the directory in which Scilab is installed. We use the Unix notation for specifying path names. For example, `SCI/routines/machine.h` is the file `machine.h` in the subdirectory `routines`. Under the Windows operating system, all the " `/` " should be replaced with " `\` ".

Even though there are no restrictions on its use, Scilab is copyrighted; see the file `license.txt` (English) or `licence.txt` (French) on the website.

1.2.2 First Steps

Running Scilab opens up a command window; see Figure 1.1. The look of this window may differ depending on the window manager.

The Scilab command window is an interactive window where the user is invited to enter a command at Scilab's prompt (`-->`). The command must be validated with a carriage return, after which Scilab executes the command and returns control to the user by displaying a new prompt.

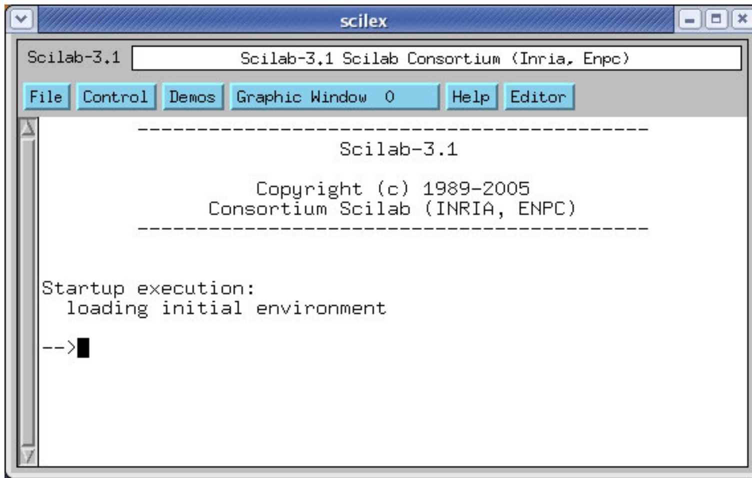


Figure 1.1. Scilab's main window.

The best way to start exploring Scilab is to run the demos. This can be done by clicking on the `Demos` button at the top of the command window (under the Windows operating system, the `Demos` button is in the "?" menu). The demos are chosen to present typical uses of the software and some of its specialized toolboxes.

For each demo, the user can see the corresponding Scilab source code, which shows that the data types used in Scilab are, for the most part, vectors and matrices. Their usage, very close to the usual matrix notation, results in compact and readable code. This, and the fact that there is no need for type declaration, compilation, or memory allocation, makes Scilab a lot easier to use than low-level languages such as C and Fortran. Just as for any other interpreted language, however, there is a price to be paid in terms of efficiency. This could become a factor in some applications.

1.2.3 Line Editor

Limited editing facilities are available in the command window. Besides the usual cut and paste operations, line editing can be done using control characters as is done in `emacs`: `Ctrl-b` (pressing `b` while holding the `Ctrl` key down) for moving the cursor back by one character, `Ctrl-f` for moving it forward, `Ctrl-a` to place the cursor at the beginning of the command line, and `Ctrl-e` for placing it at the end. Also `Ctrl-k` erases the part of the command line between the current position of the cursor and the end of the line and saves it in a buffer, and `Ctrl-y` inserts the content of the buffer at the current position of the cursor. Previously entered commands can be searched using up and down arrows or equivalently with `Ctrl-p` and `Ctrl-n`.

Under Unix and Linux operating systems, an additional feature is provided for recalling a previously entered command by typing the beginning of the command line after an exclamation point, followed by a carriage return.

All the commands entered in Scilab are automatically saved in a file called `scilab.hist` in the user's home directory.

1.2.4 Documentation

Scilab has a comprehensive online help facility, which can be consulted through the commands `help` and `apropos`. To consult the manual page corresponding to a Scilab function, the command `help` followed by the name of the function can be used. This opens up a browser window displaying the manual page in question. The manual page contains a detailed description of the function and a number of examples of its usage. The examples can be cut and pasted into Scilab's command window to be executed. The browser, which can also be accessed by clicking on the `help` button at the top of the command window, contains a list of all the functions classified by theme in different chapters (see Fig. 1.2). The manual page of a function can then be obtained by clicking on its name.

To obtain a list of Scilab functions corresponding to a keyword, the command `apropos` followed by the keyword should be used.

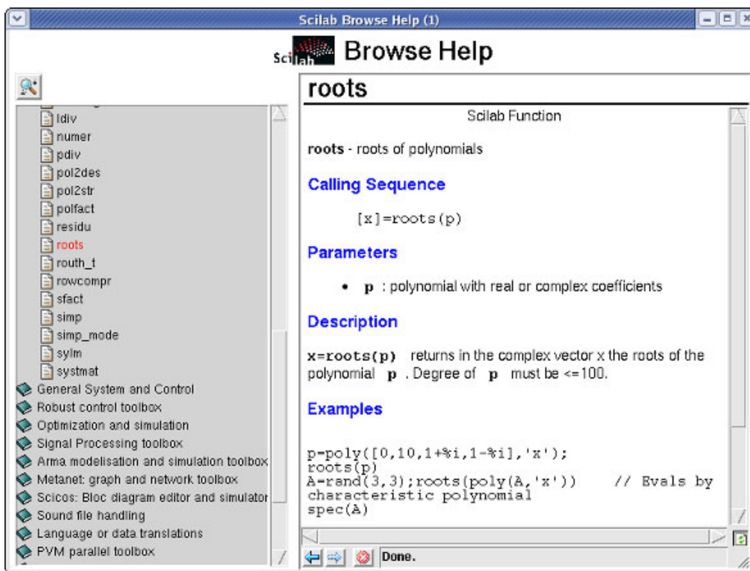


Figure 1.2. Help browser window.

The demos are also a good source of inspiration. They present simple examples of Scilab programming situations frequently encountered by users. The graphics demos, for example, give an overall picture of what can be done in Scilab as far as visualization is concerned. The demos' source codes often provide a good starting point for developing complex applications.

The demos provide examples for doing graphics, signal processing, systems control, Scilab simulation (in particular, the famous bicycle example), Scicos simulation, and a lot more.

1.3 Typical Usage

A typical Scilab user spends most of his time going back and forth between Scilab and a text editor. For the most part, Scilab programs contain very few lines of instructions,

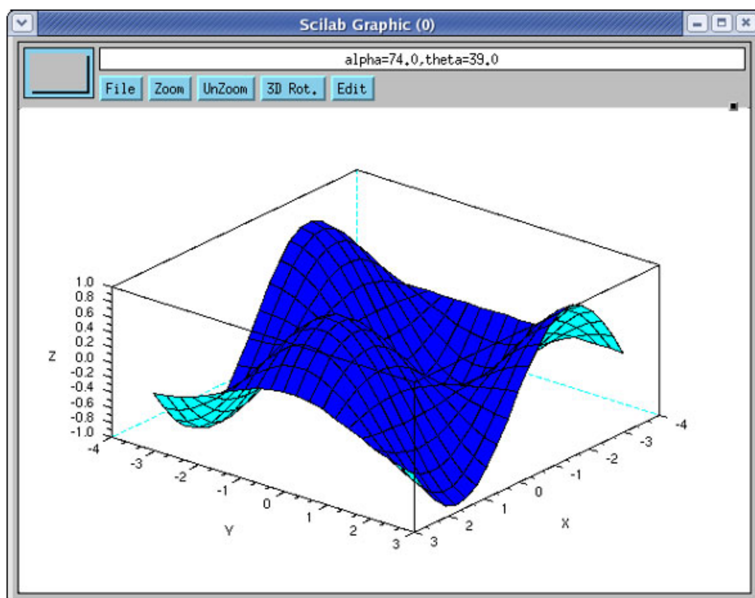


Figure 1.3. Graphic window.

thanks to the powerful data types and primitives available in Scilab. These programs can be written using a built-in editor, which is activated through the `Editor` menu button or using the user's favorite text editor (for example `vi` or `emacs`, for which there exists a Scilab mode, under Unix, and `notepad` or `wordpad` under Windows), then loaded into Scilab. This can be done through the built-in editor menus or using Scilab functions `getf` and `exec` (discussed in Section 2.2), or through the menu `File operations`.

1.4 Scilab on the Web

The latest release of Scilab, its documentation, and many third-party contributions (toolboxes) can be found on the official Scilab home page at

`http://www.scilab.org`

On this site, a bug and request report system (based on `Bugzilla`) for Scilab is available:

`http://scilabsoft.inria.fr/cgi-bin/bugzilla_bug/index.cgi`

There is also a newsgroup dedicated to Scilab:

`comp.soft-sys.math.scilab`

A specific site devoted to teaching with Scilab is also available at

`http://cermics.enpc.fr/scilab/`

Finally, the email address `scilab@inria.fr` can be used for contacting the Scilab group concerning contributions or simply for asking questions.

2

Introduction to Scilab

The Scilab language was initially devoted to matrix operations, and scientific and engineering applications were its main target. But over time, it has considerably evolved, and currently the Scilab language includes powerful operators for manipulating a large class of basic objects.

In Scilab, objects are never declared or allocated explicitly; they have a dynamic type and their sizes can dynamically change according to applied operators or functions. The following Scilab session illustrates the creation of an object, its dynamic change, and its deletion. Note that the function `typeof` returns the type of the object, `isdef` tests the presence of the object, and `clear` destroys it and frees the corresponding memory.

```
⟼ a= rand(2,3);                       ← 2x3 random matrix a is created
⟼ typeof(a)          ← type of a, constant stands for real or complex matrix
 ans  =

 constant

⟼ a= [a, zeros(2,1)]                  ← size of a dynamically increases
 a  =

!   0.2113249    0.0002211    0.6653811    0.  !
!   0.7560439    0.3303271    0.6283918    0.  !

⟼ a= 'scilab'; typeof(a)              ← redefining a through reassignment
 ans  =

 string

⟼ exists('a')                         ← checks if a exists
 ans  =

    1.

⟼ clear('a'); exists('a')    ← removing a explicitly from current environ-
                               ment and verifying that a is gone
 ans  =

    0.
```

Available memory for a Scilab session can be controlled through the use of the `stacksize` command. It is of course bounded by available dynamic heap memory (allocated memory through the use of the `malloc` function). Using Scilab commands `who` and `whos`, one can get the name and size of dynamic objects present in the current session:

```
⟼ stacksize()                       ← available memory (number of double) and maximum number of variables
 ans  =

!   1000000.    6185. !

⟼ A=rand(1000,1000);
                    !--error    17
rand: stack size exceeded (Use stacksize  function to increase it)

⟼ stacksize(6000000);               ← increasing Scilab available memory
⟼ A=rand(1000,1000);                ← A is created, note that an expression ending with ; is evaluated without display
⟼ whos('-name','A');                ← get info on variable A
Name                      Type           Size           Bytes

A                         constant       1000 by 1000   8000016
```

By default, numbers in Scilab are coded as double-precision floats. The accuracy of double-precision computation is machine-dependent and can be obtained through the predefined Scilab variable `%eps`, called the machine precision. `%eps` is the largest double-precision number for which `1+(%eps)/2` is indistinguishable from 1.

The set of numbers in Scilab can be extended by adding `%inf` (infinity) and `%nan` (not a number). This is done with the Scilab function `ieee`:

```
⟼ ieee()                            ← get the default floating-point exception mode
 ans  =

    0.

⟼ 1/0                               ← 1/0 raises an error in the default mode
    !--error    27
division by zero...

⟼ ieee(2)                           ← using the standard ieee floating-point exception mode
⟼ 1/0                               ← 1/0 evaluates now to +∞
 ans  =

   Inf

⟼ 0/0                               ← no error is produced
 ans  =

   Nan
```

Scilab expressions are evaluated by the `scilab` interpreter. The interpretor has access to a large set of functions. Some of these functions are Scilab-coded functions (we shall see

later that Scilab is a programming language), some are hard-coded functions (C or Fortran coded). This ability of Scilab to include C or Fortran coded functions means that a large amount of existing scientific and engineering software can be used from within Scilab.

Scilab is a rich and powerful programming environment because of this large number of available functions. The users can also enrich the basic Scilab environment with their own Scilab or hard-coded functions, and moreover this can be done dynamically (hard-coded functions can be dynamically loaded in a running Scilab session); we shall see how this can be done later. Thus, toolboxes can be developed to enrich and adapt Scilab to specific needs or areas of application.

2.1 Scilab Objects

Scilab provides users with a large variety of basic objects starting with numbers and character strings up to more sophisticated objects such as booleans, polynomials, and structures. A Scilab object is a basic object or a set of basic objects arranged in a vector, a matrix, a hypermatrix, or a structure (list).

As already mentioned, Scilab is devoted to scientific computing, and the basic object is a two-dimensional matrix with floating-point double-precision number entries. In fact, a real scalar is nothing but a 1×1 matrix. The next Scilab session illustrates this type of object. Note that some numerical constants are predefined in Scilab. Their corresponding variable names start with %. In particular, π is %pi, and e, the base of the natural log, is %e. Note also that a:b:c gives numbers starting from a to c spaced b units apart.

```
⟼ a=1:0.6:3                          ← a is now a scalar matrix (double)
 a  =

!   1.    1.6    2.2    2.8 !

⟼ b=[%e,%pi]         ← b is the 1x2 row vector filled with predefined values (e, π)
 b  =

!   2.7182818    3.1415927 !
```

New types have been added as Scilab has evolved, but the matrix aspect has always been kept. String matrices, boolean matrices, sparse matrices, integer matrices (int8, int16, int32), polynomial matrices, and rational matrices are now available in the standard Scilab environment. Complex structures called lists (list, tlist, and mlist) are also available. Note also that functions in Scilab are considered as objects as well.

```
⟼ a= "Scilab"                                   ← a 1x1 string matrix
 a  =

 Scilab

⟼ b=rand(2,2)                                           ← a matrix
 b  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !
```

```
⟼b= b>= 0.5                                              ← a boolean matrix
 b  =

! F F !
! T F !

⟼L=list(a,b)                                             ← a list
 L  =

       L(1)

 Scilab

       L(2)

! F F !
! T F !

⟼A.x = 32;A.y = %t                  ← a structure implemented using mlist
 A  =

   x: 32
   y: %t

⟼a= spec(rand(3,3))             ← eigenvalues: a vector of complex numbers
 a  =

!   1.8925237               !
!   0.1887123 + 0.0535764i  !
!   0.1887123 - 0.0535764i  !
```

It is possible to define new types in Scilab in the sense that it is possible to define objects whose dynamic type (the value returned by `typeof`) is user-defined. Scilab operators, for example the `+` and `*` operators, can be overloaded for these dynamically defined types. The new types are defined and implemented with `tlist` and `mlist` primitive types (see Section 2.1.6).

2.1.1 Matrix Construction and Manipulation

As already pointed out, one of the goals of Scilab is to give access to matrix operations for any kind of matrix types. In this section we highlight general functions and operators that are common to all matrix types.

A matrix in Scilab refers to one- or two-dimensional arrays, which are internally stored as a one-dimensional array (two-dimensional arrays are stored in column order). It is therefore always possible to access matrix entries with one or two indices. Vectors and scalars are stored as matrices.

Multidimensional matrices can also be used in Scilab. They are called hypermatrices.

Elementary construction operators, which are overloaded for all matrix types, are the row concatenation operator “`;`” and the column concatenation operator “`,`”. These two operators perform the concatenation operation when used in a matrix context, that is, when they appear between “`[`” and “`]`”. All the associated entries must be of the same

type. Note that in the same matrix context a white space means the same thing as "," and a line feed means the same thing as ";". However, this equivalence can lead to confusion when, for example, a space appears inside an entry, as illustrated in the following:

```
⟼A=[1,2,3 +5]                          ← here A=[1,2,3,+5] with a unary +
 A  =

!   1.    2.    3.    5. !

⟼A=[1,2,3 *5]                          ← here A=[1,2,3*5] with a binary *
 A  =

!   1.    2.    15. !

⟼A=[A,0; 1,2,3,4]
 A  =

!   1.    2.    15.    0. !
!   1.    2.    3.     4. !
```

' and .'	transpose (conjugate or not)
diag	(m,n) matrix with given diagonal (or diagonal extraction)
eye	(m,n) matrix with one on the main diagonal
grand	(m,n) random matrix
int	integer part of a matrix
linspace or " : "	linearly spaced vector
logspace	logarithmically spaced vector
matrix	reshape an (m,n) matrix (m*n is kept constant)
ones	(m,n) matrix consisting of ones
rand	(m,n) random matrix (uniform or Gaussian)
zeros	(m,n) matrix consisting of zeros
.*.	Kronecker operator

Table 2.1. A set of functions for creating matrices.

Table 2.1 describes frequently used matrix functions that can be used to create special matrices. These matrix functions are illustrated in the following examples:

```
⟼A= [eye(2,1), 3*ones(2,3); linspace(3,9,4); zeros(1,4)]
 A  =

!   1.    3.    3.    3. !
!   0.    3.    3.    3. !
!   3.    5.    7.    9. !
!   0.    0.    0.    0. !

⟼d=diag(A)'                          ← main diagonal of A as a row matrix
 d  =
```

```
!   1.    3.    7.    0. !

⟼B=diag(d)                                  ← builds a diagonal matrix
 B  =

!   1.    0.    0.    0. !
!   0.    3.    0.    0. !
!   0.    0.    7.    0. !
!   0.    0.    0.    0. !

⟼C=matrix(d,2,2)                            ← reshape vector d
 C  =

!   1.    7. !
!   3.    0. !
```

The majority of Scilab functions are implemented in such a way that they accept matrix arguments. Most of the time this is implemented by applying mathematical functions elementwise. For example, the exponential function `exp` applied to a matrix returns the elementwise exponential and differs from the important matrix exponential function `expm`.

```
⟼A=rand(2,2);                     ← a random matrix (uniform law)
⟼B=exp(A)                         ← elementwise exponential
 B  =

!   1.2353136    1.0002212 !
!   2.1298336    1.3914232 !

⟼B=expm(A)                        ← square matrix exponential
 B  =

!   1.2354211    0.0002901 !
!   0.9918216    1.391535  !
```

Extraction, Insertion, and Deletion

To specify a set of matrix entries for the matrix `A` we use the syntax `A(B)` or `A(B,C)`, where `B` and `C` are numeric or boolean matrices that are used as indices. The interpretation of `A(B)` and `A(B,C)` depends on whether it is on the left- or right-hand side of an assignment `=` if an assignment is present.

If we have `A(B)` or `A(B,C)` on the left, and the right-hand side evaluates to a nonnull matrix, then an assignment operation is performed. In that case the left member expression stands for a submatrix description whose entries are replaced by the ones of the right matrix entries. Of course, the right and left submatrices must have compatible sizes, that is, they must have the same size, or the right-hand-side matrix must be a scalar.

If the right-hand-side expression evaluates to an empty matrix, then the operation is a deletion operation. Entries of the left-hand-side matrix expression are deleted. Assignment or deletion can change dynamically the size of the left-hand-side matrix.

```
|—→ clear A;
|—→ A(2,4) = 1                        ← assigns 1 to (2,4) entry of A
 A  =

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

|—→ A([1,2],[1,2])=int(5*rand(2,2))   ← assignment for changing a submatrix of A
 A  =

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

|—→ A([1,2],[1,3])=[]                 ← submatrix deletion
 A  =

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

|—→ A(:,1)= 8                         ← ":" stands for all indices, here all the rows of A
 A  =

!   8.    0. !
!   8.    1. !

|—→ A(:,$)=[]                         ← deletion, "$" stands for the last index
 A  =

!   8. !
!   8. !

|—→ A(:,$+1)=[4;5]                    ← adding a new column through assignment
 A  =

!   8.    4. !
!   8.    5. !
```

When an expression `A(B)` or `A(B,C)` is not the left member of an assignment, then it stands for a submatrix extraction and its evaluation builds a new matrix.

```
|—→ A = int(10*rand(3,7));            ← int integer part
|—→ B=A([1,3],$-1:$)                  ← extracts a submatrix using row 1 and 3
                                        and the two last columns of A
 B  =

!   2.    8. !
!   2.    3. !
```

When `B` and `C` are boolean vectors and `A` is a numerical matrix, `A(B)` and `A(B,C)` specify a submatrix of matrix `A` where the indices of the submatrix are those for which `B` and `C` take the boolean value `T`. We shall see more on that in the section on boolean matrices.

Elementary Matrix Operations

Table 2.2 contains common operators for matrix types and in particular numerical matrices. If an operator is not defined for a given type, it can be overloaded. The operators in the table are listed in increasing order of precedence. In each row of the table, operators share the same precedence with left associativity except for the power operators, which are right associative.

Operators whose name starts with the dot symbol "." generally stand for term-by-term (elementwise) operations. For example, `C=A.*B` is the term-by-term multiplication of matrix `A` and matrix `B`, which is the matrix `C` with entries `C(i,j)=A(i,j)*B(i,j)`.

Operator	Description
`|`	logical or
`&`	logical and
`~`	logical not
`==, >=, <=, >, < ,<>, ~=`	comparison operators
`+,-`	binary addition and subtraction
`+,-`	unary addition and subtraction
`.*, ./, .\, .*., ./., .\., *, /, /., \.`	"multiplications" and "divisions"
`^,** , .^ , .**`	power
`', .'`	transpose

Table 2.2. Operator precedence.

Note that these operators include many important equation-solving operations such as least squared solutions. The following calculations illustrate the use of several of these operators.

```
⟼A=(1:3)' * ones(1,3)          ← transpose ' and usual matrix product *
 A  =

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

⟼A.* A'          ← multiplication tables: using a term-by-term product
 ans  =

!   1.    2.    3. !
!   2.    4.    6. !
!   3.    6.    9. !

⟼t=(1:3)';m=size(t,'r');n=3;
⟼A=(t*ones(1,n+1)).^(ones(m,1)*[0:n])    ← term-by-term exponentiation to build
                                           a Vandermonde matrix A(i,j) = t_i^(j-1)
 A  =

!   1.    1.    1.    1.  !
!   1.    2.    4.    8.  !
!   1.    3.    9.    27. !

⟼A=eye(2,2).*.[1,2;3,4]          ← Kronecker product
 A  =
```

```
!   1.    2.    0.    0. !
!   3.    4.    0.    0. !
!   0.    0.    1.    2. !
!   0.    0.    3.    4. !

|--> A=[1,2;3,4];b=[5;6];
|-->  x = A \ b ; norm(A*x -b)              <-- \ for solving a linear system Ax=b.
 ans  =

    0.

|--> A1=[A,zeros(A)]; x = A1 \ b           <-- underdetermined system: a solution with
                                               minimum norm is returned
 x  =

! - 4.  !
!   4.5 !
!   0.  !
!   0.  !

|--> A1=[A;A]; x = A1\ [b;7;8]             <-- overdetermined system: a least squared
                                               solution is returned
 x  =

! - 5.  !
!   5.5 !
```

2.1.2 Strings

Strings in Scilab are delimited by either single or double quotes “ ’ ” or “ " ”, which are equivalent. If one of these two characters is to be inserted in a string, it has to be preceded by a delimiter, which is again a single or double quote. Basic operations on strings are the concatenation (operator “ + ”) and the function `length`, which gives the string length. As expected, a matrix whose entries are strings can be built in Scilab, and the two previous operations extend to string matrix arguments as do the usual row and column concatenation operators. A string is just a `1x1` string matrix whose type is denoted by `string` (as returned by `typeof`).

```
|--> S="a string with a quote character <<"'>> "
 S  =

 a string with a quote character <<'>>

|--> S='a long string 0...                         <-- ... used to continue on next line
|-->    using continuation '
 S  =

 a long string     using continuation

|--> S=['A','string';'2x2','matrix']                          <-- a string matrix
 S  =
```

```
!A    string  !
!             !
!2x2  matrix  !

-->length(S)                          <- length of each string of S in a matrix
 ans  =

!   1.    6. !
!   3.    6. !
```

ascii	conversion from string to ascii values
execstr	send a string to the Scilab interpreter
grep	search for occurences of a string in a string matrix
part	substring extraction
msscanf	scans input from a string according to a format
msprintf	builds a string by output according to a format
strindex	finds occurrences of strings in a string
string	converts data to string
stripblanks	remove leading and trailing white (blank) characters
strsubst	string substitution in a string matrix
tokens	string tokenizer
strcat	concatenate string matrix entries
length	length of string matrix entries

Table 2.3. Some string matrix functions.

String matrix utility functions are listed in Table 2.3. The next session shows how some of them can be used.

```
-->A=rand(2,8,'n');                          <- Normal law
-->A=sign(A);                                <- just keep the signs
-->A=string(A)                               <- convert to string matrix
 A  =

!-1  1  1  1  1   1   -1  1  !
!                            !
!1   1  1  1  -1  -1  -1  1  !

-->A=strsubst(A,'1','+');                    <- string substitution
-->A=strsubst(A,'-+','-')                    <- string substitution
 A  =

!-  +  +  +  +  +  -  +  !
!                        !
!+  +  +  +  -  -  -  +  !
```

The command `execstr` can be used to evaluate a string as a Scilab expression. The given string is evaluated using values from the current context. Thus, string matrices

can be used to build Scilab expressions, which then can be evaluated as if they were entered interactively in Scilab. An important extra argument `'errcatch'` can be given to the `execstr` function in order to supress the Scilab standard error mechanism while the string is evaluated.

```
↦ name ='x'; n=3; val=[45,67,34];
↦ str = name +string(1:n)+'=val(' +string(1:n)+');'        ← a string vector
 str  =

!x1=val(1);  x2=val(2);  x3=val(3);  !

↦ execstr(str);                                   ← Scilab evaluation of str
↦ [x1,x2,x3]                                      ← x1, x2 and x3 are now defined
  ans  =

!   45.    67.    34. !
```

2.1.3 Boolean Matrices

A boolean variable can take only the two values "true" `T` and "false" `F`. Two predefined Scilab variables `%t` and `%f` respectively evaluate to `T` and `F` and can be used to build boolean matrices, for example through the use of concatenation operations.

Comparison operators (" `==` ", " `>` ", " `>=` ", " `<=` " and "`~=`") also give boolean matrices as the result when their arguments are matrices or one matrix and one scalar. Logical operators such as " `&` " (and) , " `|` " (or), and "`~`" (not) can be used as expected with boolean matrix arguments. The logical function `and` (resp. `or`) takes as argument a single boolean matrix and returns the logical and (resp. or) of the matrix entries.

Boolean matrices are used in Scilab in conjunction with conditional expressions such as the `if` and `while` conditions, which will be described later.

```
↦ true=%t;                                        ← define boolean variable
↦  if true  then disp("Hello"), end               ← conditional display
 Hello
```

The following Scilab session shows simple instructions involving booleans. We see in particular that even though matrix booleans are not coded as numbers, they can be used in numerical computations:

```
↦ ~(1>=2)
  ans  =

  T

↦ %t&%t
  ans  =

  T

↦ x=-10:0.1:10;
```

```
⟼y=((x>=0).*exp(-x))+((x<0).*exp(x));          ← automatic boolean-to-scalar conversion
⟼y=bool2s([%t,%f])                               ← explicit boolean-to-scalar conversion
 y  =

!   1.    0. !
```

We have mentioned previously that submatrix extraction can be done with boolean vectors. This is illustrated in the following session, where we also introduce the function `find`, which returns the indices of the true entries of a boolean matrix.

```
⟼A = int(10*rand(1,7))
 A  =

!   2.    7.    0.    3.    6.    6.    8. !

⟼A( A>= 3) = 0                    ← indices are given by a boolean matrix A>=5
 A  =

!   2.    0.    0.    0.    0.    0.    0. !

⟼I=find(A== 0)                                ← indices of A entries equal to 0
 I  =

!   2.    3.    4.    5.    6.    7. !
```

2.1.4 Polynomial Matrices

Polynomials are Scilab objects. Most operations available for constant matrices are also available for polynomial matrices. A polynomial can be defined using the Scilab function `poly`. It can be defined based either on its roots or its coefficients, as can be seen from the following Scilab session.

```
⟼p=poly([1 3],'s')                          ← polynomial defined by its roots
 p  =

              2
   3 - 4s + s

⟼q=poly([1 2],'s','c')                 ← polynomial defined by its coefficients
 q  =

   1 + 2s
```

Note that the `'s'` argument in `poly` specifies the character to be used for displaying the formal parameter of the polynomial. At initialization, the variable `%s` is defined to be the polynomial `s`.

Polynomials can be added together, multiplied, concatenated to form matrices, etc., provided they use the same formal parameter.

```
⟼ p+q+1                                          ← polynomial/constant addition
 ans  =

              2
   5 - 2s + s

⟼ [p*q,1]                                        ← matrix polynomial
 ans  =

!              2    3         !
!   3 + 2s - 7s + 2s      1   !
```

A polynomial can even be divided by a polynomial, and a polynomial matrix inverted, but the result, in general, is not a polynomial. It is a rational function, which is another Scilab object constructed using a list object, as we shall see later.

2.1.5 Sparse Matrices

Many areas of application lead to problems involving large matrices. But very often, large matrices involved in applications have many zero entries. Thus representations of matrices where matrix storage is reduced by coding only the nonzero matrix entries have been developed and are known as sparse matrix representations. Different coding schemes exist for performing this compression task and some schemes are tailored to specific applications. In Scilab, the sparse coding of an `(m,n)` matrix is as follows. A first array of size `m` contains pointers to row descriptors. Each row descriptor is coded with two arrays. The first array contains the column indices of the nonnull row entries, and the second array contains the associated nonzero values.

Some problems that could not be stored in Scilab memory in full form can be accessible through the use of sparse storage. As one can imagine, the reduced storage leads to more complex and slower operations on sparse matrices than on full matrices. The basic function used for building sparse matrices in Scilab is the `sparse` function. Note that only numerical or boolean matrices can be stored as sparse matrices.

```
⟼ A=sprand(100,100,0.1);          ← a (100,100) sparse matrix with 10% of
                                     nonzero values
⟼ whos('-type','sparse')                          ← check used memory
Name                     Type          Size          Bytes

A                        sparse        100 by 100    13360

⟼ B=full(A);
⟼ whos('-name','B');               ← used memory for the same matrix with a
                                     full implementation
Name                     Type          Size          Bytes

B                        constant      100 by 100    80016

⟼ timer();inv(B);timer()             ← CPU time for full matrix inversion
 ans  =

   0.01
```

```
⟼timer();inv(A);timer()                ← CPU time for sparse matrix inversion
 ans  =

   0.28
```

While useful in illustrating the commands, this example was only 100×100, which is not considered today to be a very large matrix. Also, this matrix was random, and many sparse matrices, such as those that arise by the discretization of PDEs, are not random. They have stronger structural properties that can be exploited.

2.1.6 Lists

Scilab lists are built with the `list`, `tlist`, and `mlist` functions. These three functions do not exactly build lists, but they can be considered to be structure builders in the sense that they are used to aggregate under a unique variable name a set of objects of different types. They are implemented as an array of variable-size objects that is not a list implementation. A type corresponds to each builder function, and they are recursive types (a list element can be a list).

- If the `list` constructor is used, then the stored objects are accessed by an index giving their position in the list.
- If the `tlist` constructor is used, then the built object has a new dynamic type and stored objects can be accessed through names. Note also that the fields are dynamic, which means that new fields can be dynamically added or removed from an occurrence of a `tlist`. A `tlist` remains a list, and access to stored objects through indices is also possible.
- The `mlist` constructor is a slight variation of the `tlist` constructor. The only difference is that the predefined access to stored objects through indices is no longer effective (it is, however, possible using the `getfield` and `setfield` functions). Also, extraction and insertion operators can be overloaded for `mlist` objects. This means that it is possible to give a meaning to multi-indices extraction or insertion operations. `hypermat` objects that implement multidimensional arrays are implemented using `mlist` in Scilab.

Note that many Scilab objects are implemented as `tlist` and `mlist`, and from the user point of view this is not important. For example, suppose that you want to define a variable with an extendable number of fields. This is done very easily through the use of the “.” operator:

```
⟼x.color = 4;
⟼x.value = rand(1,3);
⟼x.name = 'foo';
⟼x                                    ←x is a tlist of type struct (st)
 ans  =

  color: 4
  value: [0.2113249,0.7560439,0.0002211]
  name: "foo"
```

Among Scilab objects coded this way, it is important to mention the rational matrices and the linear state-space systems, which play important roles in modeling and analysis of linear systems.

The following session illustrates the use of rational matrices in Scilab (recall that `%s` is by default the polynomial `s`). Note that all the elementary operations are overloaded for rational matrices.

```
⟼ r=1/%s                                    ← defining a rational number
 r  =

    1
    -
    s

⟼ a=[1,r;1,1]                               ← rational matrix construction
 a  =

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

⟼ b=inv(a)                                  ← matrix inversion
 b  =

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

⟼ b.num                                     ← numerator field
  ans  =

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

⟼ b.den                                     ← denominator field
  ans  =

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

A linear state-space system is characterized in terms of four matrices, A, B, C, and D; we will describe and use these systems later for modeling linear systems. The Scilab function `ssrand` defines a random system with given input, output, and state sizes.