INTRODUCTION TO SCILAB

CONSORTIUM SCILAB
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex France

Recherche en sciences & technologies de l'information

Abstract

In this document, we make an overview of Scilab features so that we can get familiar with this environment as fast as possible. The goal is to present the *core* of skills necessary to start with Scilab. In the first part, we present how to get and install this software on our computer. We also present how to get some help with the provided in-line documentation and also thanks to web resources and forums. In the remaining sections, we present the Scilab language, especially its structured programming features. We present an important feature of Scilab, that is the management of real matrices and overview the linear algebra library. The definition of functions and the elementary management of input and output variables is presented. We present Scilab's graphical features and show how to create a 2D plot, how to configure the title and the legend and how to export that plot into a vectorial or bitmap format.

Contents

1 Overview

Scilab is a programming language associated with a rich collection of numerical algorithms covering many aspects of scientific computing problems.

From the software point of view, Scilab is an *interpreted* language. This generally allows to get faster development processes, because the user directly accesses to a high level language, with a rich set of features provided by the library. The Scilab language is meant to be extended so that user-defined data types can be defined with possibly overloaded operations. Scilab users can develop their own module so that they can solve their particular problems. The Scilab language allows to dynamically compile and link other languages such as Fortran and C: this way, external libraries can be used as if they were a part of Scilab built-in features. Scilab also interfaces LabVIEW, a platform and development environment for a visual programming language from National Instruments.

From the license point of view, Scilab is a free software in the sense that the user does not pay for it and Scilab is an open source software, provided under the Cecill license [2]. The software is distributed with source code, so that the user has an access to Scilab most internal aspects. Most of the time, the user downloads and installs, a binary version of Scilab since the Scilab consortium provides Windows, Linux and Mac OS executable versions. An online help is provided in many local languages.

From a scientific point of view, Scilab comes with many features. At the very beginning of Scilab, features were focused on linear algebra. But, rapidly, the number of features extended to cover many areas of scientific computing. The following is a short list of its capabilities:

- Linear algebra, sparse matrices,
- Polynomials and rational functions,
- Interpolation, approximation,
- Linear, quadratic and non linear optimization,
- Ordinary Differential Equation solver and Differential Algebraic Equations solver,
- Classic and robust control, Linear Matrix Inequality optimization,
- Differentiable and non-differentiable optimization,
- Signal processing,
- Statistics.

Scilab provides many graphics features, including a set of plotting functions, which allow to create 2D and 3D plots as well as user interfaces. The Xcos environment provides an hybrid dynamic systems modeler and simulator.

1.1 How to get and install Scilab

Whatever your platform is (i.e. Windows, Linux or Mac), Scilab binaries can be downloaded directly from the Scilab homepage

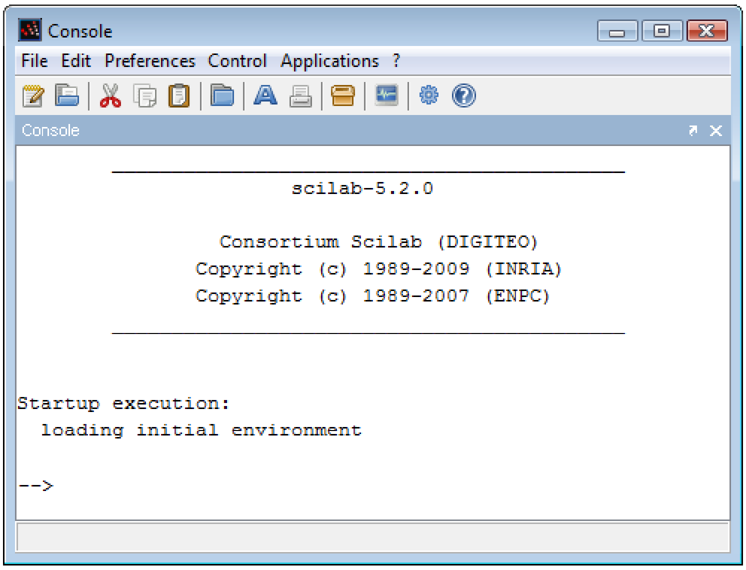

Figure 1: Scilab console under Windows.

http://www.scilab.org

or from the Download area

http://www.scilab.org/download

Scilab binaries are provided for both 32 and 64 bits platforms so that it matches the target installation machine.

Scilab can also be downloaded in source form, so that you can compile Scilab by yourself and produce your own binary. Compiling Scilab and generating a binary is especially interesting when we want to understand or debug an existing feature, or when we want to add a new feature. To compile Scilab, some prerequisites binary files are necessary, which are also provided in the Download center. Moreover, a Fortran and a C compiler are required. Compiling Scilab is a process which will not be detailed further in this document, because this chapter is mainly devoted to the external behavior of Scilab.

1.1.1 Installing Scilab under Windows

Scilab is distributed as a Windows binary and an installer is provided so that the installation is really easy. The Scilab console is presented in figure 1. Several comments may be done about this installation process.

On Windows, if your machine is based on an Intel processor, the Intel Math Kernel Library (MKL) [4] enables Scilab to perform faster numerical computations.

1.1.2 Installing Scilab under Linux

Under Linux, the binary versions are available from Scilab website as .tar.gz files. There is no need for an installation program with Scilab under Linux: simply unzip the file in one target directory. Once done, the binary file is located in *<path>/scilab-5.2.0/bin/scilab*. When this script is executed, the console immediately appears and looks exactly the same as on Windows.

Notice that Scilab is also distributed with the packaging system available with Linux distributions based on Debian (for example, Ubuntu). That installation method is extremely simple and efficient. Nevertheless, it has one little drawback: the version of Scilab packaged for your Linux distribution may not be up-to-date. This is because there is some delay (from several weeks to several months) between the availability of an up-to-date version of Scilab under Linux and its release in Linux distributions.

For now, Scilab comes on Linux with a linear algebra library which is optimized and guarantees portability. Under Linux, Scilab does not come with a binary version of ATLAS [1], so that linear algebra is a little slower for that platform.

1.1.3 Installing Scilab under Mac OS

Under Mac OS, the binary versions are available from Scilab website as a .dmg file. This binary works for Mac OS versions starting from version 10.5. It uses the Mac OS installer, which provides a classical installation process. Scilab is not available on Power PC systems.

Scilab version 5.2 for Mac OS comes with a Tcl / Tk library which is disabled for technical reasons. As a consequence, there are some small limitations on the use of Scilab on this platform. For example, the Scilab / Tcl interface (TclSci), the graphic editor and the variable editor are not working. These features will be rewritten in Java in future versions of Scilab and these limitations will disappear.

Still, using Scilab on Mac OS system is easy, and uses the shorcuts which are familiar to users of this platform. For example, the console and the editor use the Cmd key (Apple key) which is found on Mac keyboards. Moreover, there is no right-click on this platform. Instead, Scilab is sensitive to the Control-Click keyboard event.

For now, Scilab comes on Mac OS with a linear algebra library which is optimized and guarantees portability. Under Mac OS, Scilab does not come with a binary version of ATLAS [1], so that linear algebra is a little slower for that platform.

1.2 How to get help

The most simple way to get the online help integrated to Scilab is to use the function `help`. The figure 2 presents the Scilab help window. To use this function, simply type "`help`" in the console and press the <Enter> key, as in the following session.

```
help
```

Suppose that you want some help about the `optim` function. You may try to browse the integrated help, find the optimization section and then click on the `optim` item to display its help.

Another possibility is to use the function `help`, followed by the name of the function which help is required, as in the following session.

```
help optim
```

Scilab automatically opens the associated entry in the help.

We can also use the help provided on Scilab web site

http://www.scilab.org/product/man

Figure 2: Scilab help window.

That page always contains the help for the up-to-date version of Scilab. By using the "search" feature of my web browser, I can most of the time quickly find the help I need. With that method, I can see the help of several Scilab commands at the same time (for example the commands `derivative` and `optim`, so that I can provide the cost function suitable for optimization with `optim` by computing derivatives with `derivative`).

A list of commercial books, free books, online tutorials and articles is presented on the Scilab homepage:

http://www.scilab.org/publications

1.3 Mailing lists, wiki and bug reports

The mailing list *users@lists.scilab.org* is designed for all Scilab usage questions. To subscribe to this mailing list, send an e-mail to *users-subscribe@lists.scilab.org*. The mailing list *dev@lists.scilab.org* focuses on the development of Scilab, be it the development of Scilab core or of complicated modules which interacts deeply with Scilab core. To subscribe to this mailing list, send an e-mail to *dev-subscribe@lists.scilab.org*.

These mailing lists are archived at:

http://dir.gmane.org/gmane.comp.mathematics.scilab.user

and:

http://dir.gmane.org/gmane.comp.mathematics.scilab.devel

Therefore, before asking a question, users should consider looking in the archive if the same question or subject has already been answered.

A question posted on the mailing list may be related to a very specific technical point, so that it requires an answer which is not general enough to be public. The address *scilab.support@scilab.org* is designed for this purpose. Developers of the Scilab team provide accurate answers via this communication channel.

The Scilab wiki is a public tool for reading and publishing general information about Scilab:

http://wiki.scilab.org

It is used both by Scilab users and developers to publish information about Scilab. From a developer point of view, it contains step-by-step instructions to compile Scilab from the sources, dependencies of various versions of Scilab, instructions to use Scilab source code repository, etc...

The Scilab Bugzilla *http://bugzilla.scilab.org* allows to submit a report each time we find a new bug. It may happen that this bug has already been discovered by someone else before. This is why it is advised to search in the bug database for existing related problems before reporting a new bug. If the bug is not already reported, it is a very good thing to report it, along with a test script. This test script should remain as simple as possible, which allows to reproduce the problem and identify the source of the issue.

An efficient way of getting up-to-date information is to use RSS feeds. The RSS feed associated with the Scilab website is

http://www.scilab.org/en/rss_en.xml

This channel delivers regularly press releases and general announces.

1.4 Getting help from Scilab demonstrations and macros

The Scilab consortium maintains a collection of demonstration scripts, which are available from the console, in the menu *?>Scilab Demonstrations*. The figure 3 presents the demonstration window. Some demonstrations are graphic, while some others are interactive, which means that the user must type on the <Enter> key to go on to the next step of the demo.

The associated demonstrations scripts are located in the Scilab directory, inside each module. For example, the demonstration associated with the optimization module is located in the file

```
<path>\scilab-5.2.0\modules\optimization\demos\datafit\datafit.dem.sce
```

Of course, the exact path of the file depends on your particular installation and your operating system.

Analyzing the content of these demonstration files is often an efficient solution for solving common problems and to understand particular features.

Another method to find some help is to analyze the source code of Scilab itself (Scilab is indeed open-source!). For example, the `derivative` function is located in

```
<path>\scilab-5.2.0\modules\optimization\macros\derivative.sci
```

Most of the time, Scilab macros are very well written, taking care of all possible combinations of input and output arguments and many possible values of the input arguments. Often, difficult numerical problems are solved in these scripts so that they provide a deep source of inspiration for developing your own scripts.

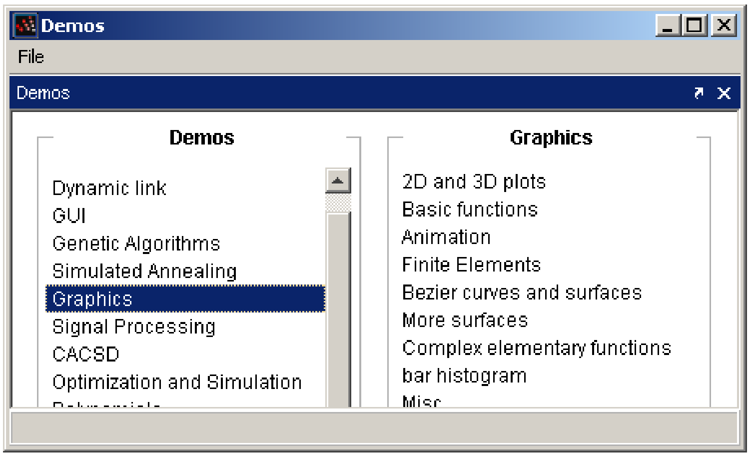

Figure 3: Scilab demos window.

2 Getting started

In this section, we make our first steps with Scilab and present some simple tasks we can perform with the interpreter.

There are several ways of using Scilab and the following paragraphs present three methods:

- using the console in interactive mode,
- using the `exec` function against a file,
- using *batch* processing.

2.1 The console

The first way is to use Scilab interactively, by typing commands in the console, analyzing Scilab result, continuing this process until the final result is computed. This document is designed so that the Scilab examples which are printed here can be copied into the console. The goal is that the reader can experiment by himself Scilab behavior. This is indeed a good way of understanding the behavior of the program and, most of the time, it allows a quick and smooth way of performing the desired computation.

In the following example, the function `disp` is used in interactive mode to print out the string "Hello World !".

```
-->s="Hello World!"
 s  =
 Hello World!
-->disp(s)
 Hello World!
```

In the previous session, we did not type the characters "`-->`" which is the *prompt*, and which is managed by Scilab. We only type the statement `s="Hello World!"` with our keyboard and then hit the `<Enter>` key. Scilab answer is `s =` and `Hello World!`. Then we type `disp(s)` and Scilab answer is `Hello World!`.

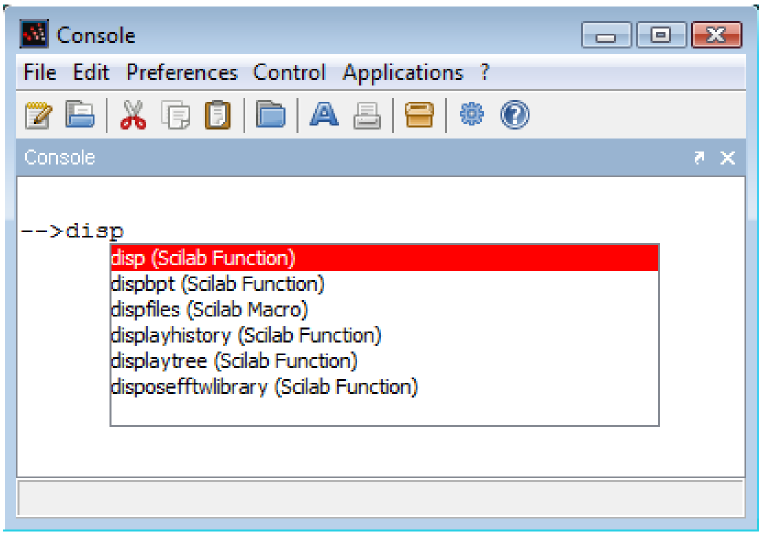

Figure 4: The completion in the console.

When we edit the command, we can use the keyboard, as with a regular editor. We can use the left ← and right → arrows in order to move the cursor on the line and use the <Backspace> and <Suppr> keys in order to fix errors in the text.

In order to get an access to previously executed commands, use the up arrow ↑ key. This allows to browse the previous commands by using the up ↑ and down ↓ arrow keys.

The <Tab> key provides a very convenient completion feature. In the following session, we type the statement `disp` in the console.

```
-->disp
```

Then we can type on the <Tab> key, which makes a list appear in the console, as presented in the figure 4. Scilab displays a listbox, where items correspond to all functions which begin with the letters "disp". We can then use the up and down arrow keys to select the function we want.

The auto-completion works with functions, variables, files and graphic handles and makes the development of scripts easier and faster.

2.2 The editor

Scilab version 5.2 provides a new editor which allows to edit scripts easily. The figure 5 presents the editor while it is editing the previous "Hello World!" example.

The editor can be accessed from the menu of the console, under the *Applications* > *Editor* menu, or from the console, as presented in the following session.

```
-->editor()
```

This editor allows to manage several files at the same time, as presented in the figure 5, where we edit five files at the same time.

There are many features which are worth to mention in this editor. The most commonly used features are under the *Execute* menu.

- *Load into Scilab* allows to execute the statements in the current file, as if we did a copy and paste. This implies that the statements which do not end with a ";" character will produce

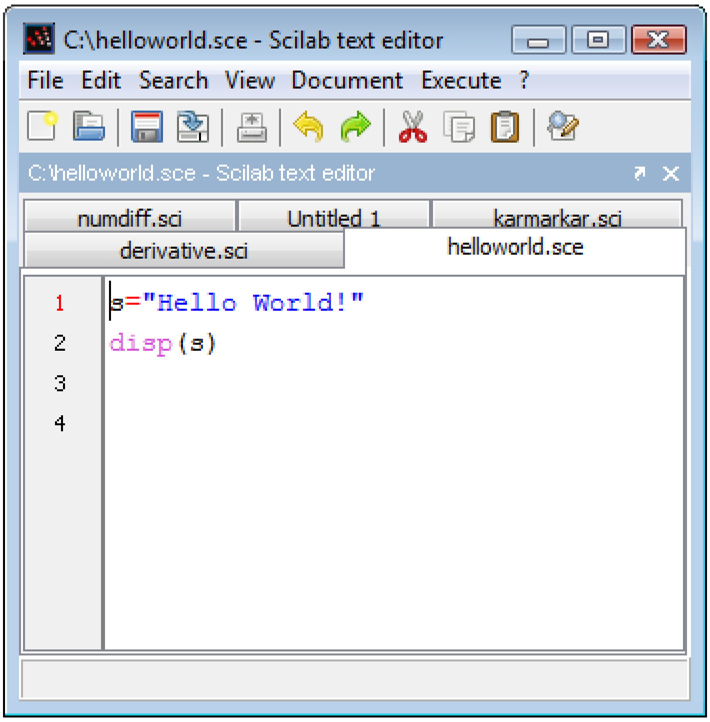

Figure 5: The editor.

an output in the console.

- *Evaluate Selection* allows to execute the statements which are currently selected.
- *Execute File Into Scilab* allows to execute the file, as if we used the `exec` function. The results which are produced in the console are only those which are associated with printing functions, such as `disp` for example.

We can also select a few lines in the script, right click (or Cmd+Click under Mac), and get the context menu which is presented in figure 6.

The *Edit* menu provides a very interesting feature, commonly known as a "pretty printer" in most languages. This is the *Edit > Correct Indentation* feature, which automatically indents the current selection. This feature is extremelly convenient, as it allows to format algorithms so that the `if`, `for` and other structured blocks can be easy to analyze.

The editor provides a fast access to the inline help. Indeed, assume that we have selected the `disp` statement, as presented in the figure 7. When we right-click in the editor, we get the context menu, where the *Help about "disp"* entry allows to open the help associated with the `disp` function.

2.3 Docking

The graphics in Scilab version 5 has been updated so that many components are now based on Java. This has a number of advantages, including the possibility to manage docking windows.

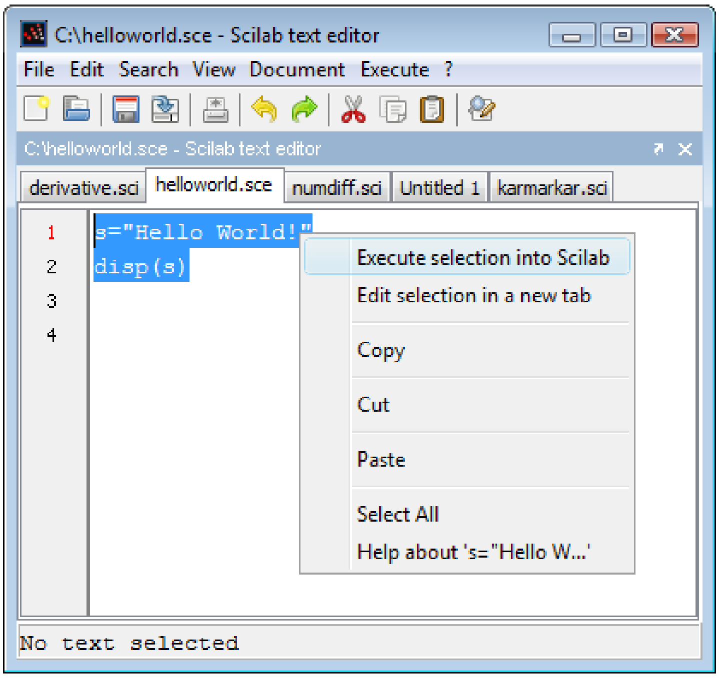

Figure 6: Context menu in the editor.

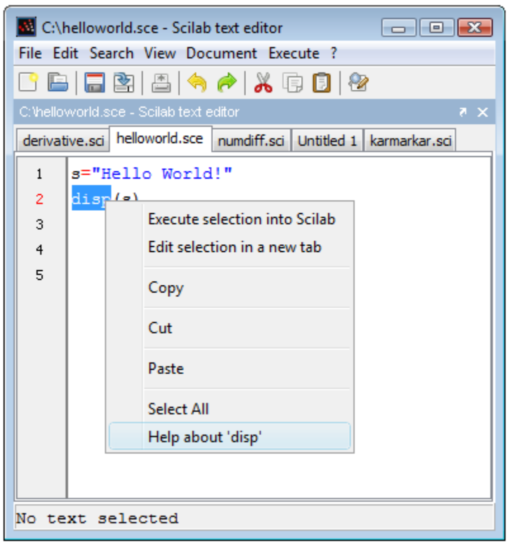

Figure 7: Context help in the editor.

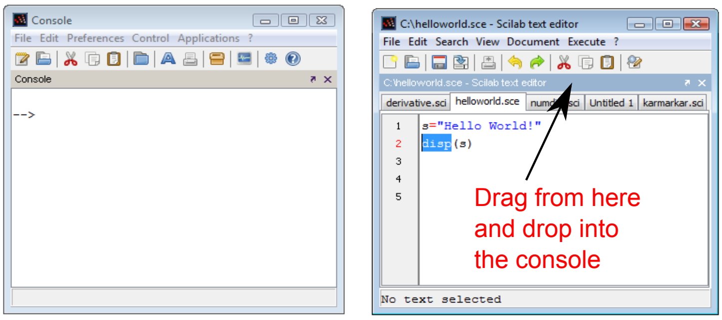

Figure 8: The title bar in the source window. In order to dock the editor into the console, drag and drop the title bar of the editor into the console.

The docking system uses Flexdock [6], an open-source project providing a Swing docking framework. Assume that we have both the console and the editor opened in our environment, as presented in figure 8. It might be annoying to manage two windows, because one may hide the other, so that we constantly have to move them around in order to actually see what happens.

The Flexdock system allows to drag and drop the editor into the console, so that we finally have only one window, with several sub-windows. All Scilab windows are dockable, including the console, the editor, the help and the plotting windows. In the figure 9, we present a situation where we have docked four windows into the console window.

In order to dock one window into another window, we must drag and drop the source window into the target window. To do this, we left-click on the title bar of the docking window, as indicated in figure 8. Before releasing the click, let us move the mouse over the target window and notice that a window, surrounded by dotted lines is displayed. This "fantom" window indicates the location of the future docked window. We can choose this location, which can be on the top, the bottom, the left or the right of the target window. Once we have chosen the target location, we release the click, which finally moves the source window into the target window, as in the figure 9.

We can also release the source window *over* the target window, which creates tabs, as in the figure 10.

2.4 Using `exec`

When several commands are to be executed, it may be more convenient to write these statements into a file with Scilab editor. To execute the commands located in such a file, the `exec` function can be used, followed by the name of the script. This file generally has the extension .sce or .sci, depending on its content:

- files having the .sci extension are containing Scilab functions and executing them loads the functions into Scilab environment (but does not execute them),
- files having the .sce extension are containing both Scilab functions and executable statements.

Executing a .sce file has generally an effect such as computing several variables and displaying the results in the console, creating 2D plots, reading or writing into a file, etc...

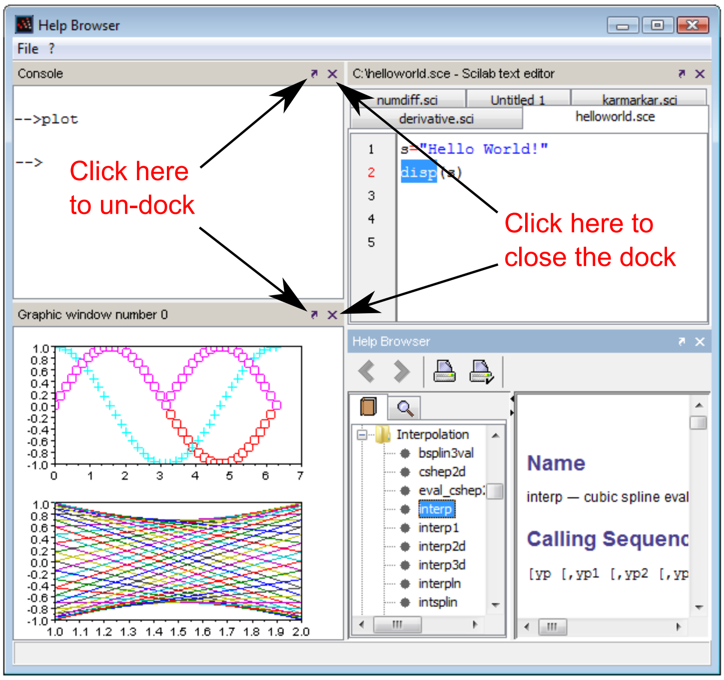

Figure 9: Actions in the title bar of the docking window. The round arrow in the title bar of the window allows to undock the window. The cross allows to close the window.

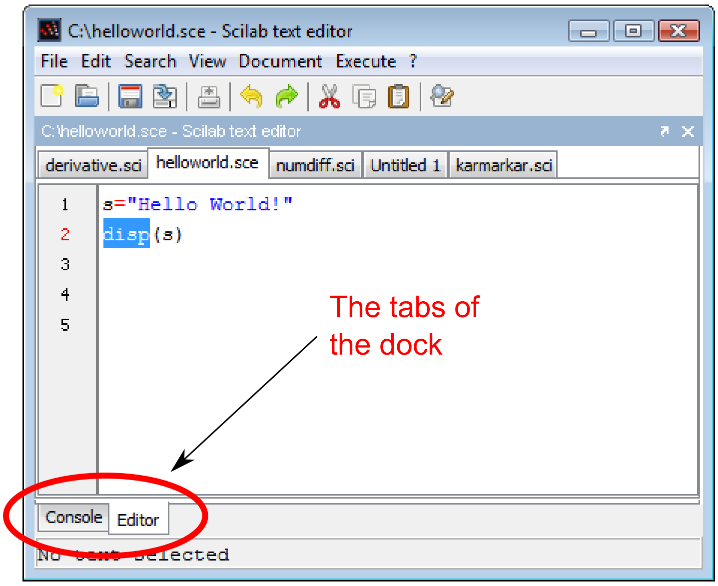

Figure 10: Docking tabs.

Assume that the content of the file *myscript.sce* is the following.

```
disp("Hello World !")
```

In the Scilab console, we can use the *exec* function to execute the content of this script.

```
-->exec("myscript.sce")
-->disp("Hello World !")
 Hello World !
```

In practical situations such as debugging a complicated algorithm, the interactive mode is most of the time used with a sequence of calls to the `exec` and `disp` functions.

2.5 Batch processing

Another way of using Scilab is from the command line. Several command line options are available and are presented in figure 11. Whatever the operating system is, binaries are located in the directory *scilab-5.2.0/bin*. Command line options must be appended to the binary for the specific platform, as described below. The *-nw* option allows to disable the display of the console. The *-nwni* option allows to launch the non-graphics mode: in this mode, the console is not displayed and plotting functions are disabled (using them will generate an error).

- Under Windows, two binary executable are provided. The first executable is *WScilex.exe*, the usual, graphics, interactive console. This executable corresponds to the icon which is available on the desktop after the installation of Scilab. The second executable is *Scilex.exe*, the non-graphics console. With the *Scilex.exe* executable, the Java-based console is not loaded and the Windows terminal is directly used. The *Scilex.exe* program is sensitive to the *-nw* and *-nwni* options.
- Under Linux, the *scilab* script provides options which allow to configure its behavior. By default, the graphics mode is launched. The *scilab* script is sensitive to the *-nw* and *-nwni* options. There are two extra executables on Linux: *scilab-cli* and *scilab-adv-cli*. The *scilab-adv-cli* executable is equivalent to the *-nw* option, while the *scilab-cli* is equivalent to the *-nwni* option[5].
- Under Mac OS, the behavior is similar to the Linux platform.

In the following Windows session, we launch the *Scilex.exe* program with the *-nwni* option. Then we run the `plot` function in order to check that this function is not available in the non-graphics mode.

```
D:\Programs\scilab-5.2.0\bin>Scilex.exe -nwni
        ___________________________________________
                       scilab-5.2.0
                 Consortium Scilab (DIGITEO)
               Copyright (c) 1989-2009 (INRIA)
               Copyright (c) 1989-2007 (ENPC)
        ___________________________________________
Startup execution:
  loading initial environment
-->plot()
       !--error 4
Undefined variable: plot
```

-e instruction	execute the Scilab instruction given in instruction
-f file	execute the Scilab script given in the file
-l lang	setup the user language 'fr' for french and 'en' for english (default is 'en')
-mem N	set the initial stacksize.
-ns	if this option is present, the startup file scilab.start is not executed.
-nb	if this option is present, then Scilab welcome banner is not displayed.
-nouserstartup	don't execute user startup files SCIHOME/.scilab or SCIHOME/scilab.ini.
-nw	start Scilab as command line with advanced features (e.g., graphics).
-nwni	start Scilab as command line without advanced features.
-version	print product version and exit.

Figure 11: Scilab command line options.

The most useful command line option is the -f option, which allows to execute the commands from a given file, a method generally called *batch* processing. Assume that the content of the file *myscript2.sce* is the following, where the `quit` function is used to exit from Scilab.

```
disp("Hello World !")
quit()
```

The default behavior of Scilab is to wait for new user input: this is why the `quit` command is used, so that the session terminates. To execute the demonstration under Windows, we created the directory "C:\scripts" and wrote the statements in the file *C:\scripts\myscript2.sce*. The following session, executed from the MS Windows terminal, shows how to use the -f option to execute the previous script. Notice that we used the absolute path of the *Scilex.exe* executable.

```
C:\scripts>D:\Programs\scilab-5.2.0\bin\Scilex.exe -f myscript2.sce
          ___________________________________________
                       scilab-5.2.0
                  Consortium Scilab (DIGITEO)
                Copyright (c) 1989-2009 (INRIA)
                Copyright (c) 1989-2007 (ENPC)
          ___________________________________________
Startup execution:
  loading initial environment
 Hello World !
C:\scripts>
```

Any line which begins with the two characters "//" is considered by Scilab as a comment and is ignored. To check that Scilab stays by default in interactive mode, we comment out the `quit` statement with the "//" syntax, as in the following script.

```
disp("Hello World !")
//quit()
```

If we type the "scilex -f myscript2.sce" command in the terminal, Scilab waits now for user input, as expected. To exit, we interactively type the `quit()` statement in the terminal.

3 Basic elements of the language

Scilab is an interpreted language, which means that it allows to manipulate variables in a very dynamic way. In this section, we present the basic features of the language, that is, we show how to create a real variable, and what elementary mathematical functions can be applied to a real variable. If Scilab provided only these features, it would only be a super desktop calculator. Fortunately, it is a lot more and this is the subject of the remaining sections, where we will show how to manage other types of variables, that is booleans, complex numbers, integers and strings.

It seems strange at first, but it is worth to state it right from the start: *in Scilab, everything is a matrix*. To be more accurate, we should write: *all real, complex, boolean, integer, string and polynomial variables are matrices*. Lists and other complex data structures (such as tlists and mlists) are not matrices (but can contain matrices). These complex data structures will not be presented in this document.

This is why we could begin by presenting matrices. Still, we choose to present basic data types first, because Scilab matrices are in fact a special organization of these basic building blocks.

In Scilab, we can manage real and complex numbers. This always leads to some confusion if the context is not clear enough. In the following, when we write *real variable*, we will refer to a variable which content is not complex. Complex variables will be covered in section 3.7 as a special case of real variables. In most cases, real variables and complex variables behave in a very similar way, although some extra care must be taken when complex data is to be processed. Because it would make the presentation cumbersome, we simplify most of the discussions by considering only real variables, taking extra care with complex variables only when needed.

3.1 Creating real variables

In this section, we create real variables and perform simple operations with them.

Scilab is an interpreted language, which implies that there is no need to declare a variable before using it. Variables are created at the moment where they are first set.

In the following example, we create and set the real variable x to 1 and perform a multiplication on this variable. In Scilab, the "=" operator means that we want to set the variable on the left hand side to the value associated to the right hand side (it is not the comparison operator, which syntax is associated to the "==" operator).

```
-->x=1
 x  =
    1.
-->x = x * 2
 x  =
    2.
```

The value of the variable is displayed each time a statement is executed. That behavior can be suppressed if the line ends with the semicolon ";" character, as in the following example.

```
-->y=1;
-->y=y*2;
```

All the common algebraic operators presented in figure 12 are available in Scilab. Notice that the power operator is represented by the hat "^" character so that computing x^2 in Scilab is performed by the "x^2" expression or equivalently by the "x**2" expression. The single quote operator " ' " will be presented in more depth in section 3.7, which presents complex numbers.

$+$	addition
-	substraction
$*$	multiplication
/	right division i.e. $x/y = xy^{-1}$
\	left division i.e. $x\backslash y = x^{-1}y$
ˆ	power i.e. x^y
$**$	power (same as ˆ)
'	transpose conjugate

Figure 12: Scilab mathematical elementary operators.

3.2 Variable name

Variable names may be as long as the user wants, but only the first 24 characters are taken into account in Scilab. For consistency, we should consider only variable names which are not made of more than 24 characters. All ASCII letters from "a" to "z", from "A" to "Z" and from "0" to "9" are allowed, with the additional letters "%", "_", "#", "!", "$", "?". Notice though that variable names which first letter is "%" have a special meaning in Scilab, as we will see in section 3.5, which presents the mathematical pre-defined variables.

Scilab is case sensitive, which means that upper and lower case letters are considered to be different by Scilab. In the following script, we define the two variables `A` and `a` and check that these two variables are considered to be different by Scilab.

```
-->A = 2
 A  =
    2.
-->a = 1
 a  =
    1.
-->A
 A  =
    2.
-->a
 a  =
    1.
```

3.3 Comments and continuation lines

Any line which begins with two slashes "//" is considered by Scilab as a comment and is ignored. There is no possibility to comment out a block of lines, such as with the "/* ... */" comments in the C language.

When an executable statement is too long to be written on a single line, the second line and above are called continuation lines. In Scilab, any line which ends with two dots is considered to be the start of a new continuation line. In the following session, we give examples of Scilab comments and continuation lines.

```
-->// This is my comment.
-->x=1..
-->+2..
-->+3..
```

acos	acosd	acosh	acoshm	acosm	acot	acotd	acoth
acsc	acscd	acsch	asec	asecd	asech	asin	asind
asinh	asinhm	asinm	atan	atand	atanh	atanhm	atanm
cos	cosd	cosh	coshm	cosm	cotd	cotg	coth
cothm	csc	cscd	csch	sec	secd	sech	sin
sinc	sind	sinh	sinhm	sinm	tan	tand	tanh
tanhm	tanm						

Figure 13: Scilab mathematical elementary functions: trigonometry.

exp	expm	log	log10	log1p	log2	logm	max
maxi	min	mini	modulo	pmodulo	sign	signm	sqrt
sqrtm							

Figure 14: Scilab mathematical elementary functions: other functions.

```
-->+4
 x  =
    10.
```

3.4 Elementary mathematical functions

The tables 13 and 14 present a list of elementary mathematical functions. Most of these functions take one input argument and return one output argument. These functions are vectorized in the sense that their input and output arguments are matrices. This allows to compute data with higher performance, without any loop.

In the following example, we use the `cos` and `sin` functions and check the equality $\cos(x)^2 + \sin(x)^2 = 1$.

```
-->x = cos(2)
 x  =
  - 0.4161468
-->y = sin(2)
 y  =
    0.9092974
-->x^2+y^2
 ans  =
    1.
```

3.5 Pre-defined mathematical variables

In Scilab, several mathematical variables are pre-defined variables, which name begins with a percent "%" character. The variables which have a mathematical meaning are summarized in figure 15.

In the following example, we use the variable `%pi` to check the mathematical equality $\cos(x)^2 + \sin(x)^2 = 1$.

%i	the imaginary number i
%e	Euler's constant e
%pi	the mathematical constant π

Figure 15: Pre-defined mathematical variables.

a&b	logical and
a|b	logical or
∼a	logical not
a==b	true if the two expressions are equal
a∼=b or a<>b	true if the two expressions are different
a<b	true if a is lower than b
a>b	true if a is greater than b
a<=b	true if a is lower or equal to b
a>=b	true if a is greater or equal to b

Figure 16: Comparison operators.

```
-->c=cos(%pi)
 c  =
  - 1.
-->s=sin(%pi)
 s  =
    1.225D-16
-->c^2+s^2
 ans  =
    1.
```

The fact that the computed value of $\sin(\pi)$ is *not exactly* equal to 0 is a consequence of the fact that Scilab stores the real numbers with floating point numbers, that is, with limited precision.

3.6 Booleans

Boolean variables can store true or false values. In Scilab, true is written with "%t" or "%T" and false is written with "%f" or "%F". The figure 16 presents the several comparison operators which are available in Scilab. These operators return boolean values and take as input arguments all basic data types (i.e. real and complex numbers, integers and strings).

In the following example, we perform some algebraic computations with Scilab booleans.

```
-->a=%T
 b  =
  T
-->b = ( 0 == 1 )
 a  =
  F
-->a&b
 ans  =
  F
```

real	real part
imag	imaginary part
imult	multiplication by i, the imaginary unitary
isreal	returns true if the variable has no complex entry

Figure 17: Scilab complex numbers elementary functions.

3.7 Complex numbers

Scilab provides complex numbers, which are stored as pairs of floating point numbers. The predefined variable "%i" represents the mathematical imaginary number i which satisfies $i^2 = -1$. All elementary functions previously presented before, such as `sin` for example, are overloaded for complex numbers. This means that, if their input argument is a complex number, the output is a complex number. The figure 17 presents functions which allow to manage complex numbers.

In the following example, we set the variable x to $1+i$, and perform several basic operations on it, such as retrieving its real and imaginary parts. Notice how the single quote operator, denoted by " ' ", is used to compute the conjugate of a complex number.

```
-->x= 1+%i
 x  =
    1. + i
-->isreal(x)
 ans  =
  F
-->x'
 ans  =
    1. - i
-->y=1-%i
 y  =
    1. - i
-->real(y)
 ans  =
    1.
-->imag(y)
 ans  =
  - 1.
```

We finally check that the equality $(1+i)(1-i) = 1 - i^2 = 2$ is verified by Scilab.

```
-->x*y
 ans  =
    2.
```

3.8 Strings

Strings can be stored in variables, provided that they are delimited by double quotes " `"` ". The concatenation operation is available from the "+" operator. In the following Scilab session, we define two strings and then concatenate them with the "+" operator.

```
-->x = "foo"
 x  =
 foo
```

```
-->y="bar"
 y  =
 bar
-->x+y
 ans  =
 foobar
```

They are many functions which allow to process strings, including regular expressions. We will not give further details about this topic in this document.

3.9 Dynamic type of variables

When we create and manage variables, Scilab allows to change the type of a variable dynamically. This means that we can create a real value, and then put a string variable in it, as presented in the following session.

```
-->x=1
 x  =
    1.
-->x+1
 ans  =
    2.
-->x="foo"
 x  =
 foo
-->x+"bar"
 ans  =
 foobar
```

We emphasize here that Scilab is not a typed language, that is, we do not have to declare the type of a variable before setting its content. Moreover, the type of a variable can change during the life of the variable.

4 Matrices

In the Scilab language, matrices play a central role. In this section, we introduce Scilab matrices and present how to create and query matrices. We also analyze how to access to elements of a matrix, either element by element, or by higher level operations.

4.1 Overview

In Scilab, the basic data type is the matrix, which is defined by:

- the number of rows,
- the number of columns,
- the type of data.

The data type can be real, integer, boolean, string and polynomial. When two matrices have the same number of rows and columns, we say that the two matrices have the same *shape*.

In Scilab, vectors are a particular case of matrices, where the number of rows (or the number of columns) is equal to 1. Simple scalar variables do not exist in Scilab: a scalar variable is a matrix with 1 row and 1 column. This is why in this chapter, when we analyze the behavior of Scilab matrices, there is the same behavior for row or column vectors (i.e. $n \times 1$ or $1 \times n$ matrices) as well as scalars (i.e. 1×1 matrices).

It is fair to say that Scilab was designed mainly for matrices of *real* variables. This allows to perform linear algebra operations with a high level language.

By design, Scilab was created to be able to perform matrices operations as fast as possible. The building block for this feature is that Scilab matrices are stored in an internal data structure which can be managed at the interpreter level. Most basic linear algebra operations, such as addition, substraction, transpose or dot product are performed by a compiled, optimized, source code. These operations are performed with the common operators ”+”, ”-”, ”*” and the single quote ” ' ”, so that, at the Scilab level, the source code is both simple and fast.

With these high level operators, most matrix algorithms do not require to use loops. In fact, a Scilab script which performs the same operations with loops is typically from 10 to 100 times slower. This feature of Scilab is known as the *vectorization*. In order to get a fast implementation of a given algorithm, the Scilab developper should always use high level operations, so that each statement process a matrix (or a vector) instead of a scalar.

More complex tasks of linear algebra, such as the resolution of systems of linear equations $Ax = b$, various decompositions (for example Gauss partial pivotal $PA = LU$), eigenvalue/eigenvector computations, are also performed by compiled and optimized source codes. These operations are performed by common operators like the slash ”/” or backslash ”\” or with functions like `spec`, which computes eigenvalues and eigenvectors.

4.2 Create a matrix of real values

There is a simple and efficient syntax to create a matrix with given values. The following is the list of symbols used to define a matrix:

- square brackets ”[” and ”]” mark the beginning and the end of the matrix,
- commas ”,” separate the values on different columns,
- semicolons ”;” separate the values of different rows.

The following syntax can be used to define a matrix, where blank spaces are optional (but make the line easier to read) and ”...” are designing intermediate values:

```
A = [a11 , a12 , ... , a1n; a21 , a22 , ... , a2n; ...; an1 , an2 , ... , ann].
```

In the following example, we create a 2×3 matrix of real values.

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

A simpler syntax is available, which does not require to use the comma and semicolon characters. When creating a matrix, the blank space separates the columns while the new line separates the rows, as in the following syntax:

eye	identity matrix
linspace	linearly spaced vector
ones	matrix made of ones
zeros	matrix made of zeros
testmatrix	generate some particular matrices
grand	random number generator
rand	random number generator

Figure 18: Functions which generate matrices.

```
A = [a11 a12 ... a1n
a21 a22 ... a2n
...
an1 an2 ... ann]
```

This allows to lighten considerably the management of matrices, as in the following session.

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

The previous syntax for matrices is useful in the situations where matrices are to be written into data files, because it simplifies the human reading (and checking) of the values in the file, and simplifies the reading of the matrix in Scilab.

Several Scilab commands allow to create matrices from a given size, i.e. from a given number of rows and columns. These functions are presented in figure 18. The most commonly used are `eye`, `zeros` and `ones`. These commands takes two input arguments, the number of rows and columns of the matrix to generate.

```
-->A = ones(2,3)
 A  =
    1.     1.     1.
    1.     1.     1.
```

4.3 The empty matrix []

An empty matrix can be created by using empty square brackets, as in the following session, where we create a 0×0 matrix.

```
-->A=[]
 A  =
     []
```

This syntax allows to delete the content of a matrix, so that the associated memory is deleted.

```
-->A = ones(100,100);
-->A = []
 A  =
     []
```

size	size of objects
matrix	reshape a vector or a matrix to a different size matrix
resize_matrix	create a new matrix with a different size

Figure 19: Functions which query or modify matrices.

4.4 Query matrices

The functions in figure 19 allow to query or update a matrix.

The `size` function returns the two output arguments `nr` and `nc`, which are the number of rows and the number of columns.

```
-->A = ones(2,3)
 A  =
    1.    1.    1.
    1.    1.    1.
-->[nr,nc]=size(A)
 nc  =
    3.
 nr  =
    2.
```

The `size` function is of important practical value when we design a function, since the processing that we must perform on a given matrix may depend on its shape. For example, to compute the norm of a given matrix, different algorithms may be use depending if the matrix is either a column vector with size $nr \times 1$ and $nr > 0$, a row vector with size $1 \times nc$ and $nc > 0$, or a general matrix with size $nr \times nc$ and $nr, nc > 0$.

The `size` function has also the following syntax

```
nr = size( A , sel )
```

which allows to get only the number of rows or the number of columns and where `sel` can have the following values

- `sel=1` or `sel="r"`, returns the number of rows,
- `sel=2` or `sel="c"`, returns the number of columns.
- `sel="*"`, returns the total number of elements, that is, the number of columns times the number of rows.

In the following session, we use the `size` function in order to compute the total number of elements of a matrix.

```
-->A = ones(2,3)
 A  =
    1.    1.    1.
    1.    1.    1.
-->size(A,"*")
 ans  =
    6.
```

4.5 Accessing the elements of a matrix

There are several methods to access the elements of a matrix `A`:

- the whole matrix, with the `A` syntax,
- element by element with the `A(i,j)` syntax,
- a range of index values with the colon ":" operator.

The colon operator will be reviewed in the next section.

To make a global access to all the elements of the matrix, the simple variable name, for example `A`, can be used. All elementary algebra operations are available for matrices, such as the addition with `+`, subtraction with `-`, provided that the two matrices have the same size. In the following script, we add all the elements of two matrices.

```
-->A = ones(2,3)
 A  =
    1.    1.    1.
    1.    1.    1.
-->B =  2 * ones(2,3)
 B  =
    2.    2.    2.
    2.    2.    2.
-->A+B
 ans  =
    3.    3.    3.
    3.    3.    3.
```

One element of a matrix can be accessed directly with the `A(i,j)` syntax, provided that i and j are valid index values.

We emphasize that, by default, the first index of a matrix is 1. This contrasts with other languages, such as the C language for instance, where the first index is 0. For example, assume that `A` is an $nr \times nc$ matrix, where nr is the number of rows and nc is the number of columns. Therefore, the value `A(i,j)` has a sense only if the index values i and j satisfy $1 \leq i \leq nr$ and $1 \leq j \leq nc$. If the index values are not valid, an error is generated, as in the following session.

```
-->A = ones(2,3)
 A  =
    1.    1.    1.
    1.    1.    1.
-->A(1,1)
 ans  =
    1.
-->A(12,1)
          !--error 21
Invalid index.
-->A(0,1)
        !--error 21
Invalid index.
```

Direct access to matrix elements with the `A(i,j)` syntax should be used only when no other higher level Scilab commands can be used. Indeed Scilab provides many features which allow to produce simpler and faster computations, based on vectorization. One of these features is the colon ":" operator, which is very important in practical situations.

4.6 The colon ":" operator

The simplest syntax of the colon operator is the following:

```
v = i:j
```

where `i` is the starting index and `j` is the ending index with $i \leq j$. This creates the vector $v = (i, i+1, \ldots, j)$. In the following session, we create a vector of index values from `2` to `4` in one statement.

```
-->v = 2:4
 v  =
    2.    3.    4.
```

The complete syntax allows to configure the increment used when generating the index values, i.e. the *step*. The complete syntax for the colon operator is

```
v = i:s:j
```

where `i` is the starting index, `j` is the ending index and `s` is the step. This command creates the vector $v = (i, i+s, i+2s, \ldots, i+ns)$ where n is the greatest integer such that $i+ns \leq j$. If s divides $j-i$, then the last index in the vector of index values is j. In other cases, we have $i + ns < j$. While in most situations, the step `s` is positive, it might also be negative.

In the following session, we create a vector of increasing index values from `3` to `10` with a step equal to `2`.

```
-->v = 3:2:10
 v  =
    3.    5.    7.    9.
```

Notice that the last value in the vector `v` is $i + ns = 9$, which is smaller that $j = 10$.

In the following session, we present two examples where the step is negative. In the first case, the colon operator generates decreasing index values from `10` to `4`. In the second example, the colon operator generates an empty matrix because there are no values lower than 3 and greater than 10.

```
-->v = 10:-2:3
 v  =
    10.    8.    6.    4.
-->v = 3:-2:10
 v  =
      []
```

With a vector of index values, we can access to the elements of a matrix in a given range, as with the following simplified syntax

```
A(i:j,k:l)
```

where `i,j,k,l` are starting and ending index values. The complete syntax is `A(i:s:j,k:t:l)`, where `s` and `t` are the steps.

For example, suppose that `A` is a 4×5 matrix, and that we want to access to the elements $a_{i,j}$ for $i = 1, 2$ and $j = 3, 4$. With the Scilab language, this can be done in just one statement, by using the syntax `A(1:2,3:4)`, as showed in the following session.

```
-->A = testmatrix("hilb",5)
 A  =

    25.    - 300.      1050.    - 1400.      630.
```

```
  - 300.      4800.    - 18900.      26880.    - 12600.
    1050.   - 18900.     79380.    - 117600.     56700.
  - 1400.     26880.   - 117600.     179200.   - 88200.
    630.    - 12600.     56700.    - 88200.      44100.
-->A(1:2,3:4)
 ans  =
    1050.   - 1400.
  - 18900.    26880.
```

In some circumstances, it may happen that the index values are the result of a computation. For example, the algorithm may be based on a loop where the index values are updated regularly. In these cases, the syntax

```
A(vi,vj),
```

where `vi,vj` are vectors of index values, can be used to designate the elements of `A` whose subscripts are the elements of `vi` and `vj`. That syntax is illustrated in the following example.

```
-->A = testmatrix("hilb",5)
 A  =

    25.     - 300.       1050.     - 1400.       630.
  - 300.      4800.    - 18900.      26880.    - 12600.
    1050.   - 18900.     79380.    - 117600.     56700.
  - 1400.     26880.   - 117600.     179200.   - 88200.
    630.    - 12600.     56700.    - 88200.      44100.
-->vi=1:2
 vi  =
    1.    2.
-->vj=3:4
 vj  =
    3.    4.
-->A(vi,vj)
 ans  =
    1050.   - 1400.
  - 18900.    26880.
-->vi=vi+1
 vi  =
    2.    3.
-->vj=vj+1
 vj  =
    4.    5.
-->A(vi,vj)
 ans  =
    26880.   - 12600.
  - 117600.    56700.
```

There are many variations on this syntax, and the figure 20 presents some of the possible combinations.

For example, in the following session, we use the colon operator in order to interchange two rows of the matrix `A`.

```
-->A = testmatrix("hilb",3)
 A  =
    9.    - 36.      30.
  - 36.     192.   - 180.
    30.   - 180.     180.
```

A	the whole matrix
A(:,:)	the whole matrix
A(i:j,k)	the elements at rows from i to j, at column k
A(i,j:k)	the elements at row i, at columns from j to k
A(i,:)	the row i
A(:,j)	the column j

Figure 20: Access to a matrix with the colon ":" operator. We make the assumption that `A` is a $nr \times nc$ matrix.

A(i,\$)	the element at row i, at column nr
A(\$,j)	the element at row nr, at column j
A(\$-i,\$-j)	the element at row $nr - i$, at column $nc - j$

Figure 21: Access to a matrix with the dollar "$" operator. The "$" operator signifies "the last index".

```
-->A([1 2],:) = A([2 1],:)
 A  =
  - 36.     192.   - 180.
    9.    - 36.      30.
    30.   - 180.     180.
```

We could also interchange the columns of the matrix `A` with the statement `A(:,[3 1 2])`.

We have analyzed in this section several practical use of the colon operator. Indeed, this operator is used in many scripts where performance matters since it allows to access to many elements of a matrix in just one statement. This is associated with the *vectorization* of scripts, a subject which is central in the Scilab language and is reviewed throughout this document.

4.7 The dollar "$" operator

The dollar `$` operator allows to reference elements *from the end* of the matrix, instead of the usual reference *from the start*. The special operator `$` signifies "the index corresponding to the last" row or column, depending on the context. This syntax is associated to an algebra, so that the index `$-i` corresponds to the index $\ell - i$, where ℓ is the number of corresponding rows or columns. Various uses of the dollar operator are presented in figure 21.

In the following example, we consider a 3×3 matrix and we access to the element `A(2,1) = A(n-1,m-2) = A($-1,$-2)` because $nr = 3$ and $nc = 3$.

```
-->A=testmatrix("hilb",3)
 A  =
    9.    - 36.      30.
  - 36.     192.   - 180.
    30.   - 180.     180.
-->A($-1,$-2)
 ans  =
  - 36.
```

The dollar `$` operator allows to add elements dynamically at the end of matrices. In the following session, we add a row at the end of the Hilbert matrix.

```
-->A($+1,:) = [1 2 3]
 A  =
    9.   - 36.     30.
  - 36.    192.  - 180.
    30.  - 180.    180.
    1.     2.      3.
```

4.8 Low-level operations

All common algebra operators, such as `+`, `-`, `*` and `/`, are available with real matrices. In the next sections, we focus on the exact signification of these operators, so that many sources of confusion are avoided.

The rules for the "+" and "-" operators are directly applied from the usual algebra. In the following session, we add two 2×2 matrices.

```
-->A = [1 2
-->3 4]
 A  =
    1.    2.
    3.    4.
-->B=[5 6
-->7 8]
 B  =
    5.    6.
    7.    8.
-->A+B
 ans  =
    6.     8.
    10.    12.
```

When we perform an addition of two matrices, if one operand is a 1×1 matrix (i.e., a scalar), the addition is performed element by element. This feature is shown in the following session.

```
-->A = [1 2
-->3 4]
 A  =
    1.    2.
    3.    4.
-->A + 1
 ans  =
    2.    3.
    4.    5.
```

The addition is possible only if the two matrices have a shape which match. In the following session, we try add a 2×3 matrix with a 2×2 matrix and check that this is not possible.

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

+	addition	.+	elementwise addition
-	substraction	.-	elementwise substraction
*	multiplication	.*	elementwise multiplication
/	right division	./	elementwise right division
\	left division	\	elementwise left division
^ or **	power i.e. x^y	.^	elementwise power
'	transpose and conjugate	.'	transpose (but not conjugate)

Figure 22: Matrix operators and elementwise operators.

```
-->A+B
    !--error 8
Inconsistent addition.
```

Elementary operators which are available for matrices are presented in figure 22. The Scilab language provides two division operators, that is,the right division / and the left division \. The right division is so that $X = A/B = AB^{-1}$ is the solution of $XB = A$. The left division is so that $X = A\backslash B = A^{-1}B$ is the solution of $AX = B$. The left division $A\backslash B$ computes the solution of the associated least square problem if A is not a square matrix.

The figure 22 separates the operators which treat the matrices as a whole and the elementwise operators, which are presented in the next section.

4.9 Elementwise operations

If a dot "." is written before an operator, it is associated with an elementwise operator, i.e. the operation is performed element-by-element. For example, with the usual multiplication operator "*", the content of the matrix C=A*B is $c_{ij} = \sum_{k=1,n} a_{ik}b_{kj}$. With the elementwise multiplication ".*" operator, the content of the matrix C=A.*B is $c_{ij} = a_{ij}b_{ij}$.

In the following session, two matrices are multiplied with the "*" operator and then with the elementwise ".*" operator, so that we can check that the result is different.

```
-->A = ones(2,2)
 A  =
    1.    1.
    1.    1.
-->B = 2 * ones(2,2)
 B  =
    2.    2.
    2.    2.
-->A*B
 ans  =
    4.    4.
    4.    4.
-->A.*B
 ans  =
    2.    2.
    2.    2.
```

chol	Cholesky factorization
companion	companion matrix
cond	condition number
det	determinant
inv	matrix inverse
linsolve	linear equation solver
lsq	linear least square problems
lu	LU factors of Gaussian elimination
qr	QR decomposition
rcond	inverse condition number
spec	eigenvalues
svd	singular value decomposition
testmatrix	a collection of test matrices
trace	trace

Figure 23: Some common functions for linear algebra.

4.10 Higher level linear algebra features

In this section, we briefly introduce higher level linear algebra features of Scilab.

Scilab has a complete linear algebra library, which is able to manage both dense and sparse matrices. A complete book on linear algebra would be required to make a description of the algorithms provided by Scilab in this field, and this is obviously out of the scope of this document. The figure 23 presents a list of the most common linear algebra functions.

5 Looping and branching

In this section, we describe how to make conditional statements, that is, we present the `if` statement. We also present Scilab loops, that is, we present the `for` and `while` statements. We also present two main tools to manage loops, that is the interrupt statement `break` and the `continue` statement.

5.1 The `if` statement

The `if` statement allows to perform a statement if a condition is satisfied. The `if` uses a boolean variable to perform its choice: if the boolean is true, then the statement is executed. A condition is closed when the `end` keyword is met. In the following script, we display the string "Hello!" if the condition `%t`, which is always true, is satisfied.

```
if ( %t ) then
  disp("Hello !")
end
```

The previous script produces:

```
 Hello !
```

If the condition is not satisfied, the `else` statement allows to perform an alternative statement, as in the following script.

```
if ( %f ) then
  disp("Hello !")
else
  disp("Goodbye !")
end
```

The previous script produces:

```
 Goodbye !
```

In order to get a boolean, any comparison operator can be used, e.g. "==", ">", etc... or any function which returns a boolean. In the following session, we use the == operator to display the message "Hello !".

```
i = 2
if ( i == 2 ) then
  disp("Hello !")
else
  disp("Goodbye !")
end
```

It is important not to use the "=" operator in the condition, i.e. we must not use the statement `if ( i = 2 ) then`. It is an error, since the = the operator allows to set a variable: it is different from the comparison operator ==. In case of an error, Scilab warns us that something wrong happened.

```
-->i = 2
 i  =
    2.
-->if ( i = 2 ) then
Warning: obsolete use of '=' instead of '=='.
         !
-->  disp("Hello !")

 Hello !
-->else
-->  disp("Goodbye !")
-->end
```

When we have to combine several conditions, the `elseif` statement is helpful. In the following script, we combine several `elseif` statements in order to manage various values of the integer `i`.

```
i = 2
if ( i == 1 ) then
  disp("Hello !")
elseif ( i == 2 ) then
  disp("Goodbye !")
elseif ( i == 3 ) then
  disp("Tchao !")
else
  disp("Au Revoir !")
end
```

We can use as many `elseif` statements that we need, and this allows to create as complicated branches as required. But if there are many `elseif` statements required, most of the time that implies that a `select` statement should be used instead.

5.2 The `select` statement

The `select` statement allows to combine several branches in a clear and simple way. Depending on the value of a variable, it allows to perform the statement corresponding to the `case` keyword. There can be as many branches as required.

In the following script, we want to display a string which corresponds to the given integer `i`.

```
i = 2
select i
case 1
  disp("One")
case 2
  disp("Two")
case 3
  disp("Three")
else
  disp("Other")
end
```

The previous script prints out ”Two”, as expected.

The `else` branch is used if all the previous `case` conditions are false.

5.3 The `for` statement

The `for` statement allows to perform loops, i.e. allows to perform a given action several times. Most of the time, a loop is performed over an integer value, which goes from a starting to an ending index value. We will see, at the end of this section, that the `for` statement is in fact much more general, as it allows to loop through the values of a matrix.

In the following Scilab script, we display the value of `i`, from 1 to 5.

```
for i = 1 : 5
  disp(i)
end
```

The previous script produces the following output.

```
    1.
    2.
    3.
    4.
    5.
```

In the previous example, the loop is performed over a matrix of floating point numbers containing integer values. Indeed, we used the colon ”`:`” operator in order to produce the vector of index values `[1 2 3 4 5]`. The following session shows that the statement `1:5` produces all the required integer values into a row vector.

```
-->i = 1:5
 i  =
    1.    2.    3.    4.    5.
```

We emphasize that, in the previous loop, the matrix `1:5` is a matrix of doubles. Therefore, the variable `i` is also a double. This point will be reviewed later in this section, when we will consider the general form of `for` loops.

We can use a more complete form of the colon operator in order to display the odd integers from 1 to 5. In order to do this, we set the step of the colon operator to 2. This is performed by the following Scilab script.

```
for i = 1 : 2 : 5
  disp(i)
end
```

The previous script produces the following output.

```
    1.
    3.
    5.
```

The colon operator can be used to perform *backward* loops. In the following script, we compute the sum of the numbers from 5 to 1.

```
for i = 5 : - 1 : 1
  disp(i)
end
```

The previous script produces the following output.

```
    5.
    4.
    3.
    2.
    1.
```

Indeed, the statement `5:-1:1` produces all the required integers.

```
-->i = 5:-1:1
 i  =
    5.    4.    3.    2.    1.
```

The `for` statement is much more general that what we have previously used in this section. Indeed, it allows to browse through the values of many data types, including row matrices and lists. When we perform a `for` loop over the elements of a matrix, this matrix may be a matrix of doubles, strings, integers or polynomials.

In the following example, we perform a `for` loop over the double values of a row matrix containing $(1.5, e, \pi)$.

```
v = [1.5 exp(1) %pi];
for x = v
  disp(x)
end
```

The previous script produces the following output.

```
    1.5
    2.7182818
    3.1415927
```

We emphasize now an important point about the `for` statement. Anytime we use a `for` loop, we must ask ourselves if a vectorized statement could perform the same computation. There can be a 10 to 100 performance factor between vectorized statements and a `for` loop. Vectorization enables to perform fast computations, even in an interpreted environment like Scilab. This is why the `for` loop should be used only when there is no other way to perform the same computation with vectorized functions.

5.4 The `while` statement

The `while` statement allows to perform a loop while a boolean expression is true. At the beginning of the loop, if the expression is true, the statements in the body of the loop are executed. When the expression becomes false (an event which must occur at certain time), the loop is ended.

In the following script, we compute the sum of the numbers i from 1 to 10 with a `while` statement.

```
s = 0
i = 1
while ( i<= 10 )
  s = s + i
  i = i + 1
end
```

At the end of the algorithm, the values of the variables `i` and `s` are:

```
 s   =
    55.
 i   =
    11.
```

It should be clear that the previous example is just an example for the `while` statement. If we really wanted to compute the sum of the numbers from 1 to 10, we should rather use the `sum` function, as in the following session.

```
-->sum(1:10)
 ans   =
    55.
```

The `while` statement has the same performance issue that the `for` statement. This is why vectorized statements should be considered first, before attempting to design an algorithm based on a `while` loop.

5.5 The `break` and `continue` statements

The `break` statement allows to interrupt a loop. Usually, we use this statement in loops where, if some condition is satisfied, the loops should not be continued further.

In the following example, we use the `break` statement in order to compute the sum of the integers from 1 to 10. When the variable `i` is greater than 10, the loop is interrupted.

```
s = 0
i = 1
while ( %t )
  if ( i > 10 ) then
    break
  end
  s = s + i
  i = i + 1
end
```

At the end of the algorithm, the values of the variables `i` and `s` are:

```
 s   =
    55.
 i   =
    11.
```

The `continue` statement allows to go on to the next loop, so that the statements in the body of the loop are not executed this time. When the `continue` statement is executed, Scilab skips the other statements and goes directly to the `while` or `for` statement and evaluates the next loop.

In the following example, we compute the sum $s = 1 + 3 + 5 + 7 + 9 = 25$. The `modulo(i,2)` function returns 0 if the number i is even. In this situation, the script increments the value of i and use the `continue` statement to go on to the next loop.

```
s = 0
i = 1
while ( i<= 10 )
  if ( modulo ( i , 2 ) == 0 ) then
    i = i + 1
    continue
  else
    s = s + i
    i = i + 1
  end
end
```

If the previous script is executed, the final values of the variables `i` and `s` are:

```
 s  =
    25.
 i  =
    11.
```

As an example of vectorized computation, the previous algorithm can be performed in one function call only. Indeed, the following script uses the `sum` function, combined with the colon operator ":" and produces same the result.

```
s = sum(1:2:10);
```

The previous script has two main advantages over the `while`-based algorithm.

1. The computation makes use of a higher-level language, which is easier to understand for human beings.
2. With large matrices, the `sum`-based computation will be much faster than the `while`-based algorithm.

This is why a careful analysis must be done before developing an algorithm based on a `while` loop.

6 Functions

In this section, we present Scilab functions. We analyze the way to define a new function and the method to load it into Scilab. We present how to create and load a *library*, which is a collection of functions. Since most of Scilab features are provided as functions, we analyze the difference between *macros* and *primitives* and detail how to inquire about Scilab functions. We also present how to manage input and output arguments.

6.1 Overview

Gathering various steps into a reusable function is one of the most common tasks of a Scilab developer. The most simple calling sequence of a function is the following:

```
outvar = myfunction ( invar )
```

where the following list presents the various variables used in the syntax:

- `myfunction` is the name of the function,
- `invar` is the name of the input arguments,
- `outvar` is the name of the output arguments.

The values of the input arguments are not modified by the function, while the values of the output arguments are actually modified by the function.

We have in fact already met several functions in this document. The `sin` function, in the `y=sin(x)` statement, takes the input argument `x` and returns the result in the output argument `y`. In Scilab vocabulary, the input arguments are called the *right hand side* and the output arguments are called the *left hand side*.

Functions can have an arbitrary number of input and output arguments so that the complete syntax for a function which has a fixed number of arguments is the following:

```
[o1 , ... , on] = myfunction ( i1 , ... , in )
```

The input and output arguments are separated by commas ",". Notice that the input arguments are surrounded by opening and closing braces, while the output arguments are surrounded by opening and closing *square* braces.

In the following Scilab session, we show how to compute the *LU* decomposition of the Hilbert matrix. The following session shows how to create a matrix with the `testmatrix` function, which takes two input arguments, and returns one matrix. Then, we use the `lu` function, which takes one input argument and returns two or three arguments depending on the provided output variables. If the third argument `P` is provided, the permutation matrix is returned.

```
-->A = testmatrix("hilb",2)
 A  =
    4.   - 6.
  - 6.     12.
-->[L,U] = lu(A)
 U  =
  - 6.     12.
    0.     2.
 L  =
  - 0.6666667    1.
    1.           0.
-->[L,U,P] = lu(A)
 P  =
    0.     1.
    1.     0.
 U  =
  - 6.     12.
    0.     2.
 L  =
    1.           0.
  - 0.6666667    1.
```

function	opens a function definition
endfunction	closes a function definition
argn	number of input/output arguments in a function call
varargin	variable numbers of arguments in an input argument list
varargout	variable numbers of arguments in an output argument list
fun2string	generates ascii definition of a scilab function
get_function_path	get source file path of a library function
getd	getting all functions defined in a directory
head_comments	display scilab function header comments
listfunctions	properties of all functions in the workspace
macrovar	variables of function

Figure 24: Scilab functions to manage functions.

Notice that the behavior of the `lu` function actually changes when three output arguments are provided: the two rows of the matrix `L` have been swapped. More specifically, when two output arguments are provided, the decomposition $A = LU$ is provided (the statement `A-L*U` allows to check this). When three output arguments are provided, permutations are performed so that the decomposition $PA = LU$ is provided (the statement `P*A-L*U` can be used to check this). In fact, when two output arguments are provided, the permutations are applied on the `L` matrix. This means that the `lu` function knows how many input and output arguments are provided to it, and changes its algorithm accordingly. We will not present in this document how to provide this feature, i.e. a variable number of input or output arguments. But we must keep in mind that this is possible in the Scilab language.

The commands provided by Scilab to manage functions are presented in figure 24. In the next sections, we will present some of the most commonly used commands.

6.2 Defining a function

To define a new function, we use the `function` and `endfunction` Scilab keywords. In the following example, we define the function `myfunction`, which takes the input argument `x`, mutiplies it by 2, and returns the value in the output argument `y`.

```
function y = myfunction ( x )
  y = 2 * x
endfunction
```

There are at least three possibilities to define the previous function in Scilab.

- The first solution is to type the script directly into the console in an interactive mode. Notice that, once the "`function y = myfunction ( x )`" statement has been written and the enter key is typed in, Scilab creates a new line in the console, waiting for the body of the function. When the "`endfunction`" statement is typed in the console, Scilab returns back to its normal edition mode.

- Another solution is available when the source code of the function is provided in a file. This is the most common case, since functions are generally quite long and complicated. We can simply copy and paste the function definition into the console. When the function definition is short (typically, a dozen lines of source code), this way is very convenient. With the editor, this is very easy, thanks to the *Load into Scilab* feature.

- We can also use the `exec` function. Let us consider a Windows system where the previous function is written in the file "`C:\myscripts\examples-functions.sce`". The following session gives an example of the use of `exec` to load the previous function.

  ```
  -->exec("C:\myscripts\examples-functions.sce")
  -->function y = myfunction ( x )
  -->  y = 2 * x
  -->endfunction
  ```

  The `exec` function executes the content of the file as if it were written interactively in the console and displays the various Scilab statements, line after line. The file may contain a lot of source code so that the output may be very long and useless. In these situations, we add the semicolon caracter "`;`" at the end of the line. This is what is performed by the *Execute file into Scilab* feature of the editor.

  ```
  -->exec("C:\myscripts\examples-functions.sce" );
  ```

Once a function is defined, it can be used as if it was any other Scilab function.

```
-->exec("C:\myscripts\examples-functions.sce");
-->y = myfunction ( 3 )
 y  =
    6.
```

Notice that the previous function sets the value of the output argument `y`, with the statement `y=2*x`. This is mandatory. In order to see it, we define in the following script a function which sets the variable `z`, but not the output argument `y`.

```
function y = myfunction ( x )
  z = 2 * x
endfunction
```

In the following session, we try to use our function with the input argument `x=1`.

```
-->myfunction ( 1 )
 !--error 4
Undefined variable: y
at line       4 of function myfunction called by :
myfunction ( 1 )
```

Indeed, the interpreter tells us that the output variable `y` has not been defined.

When we make a computation, we often need more than one function in order to perform all the steps of the algorithm. For example, consider the situation where we need to optimize a system. In this case, we might use an algorithm provided by Scilab, say `optim` for example. First, we define the cost function which is to be optimized, according to the format expected by `optim`. Second, we define a driver, which calls the `optim` function with the required arguments. At least two functions are used in this simple scheme. In practice, a complete computation often requires a dozen of functions, or more. In this case, we may collect our functions in a library and this is the topic of the next section.

genlib	build library from functions in given directory
lib	library definition

Figure 25: Scilab commands to manage functions.

6.3 Function libraries

A function library is a collection of functions defined in the Scilab language and stored in a set of files.

When a set of functions is simple and does not contain any help or any source code in a compiled language like C/C++ or Fortran, a library is a very efficient way to proceed. Instead, when we design a Scilab component with unit tests, help pages and demonstration scripts, we develop a *module*. Developing a module is both easy and efficient, but requires a more advanced knowledge of Scilab. Moreover, modules are based on function libraries, so that understanding the former allows to master the latter. Modules will not be described in this document. Still, in many practical situations, function libraries allows to efficiently manage simple collections of functions and this is why we describe this system here.

In this section, we describe a very simple library and show how to load it automatically at Scilab startup.

Let us make a short outline of the process of creating and using a library. We assume that we are given a set of `.sci` files containing functions.

1. We create a binary version of the scripts containing the functions. The `genlib` function generates binary versions of the scripts, as well as additionnal indexing files.

2. We load the library into Scilab. The `lib` function allows to load a library stored in a particular directory.

Before analyzing an example, let us consider some general rules which must be following when we design a function library. These rules will then be reviewed in the next example.

The file names containing function definitions should end with the `.sci` extension. This is not mandatory, but helps in identifying the Scilab scripts on a hard drive.

Several functions may be stored in each `.sci` file, but only the first one will be available from outside the file. Indeed, the first function of the file is considered to be the only public function, while the other functions are (implicitely) private functions.

The name of the `.sci` must be the same as the name of the first function in the file. For example, if the function is to be named `myfun`, then the file containing this function must be `myfun.sci`. This is mandatory in order to make the `genlib` function work properly.

The functions which allow to manage libraries are presented in figure 25.

We shall now give a small example of a particular library and give some details about how to actually proceed.

Assume that we use a Windows system and that the `samplelib` directory contains two files:

- `C:/samplelib/function1.sci`:

```
function y = function1 ( x )
  y = 1 * function1_support ( x )
endfunction
function y = function1_support ( x )
```

```
  y = 3 * x
endfunction
```

- C:/samplelib/function2.sci:

```
function y = function2 ( x )
  y = 2 * x
endfunction
```

In the following session, we generate the binary files with the genlib function, which takes as its first argument a string associated with the library name, and takes as its second argument the name of the directory containing the files. Notice that only the functions function1 and function2 are publicly available: the function1_support function can be used inside the library, but cannot be used outside.

```
-->genlib("mylibrary","C:/samplelib")
-->mylibrary
 mylibrary  =
Functions files location : C:\samplelib\.
 function1           function2
```

The genlib function generates the following files in the directory "C:/samplelib":

- function1.bin: the binary version of the function1.sci script,
- function2.bin: the binary version of the function2.sci script,
- lib: a binary version of the library,
- names: a text file containing the list of functions in the library.

The binary files *.bin and the lib file are cross-platform in the sense that they work equally well under Windows, Linux or Mac.

Once the genlib function has been executed, the two functions are immediately available, as detailed in the following example.

```
-->function1(3)
 ans  =
    9.
-->function2(3)
 ans  =
    6.
```

In practical situations, though, we would not generate the library everytime it is needed. Once the library is ready, we would like to load the library directly. This is done with the lib function, which takes as its first argument the name of the directory containing the library and returns the library, as in the following session.

```
-->mylibrary = lib("C:\samplelib\")
 ans  =
Functions files location : C:\samplelib\.
 function1           function2
```

If there are many libraries, it might be unconvenient to load manually all libraries at startup. In practice, the `lib` statement can be written once for all, in Scilab startup file, so that the library is immediately available at startup. The startup directory associated with a particular Scilab installation is stored in the variable `SCIHOME`, as presented in the following session, for example on Windows.

```
-->SCIHOME
 SCIHOME  =
 C:\Users\username\AppData\Roaming\Scilab\scilab-5.2.0
```

In the directory associated with the `SCIHOME` variable, the startup file is `.scilab`. The startup file is automatically read by Scilab at startup. It must be a regular Scilab script (it can contain valid comments). To make our library available at startup, we simply write the following lines in our `.scilab` file.

```
// Load my favorite library.
mylibrary = lib("C:/samplelib/")
```

With this startup file, the functions defined in the library are available directly at Scilab startup.

6.4 Managing output arguments

In this section, we present the various ways to manage output arguments. A function may have one or several input and/or output arguments. In the most simple case, the number of input and output arguments is pre-defined and using such a function is easy. But, as we are going to see, even such a simple function can be called in various ways.

Assume that the function `simplef` is defined with 2 input arguments and 2 output arguments, as following.

```
function [y1 , y2] = simplef ( x1, x2 )
  y1 = 2 * x1
  y2 = 3 * x2
endfunction
```

In fact, the number of output arguments of such a function can be 0, 1 or 2. When there is no output argument, the value of the first output argument in stored in the `ans` variable. We may also set the variable `y1` only. Finally, we may use all the output arguments, as expected. The following session presents all these calling sequences.

```
-->simplef ( 1 , 2 )
 ans  =
    2.
-->y1 = simplef ( 1 , 2 )
 y1  =
    2.
-->[y1,y2] = simplef ( 1 , 2 )
 y2  =
    6.
 y1  =
    2.
```

We have seen that the most basic way of defining functions already allows to manage a variable number of output arguments. There is an even more flexible way of managing a variable number of input and output arguments, based on the `argn`, `varargin` and `varargout` variables. This

`plot`	2D plot
`surf`	3D plot
`contour`	contour plot
`pie`	pie chart
`histplot`	histogram
`bar`	bar chart
`barh`	horizontal bar chart
`hist3d`	3D histogram
`polarplot`	plot polar coordinates
`Matplot`	2D plot of a matrix using colors
`Sgrayplot`	smooth 2D plot of a surface using colors
`grayplot`	2D plot of a surface using colors

Figure 26: Scilab plot functions

more advanced topic will not be detailed in this document, but it may be worth to use these tools when we want to design an even more flexible function.

7 Plotting

Producing plots and graphics is a very common task for analysing data and creating reports. Scilab offers many ways to create and customize various types of plots and charts. In this section, we present how to create 2D plots and contour plots. Then we customize the title and the legend of our graphics. We finally export the plots so that we can use it in a report.

7.1 Overview

Scilab can produce many types of 2D and 3D plots. The following is a short list of several common charts that Scilab can create:

- x-y plots: `plot`,
- contour plots: `contour`,
- 3D plots: `surf`,
- histograms: `histplot`,
- bar charts: `bar`,
- etc...

The most commonly used plot functions are presented in figure 26.

In order to get an example of a 3D plot, we can simply type the statement `surf()` in the Scilab console.

```
-->surf()
```

`linspace`	linearly spaced vector
`feval`	evaluates a function on a grid
`legend`	configure the legend of the current plot
`title`	configure the title of the current plot
`xtitle`	configure the title and the legends of the current plot

Figure 27: Scilab functions used when creating a plot.

During the creation of a plot, we use several functions in order to create the data or to configure the plot. The functions which are presented in figure 27 will be used in the examples of this section.

7.2 2D plot

In this section, we present how to produce a simple x-y plot. We emphasize the use of vectorized functions, which allow to produce matrices of data in one function call.

We begin by defining the function which is to be plotted. The function `myquadratic` squares the input argument `x` with the ˆ operator.

```
function f = myquadratic ( x )
  f = x^2
endfunction
```

We can use the `linspace` function in order to produce 50 values in the interval $[1, 10]$.

```
xdata = linspace ( 1 , 10 , 50 );
```

The `xdata` variable now contains a row vector with 50 entries, where the first value is equal to 1 and the last value is equal to 10. We can pass it to the `myquadratic` function and get the function value at the given points.

```
ydata = myquadratic ( xdata );
```

This produces the row vector `ydata`, which contains 50 entries. We finally use the `plot` function so that the data is displayed as a x-y plot.

```
plot ( xdata , ydata )
```

The figure 28 presents the associated x-y plot.

Notice that we could have produced the same plot without generating the intermediate array `ydata`. Indeed, the second input argument of the `plot` function can be a function, as in the following session.

```
plot ( xdata , myquadratic )
```

When the number of points to manage is large, the previous syntax may allow to save significant amount of time and memory space.

7.3 Titles, axis and legends

In this section, we present the Scilab graphics features which allow to configure the title, axis and legends of an x-y plot.

In the following example, we define a quadratic function and plot it with the `plot` function.

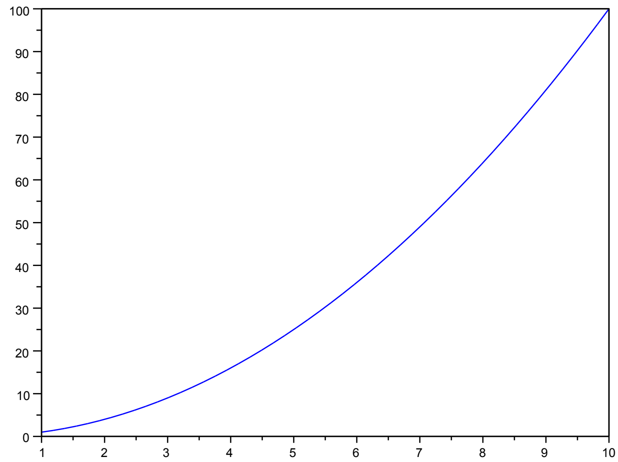

Figure 28: A simple x-y plot.

```
function f = myquadratic ( x )
  f = x.^2
endfunction
xdata = linspace ( 1 , 10 , 50 );
ydata = myquadratic ( xdata );
plot ( xdata , ydata )
```

We have now the plot which is presented in figure 28.

Scilab graphics system is based on *graphics handles*. The graphics handles provide an object-oriented access to the fields of a graphics entity. The graphics layout is decomposed into sub-objects such as the line associated with the curve, the x and y axis, the title, the legends, and so forth. Each object can be in turn decomposed into other objects if required. Each graphics object is associated with a collection of properties, such as the width or color of the line of the curve, for example. These properties can be queried and configured simply by setting or getting its value, as any other Scilab variable. Managing handles is easy and very efficient.

But most basic plot configurations can be done by simple function calls and, in this section, we will focus in these basic features.

In the following script, we use the `title` function in order to configure the title of our plot.

```
title ( "My title" );
```

We may want to configure the axis of our plot as well. For this purpose, we use the `xtitle` function in the following script.

```
xtitle ( "My title" , "X axis" , "Y axis" );
```

The figure 29 presents the produced plot.

It may happen that we want to compare two sets of datas in the same 2D plot, that is, one set of x data and two sets of y datas. In the following script, we define the two functions $f(x) = x^2$ and $f(x) = 2x^2$ and plot the data on the same x-y plot. We additionnaly add the "+-" and "o-" options to the `plot` function, so that we can distinguish the two curves $f(x) = x^2$ and $f(x) = 2x^2$.

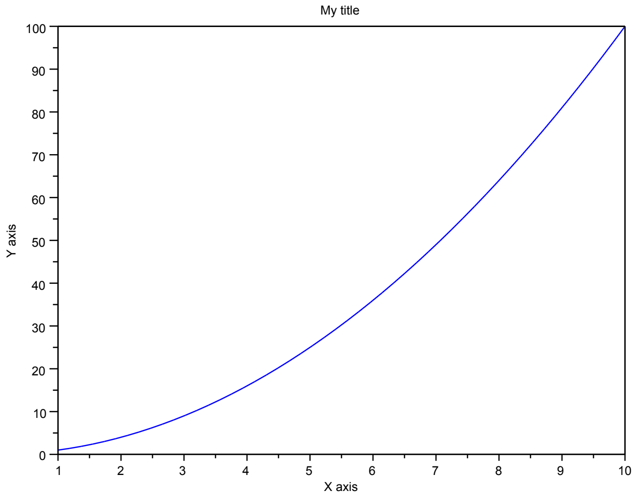

Figure 29: The x-y plot of a quadratic function – This is the same plot as in figure 28, with title and x-y axis configured.

```
function f = myquadratic ( x )
  f = x^2
endfunction
function f = myquadratic2 ( x )
  f = 2 * x^2
endfunction
xdata = linspace ( 1 , 10 , 50 );
ydata = myquadratic ( xdata );
plot ( xdata , ydata , "+-" )
ydata2 = myquadratic2 ( xdata );
plot ( xdata , ydata2 , "o-" )
xtitle ( "My title" , "X axis" , "Y axis" );
```

Moreover, we must configure a legend so that we can know what curve is associated with $f(x) = x^2$ and what curve is associated with $f(x) = 2x^2$. For this purpose, we use the `legend` function in order to print the legend associated with each curve.

```
legend ( "x^2" , "2x^2" );
```

The figure 30 presents the produced x-y plot.

We now know how to create a graphics plot and how to configure it. If the plot is sufficiently interesting, it may be worth to put it into a report. To do so, we can export the plot into a file, which is the subject of the next section.

7.4 Export

In this section, we present ways of exporting plots into files, either interactively or automatically with Scilab functions.

Scilab can export any graphics into the vectorial and bitmap formats presented in figure 31. Once a plot is produced, we can export its content into a file, by using interactively the

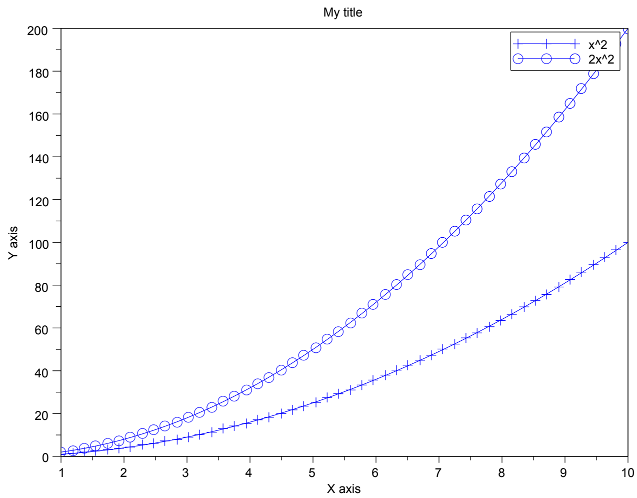

Figure 30: The x-y plot of two quadratic functions – We have configured the legend so that we can distinguish the two functions $f(x) = x^2$ and $f(x) = 2x^2$.

File>Export to... menu of the graphics window. We can then set the name of the file and its type.

We can alternatively use the `xs2*` functions, presented in figure 31. All these functions are based on the same following calling sequence:

```
xs2png ( window_number , filename )
```

where `window_number` is the number of the graphics window and `filename` is the name of the file to export. For example, the following session exports the plot which is in the graphics window number 0, which is the default graphics window, into the file `foo.png`.

```
xs2png ( 0 , "foo.png" )
```

If we want to produce higher quality documents, the vectorial formats are to be prefered. For example, LaTeX documents may use Scilab plots exported into PDF files to improve their readability, whatever the size of the document.

8 Notes and references

There are a number of topics which have not been presented in this document. We hope that the current document is a good starting point for using Scilab so that learning about these specific topics should not be a problem. We have already mentioned a number of other sources of documentation for this purpose at the begining of this document.

Further reading may be obtained from the Scilab web cite [3], in the documentation section.

Vectorial	
xs2png	export into PNG
xs2pdf	export into PDF
xs2svg	export into SVG
xs2eps	export into Encapsulated Postscript
xs2ps	export into Postscript
Bitmap	
xs2fig	export into FIG
xs2gif	export into GIF
xs2jpg	export into JPG
xs2bmp	export into BMP
xs2ppm	export into PPM
xs2emf	export into EMF (only for Windows)

Figure 31: Export functions.

9 Acknowledgments

I would like to thank Claude Gomez, Vincent Couvert, Allan Cornet and Serge Steer who let me share their comments about this document. I am also grateful to Julie Paul and Sylvestre Ledru who helped me during the writing of this document.

References

[1] Atlas - automatically tuned linear algebra software. http://math-atlas.sourceforge.net.

[2] Cecill and free software. http://www.cecill.info.

[3] The Scilab Consortium. Scilab. http://www.scilab.org.

[4] Intel. Intel math kernel library. http://software.intel.com/en-us/intel-mkl/.

[5] Sylvestre Ledru. Different execution modes of scilab. http://wiki.scilab.org/Different_execution_modes_of_Scilab.

[6] Flexdock project. Flexdock project home. https://flexdock.dev.java.net/.

Index