Introduction to Scilab

Michaël Baudin

September 2011

Abstract

In this document, we make an overview of Scilab features so that we can get familiar with this environment. 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. 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

In this section, we present an overview of Scilab. The first subsection introduces the open source project associated with the creation of this document. Then we present the software, licence and scientific aspects of Scilab. In the third subsection, we describe the methods to download and install Scilab on Windows, GNU/Linux and Mac operating systems. In the remaining subsections, we describe various sources of information needed when we have to get some help from Scilab or from other users. We describe the built-in help pages and analyse the mailing lists and wiki which are available online. Finally, we take a moment to look at the demonstrations which are provided with Scilab.

1.1 Introduction

This document is an open-source project. The LaTeX sources are available on the Scilab Forge:

http://forge.scilab.org/index.php/p/docintrotoscilab/

The LaTeX sources are provided under the terms of the Creative Commons Attribution-ShareAlike 3.0 Unported License:

http://creativecommons.org/licenses/by-sa/3.0

The Scilab scripts are provided on the Forge, inside the project, under the `scripts` sub-directory. The scripts are available under the CeCiLL licence:

http://www.cecill.info/licences/Licence_CeCILL_V2-en.txt

1.2 Overview of Scilab

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 speeds up the development process, because the user directly accesses 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 modules so that they can solve their particular problems. The Scilab language can 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's 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. Online help is provided in many local languages.

From the 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 create 2D and 3D plots as well as user interfaces. The Xcos environment provides a hybrid dynamic systems modeler and simulator.

1.3 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

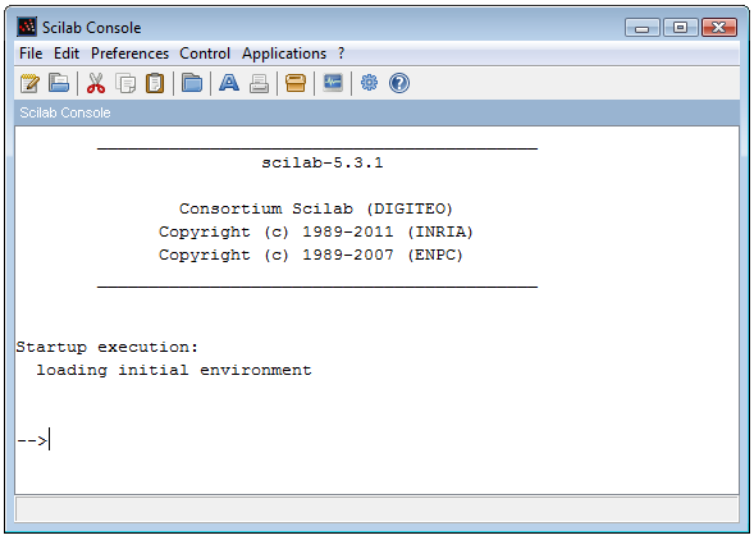
`http://www.scilab.org`

or from the Download area

`http://www.scilab.org/download`

Scilab binaries are provided for both 32 and 64-bit platforms so that they match 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.



```
                 scilab-5.3.1

            Consortium Scilab (DIGITEO)
          Copyright (c) 1989-2011 (INRIA)
          Copyright (c) 1989-2007 (ENPC)


Startup execution:
  loading initial environment

-->
```

Figure 1: Scilab console on Windows.

1.3.1 Installing Scilab on 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.

On Windows, if your machine is based on an Intel processor, the Intel Math Kernel Library (MKL) [6] enables Scilab to perform faster numerical computations.

1.3.2 Installing Scilab on GNU/Linux

On GNU/Linux, the binary versions are available from Scilab website as `.tar.gz` files. There is no need for an installation program with Scilab on GNU/Linux: simply unzip the file in one target directory. Once done, the binary file is located in *<path>/scilab-5.x.x/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). This 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 on GNU/Linux and its release in Linux distributions.

For now, Scilab comes on Linux with a binary linear algebra library which guarantees portability. On GNU/Linux, Scilab does not come with a binary version of ATLAS [1], so that linear algebra is a little slower for that platform, compared to Windows.

1.3.3 Installing Scilab on Mac OS

On 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.

For technical reasons, Scilab version 5.3 for Mac OS X comes with a disabled Tcl/Tk interface. As a consequence, there are some small limitations on the use of Scilab on this platform. For example, the Scilab / Tcl interface (TclSci) and the graphic 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 a Mac OS system is easy, and uses the shortcuts which are familiar to the 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. On Mac OS, Scilab does not come with a binary version of ATLAS [1], so that linear algebra is a little slower for that platform.

1.4 How to get help

The most simple way to get the online help integrated to Scilab is to use the function `help`. 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, for which help is required, as in the following session.

```
help optim
```

Scilab automatically opens the associated entry in the help.

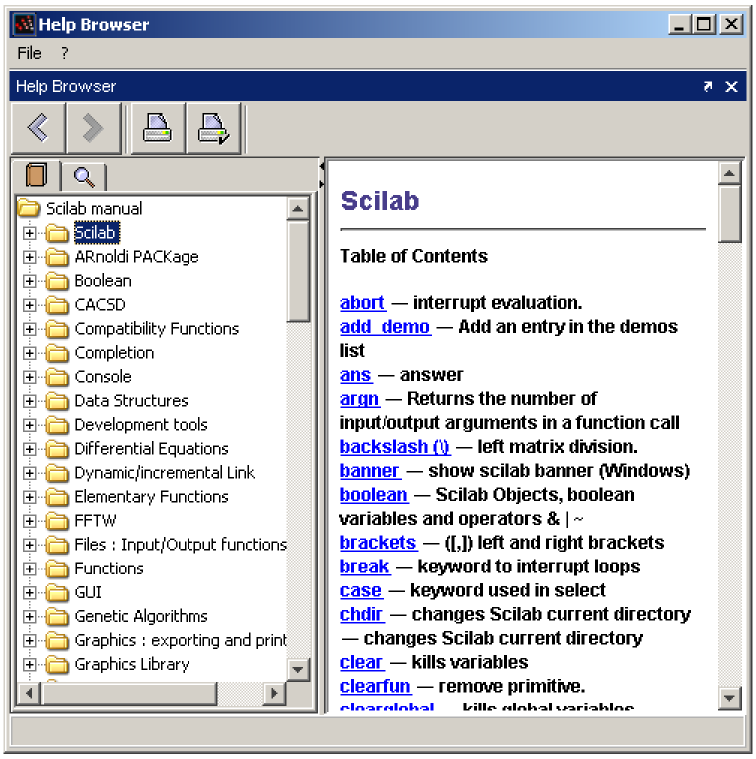
We can also use the help provided on the Scilab web site

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

This 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 page I need. With that method, I can see the help pages for 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



Scilab

Table of Contents

abort — interrupt evaluation.
add_demo — Add an entry in the demos list
ans — answer
argn — Returns the number of input/output arguments in a function call
backslash (\) — left matrix division.
banner — show scilab banner (Windows)
boolean — Scilab Objects, boolean variables and operators & | ~
brackets — ([,]) left and right brackets
break — keyword to interrupt loops
case — keyword used in select
chdir — changes Scilab current directory
— changes Scilab current directory
clear — kills variables
clearfun — remove primitive.
clearglobal — kills global variables

Figure 2: Scilab help window.

1.5 Mailing lists, the Wiki, the Forge and the 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

The wiki is used both by Scilab users and developers to publish information about Scilab. From a developer's 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 is a tool to submit a report each time we find a bug:

`http://bugzilla.scilab.org`

It may happen that this bug has already been discovered by someone else. This is why it is advised to search the bug database for existing related problems before reporting a new bug. If the bug is not reported yet, it is a very good thing to report it, along with a test script. This test script should remain as simple as possible 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 regularly delivers press releases and general announcements.

The Scilab forge is a tool for external modules developers to help them in their development work:

`http://forge.scilab.org`

The Scilab Forge is the place where we can share the source code of Scilab projects. It provides several components to facilitate the development. The forge provides:

- a source code revision system (SVN or GIT),
- a Download system (to release source archives, documents in pdf form, binaries),
- documentation pages (as a wiki),
- a bug tracker.

This tool lets developers work on a common source code as a team, even if the developers are not in the same physical location. To use the forge as a user, we simple download the releases or browse the source code with a web browser. To use the Scilab forge as a developper, it is mandatory to be familiar with version management systems such as SVN or GIT. More details on Scilab's forge can be found in [11].

1.6 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*. 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

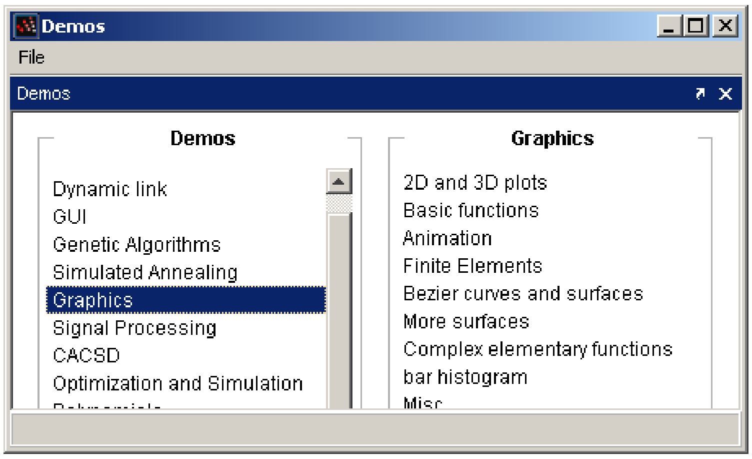


Figure 3: Scilab demos window.

```
<path>\scilab-5.3.1\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-5.3.1\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.

1.7 Exercises

Exercise 1.1 (***Installing Scilab***) Install the current version of Scilab on your system: at the time where this document is written, this is Scilab v5.3.3. It is instructive to install an older version of Scilab, in order to compare current behavior against the older one. Install Scilab 4.1.2 and see the differences.

Exercise 1.2 (***Inline help:*** `derivative`) The `derivative` function computes the numerical derivative of a function. The purpose of this exercise is to find the corresponding help page, by various means. In the inline help, find the entry corresponding to the `derivative` function. Find the corresponding entry in the online help. Use the console to find the help.

Exercise 1.3 (***Asking a question on the forum***) You probably already have one or more questions. Post your question on the users' mailing list *users@lists.scilab.org*.

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

We also present the management of the graphical windows with the docking system. Finally, we present two major features of Scilab: the localization of Scilab, which provides messages and help pages in the language of the user, and the ATOMS system, a packaging system for external modules.

2.1 The console

The first way is to use Scilab interactively, by typing commands in the console, analyzing the results and 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 with Scilab behavior by himself. This is indeed a good way of understanding the behavior of the program and, most of the time, it is a quick and smooth way of performing the desired computation.

In the following example, the function `disp` is used in the 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!`.

When we edit a 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.

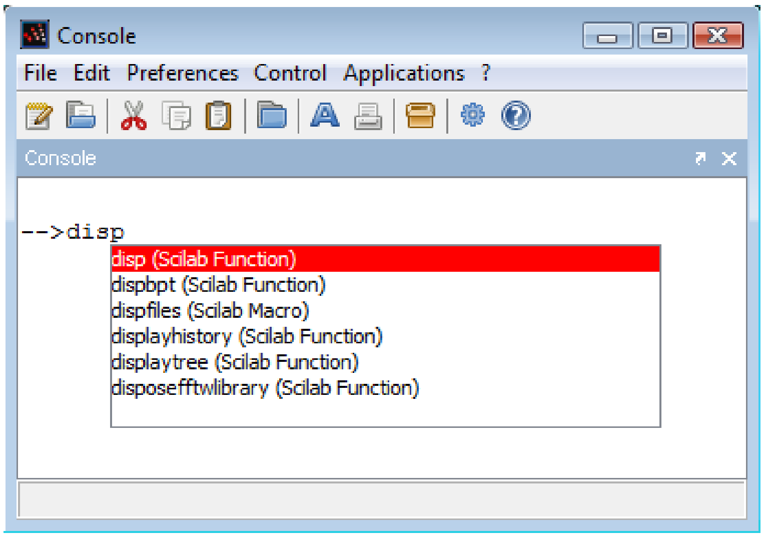
In order to get access to previously executed commands, we use the up arrow ↑ key. This lets us browse the previous commands by using the up ↑ and down ↓ arrow keys.

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 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.



```
-->disp
        disp (Scilab Function)
        dispbpt (Scilab Function)
        dispfiles (Scilab Macro)
        displayhistory (Scilab Function)
        displaytree (Scilab Function)
        disposefftwlibrary (Scilab Function)
```

Figure 4: The completion in the console.

2.2 The editor

Scilab version 5.3 provides an editor for editing scripts easily. Figure 5 presents the editor during the editing of 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()
```

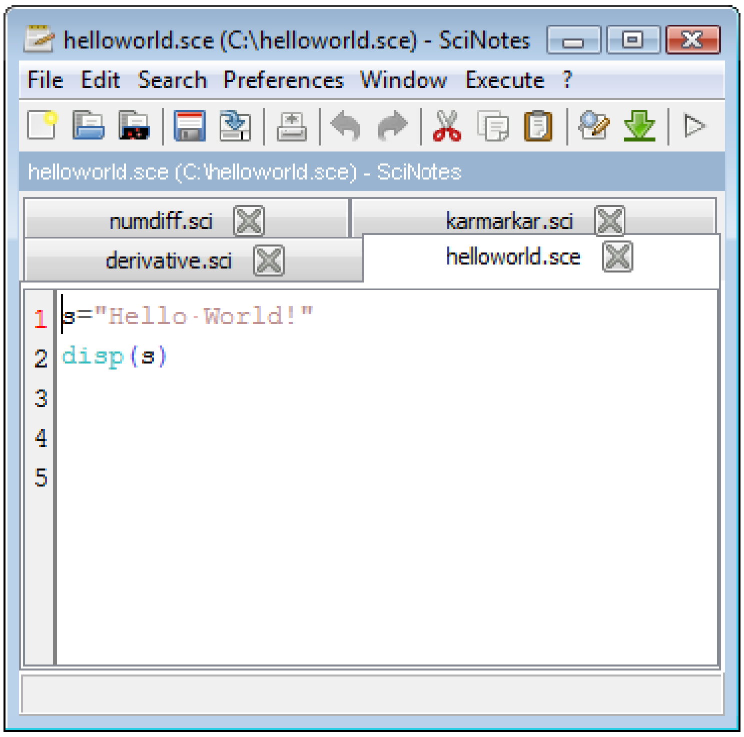
This editor manages several files at the same time, as presented in figure 5, where we edit five files at the same time.

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

- *Load into Scilab* executes the statements in the current file, as if we did a copy and paste. This implies that the statements which do not end with the semicolon ";" character will produce an output in the console.
- *Evaluate Selection* executes the statements which are currently selected.
- *Execute File Into Scilab* executes 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 extremely convenient, because it formats algorithms, so that the `if`, `for` and other structured blocks are easy to analyze.


helloworld.sce (C:\helloworld.sce) - SciNotes
File Edit Search Preferences Window Execute ?
helloworld.sce (C:\helloworld.sce) - SciNotes
numdiff.sci
karmarkar.sci
derivative.sci
helloworld.sce

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

Figure 5: The editor.

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

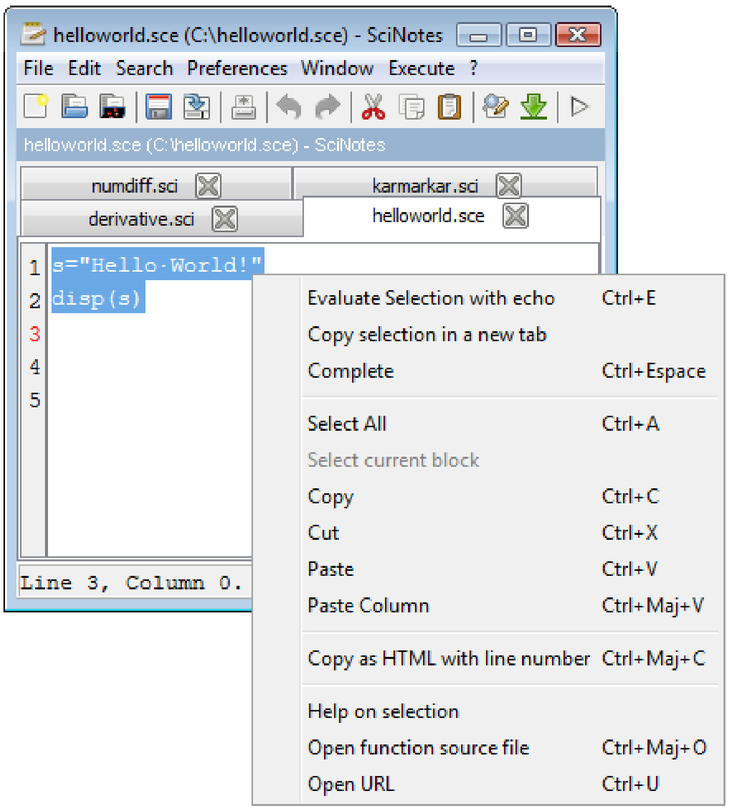
2.3 Docking

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.

The docking system uses Flexdock [13], 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.

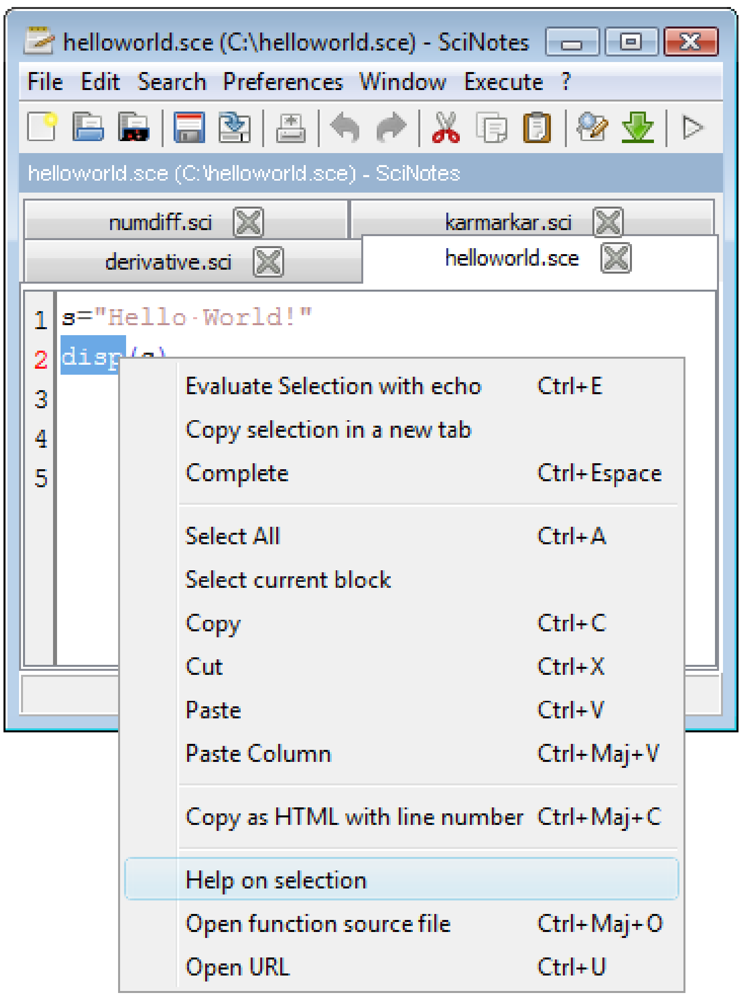
The Flexdock system allows us to drag and drop the editor into the console, so that we finally have only one window, with several sub-windows. All Scilab windows are dockable, including the console, the editor, the variable browser, the command history, the help and the plotting windows. In figure 9, 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


helloworld.sce (C:\helloworld.sce) - SciNotes
File Edit Search Preferences Window Execute ?
helloworld.sce (C:\helloworld.sce) - SciNotes
numdiff.sci
karmarkar.sci
derivative.sci
helloworld.sce
Line 3, Column 0.
Evaluate Selection with echo Ctrl+E
Copy selection in a new tab
Complete Ctrl+Espace
Select All Ctrl+A
Select current block
Copy Ctrl+C
Cut Ctrl+X
Paste Ctrl+V
Paste Column Ctrl+Maj+V
Copy as HTML with line number Ctrl+Maj+C
Help on selection
Open function source file Ctrl+Maj+O
Open URL Ctrl+U

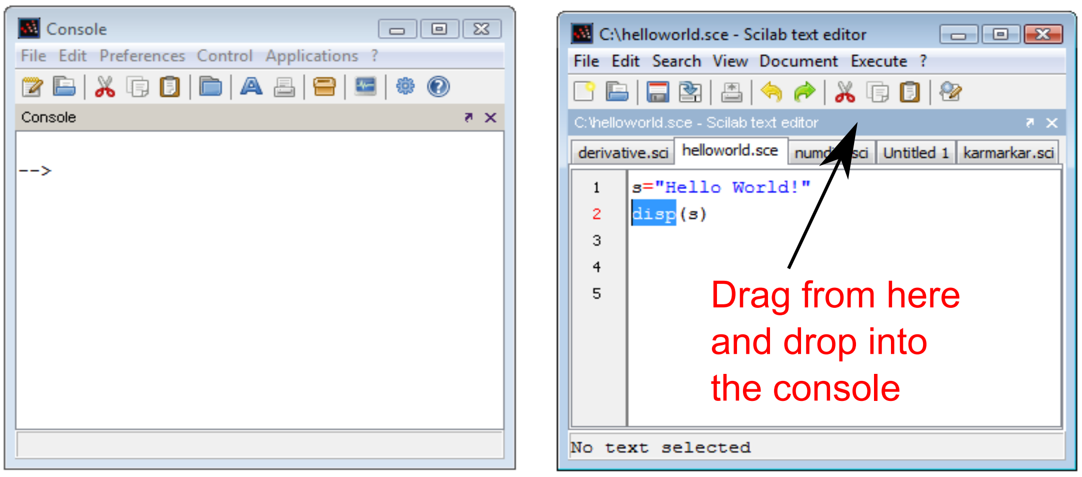
```
s="Hello World!"
disp(s)
```

Figure 6: Context menu in the editor.


helloworld.sce (C:\helloworld.sce) - SciNotes
File Edit Search Preferences Window Execute ?
helloworld.sce (C:\helloworld.sce) - SciNotes
numdiff.sci
karmarkar.sci
derivative.sci
helloworld.sce
Evaluate Selection with echo Ctrl+E
Copy selection in a new tab
Complete Ctrl+Espace
Select All Ctrl+A
Select current block
Copy Ctrl+C
Cut Ctrl+X
Paste Ctrl+V
Paste Column Ctrl+Maj+V
Copy as HTML with line number Ctrl+Maj+C
Help on selection
Open function source file Ctrl+Maj+O
Open URL Ctrl+U

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

Figure 7: Context help in the editor.

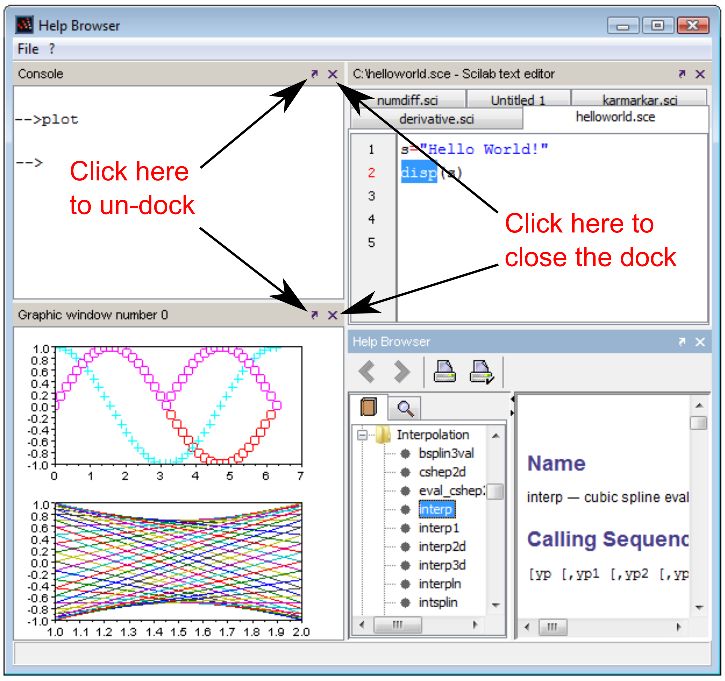

Console
File Edit Preferences Control Applications ?
Console
C:\helloworld.sce - Scilab text editor
File Edit Search View Document Execute ?
C:\helloworld.sce - Scilab text editor
derivative.sci
helloworld.sce
numdiff.sci
Untitled 1
karmarkar.sci
Drag from here
and drop into
the console

```
-->
```

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

```
No text selected
```

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.



```
-->plot

-->
```

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

Figure 9: Actions in the title bar of the docking window. The round arrow in the title bar of the window undocks the window. The cross closes the window.

mouse over the target window and notice that a window, surrounded by dotted lines is displayed. This "phantom" 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 figure 9.

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

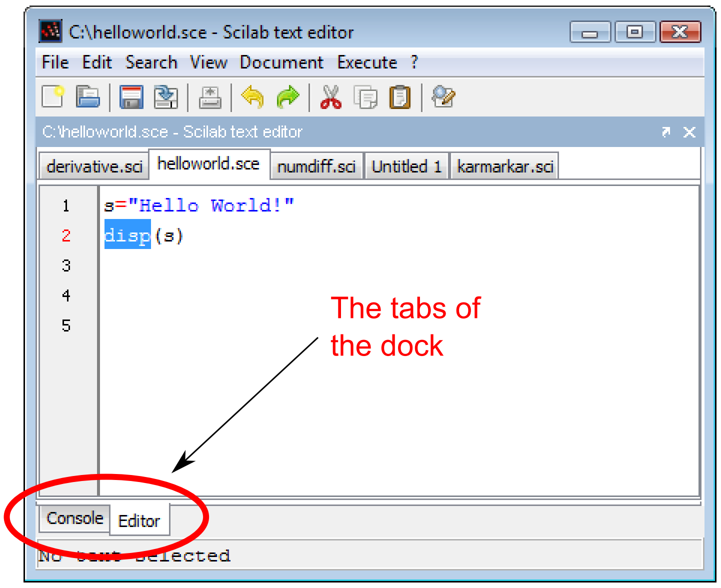
2.4 The variable browser and the command history

Scilab provides a variable browser, which displays the list of variables currently used in the environment. Figure 11 presents the state of this browser during a session.

We can access this browser through the menu Applications > Variable Browser, but the function `browsevar()` has the same effect.

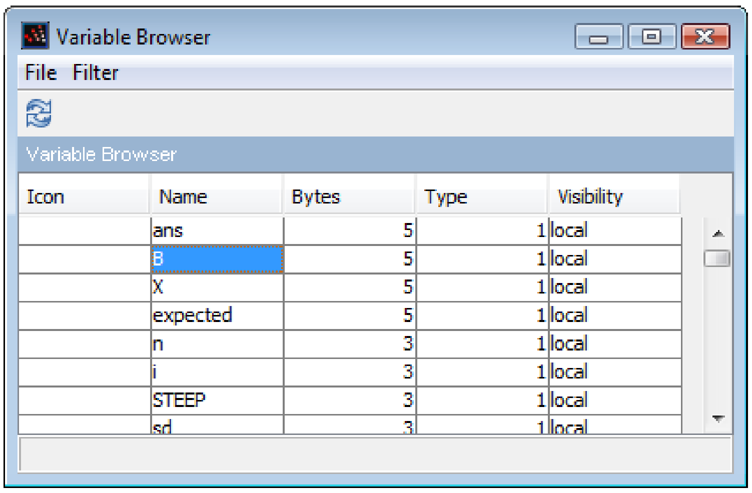
We can double click on a variable, which opens the variable editor, as presented in the figure 12. We can then interactively change the value of a variable by changing its content in a cell. On the other hand, if we change the variable content within the console, we must refresh the content of the dialog box by pushing the refresh button in the toolbar of the Variable Editor.

The Command History dialog allows browsing through the commands that we have previously executed. This dialog is available in the menu Applications > Command History and is presented in the figure 13.


C:\helloworld.sce - Scilab text editor
File Edit Search View Document Execute ?
C:\helloworld.sce - Scilab text editor
derivative.sci helloworld.sce numdiff.sci Untitled 1 karmarkar.sci
The tabs of the dock
Console Editor
No text selected

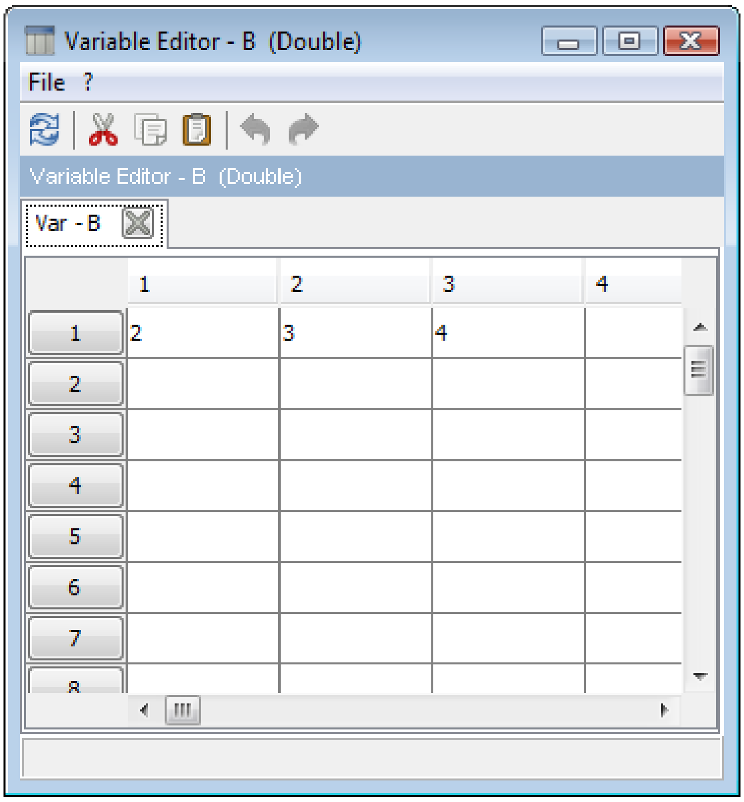
```
s="Hello World!"
disp(s)
```

Figure 10: Docking tabs.



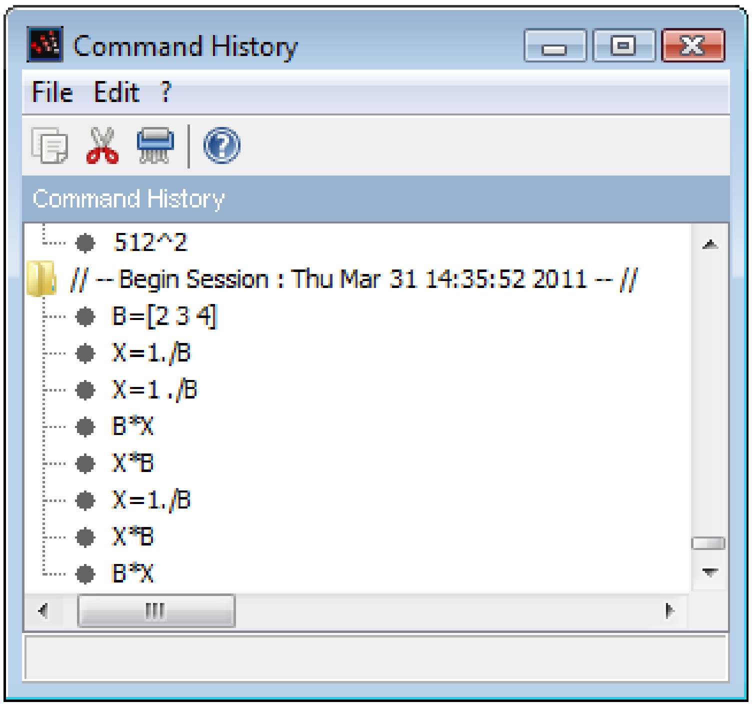
Icon	Name	Bytes	Type	Visibility
	ans	5	1	local
	B	5	1	local
	X	5	1	local
	expected	5	1	local
	n	3	1	local
	i	3	1	local
	STEEP	3	1	local
	sd	3	1	local

Figure 11: The variable browser.



	1	2	3	4
1	2	3	4	
2				
3				
4				
5				
6				
7				
8				

Figure 12: The variable editor.



```
512^2
// -- Begin Session : Thu Mar 31 14:35:52 2011 -- //
B=[2 3 4]
X=1./B
X=1 ./B
B*X
X*B
X=1./B
X*B
B*X
```

Figure 13: The command history.

We can select any command in the list and double-click on it to execute it in the console. The right-click opens a context menu which lets us evaluate the command or edit it in the editor.

2.5 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 contain Scilab functions and executing them loads the functions into Scilab environment (but does not execute them),
- files having the `.sce` extension contain 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...

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 used most of the time with a sequence of calls to the `exec` and `disp` functions.

2.6 Batch processing

Another way of using Scilab is from the command line. Several command line options are available and are presented in figure 14. Whatever the operating system is, binaries are located in the directory *scilab-5.3.1/bin*. Command line options must be appended to the binary for the specific platform, as described below. The *-nw* option disables the display of the console. The *-nwni* option launches 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.

-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 for example, ”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 14: Scilab command line options.

- Under Linux, the `scilab` script provides options which 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[7].
- 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-5.3.1\bin>Scilex.exe -nwni
        ___________________________________________
                       scilab-5.3.1
                 Consortium Scilab (DIGITEO)
               Copyright (c) 1989-2011 (INRIA)
               Copyright (c) 1989-2007 (ENPC)
        ___________________________________________
Startup execution:
  loading initial environment
-->plot()
       !--error 4
Undefined variable: plot
```

The most useful command line option is the *-f* option, which executes 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-5.3.1\bin\Scilex.exe -f myscript2.sce
        ___________________________________________
                       scilab-5.3.1
                 Consortium Scilab (DIGITEO)
               Copyright (c) 1989-2011 (INRIA)
               Copyright (c) 1989-2007 (ENPC)
        ___________________________________________
Startup execution:
  loading initial environment
 Hello World !
C:\scripts>
```

Any line which begins with the two slash 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 will now wait for user input, as expected. To exit, we interactively type the `quit()` statement in the terminal.

2.7 Localization

By default, Scilab provides its messages and its help pages in the English language. But it can also provide them in French, in Chinese, in Portuguese and in several other languages. In this section, we review these features and see their effect in Scilab.

The localization features of Scilab change two different set of features in Scilab:

- the messages of the Scilab application (menus, error messages, etc...),
- help pages.

The table 15 presents the list of languages supported by Scilab 5.3.2 for the application itself. For some of these languages, the help pages of Scilab are (partially) translated, as indicated in the table.

Scilab provides several functions which manages the localization. These functions are presented in the table 16.

On Windows, we can use the `setdefaultlanguage` function, which takes a string representing the required language as input argument. Then we restart Scilab so that the menus of the console are translated. In the following example, we use the `setdefaultlanguage` function in order to configure the language in Portuguese.

```
setdefaultlanguage("pt_BR")
```

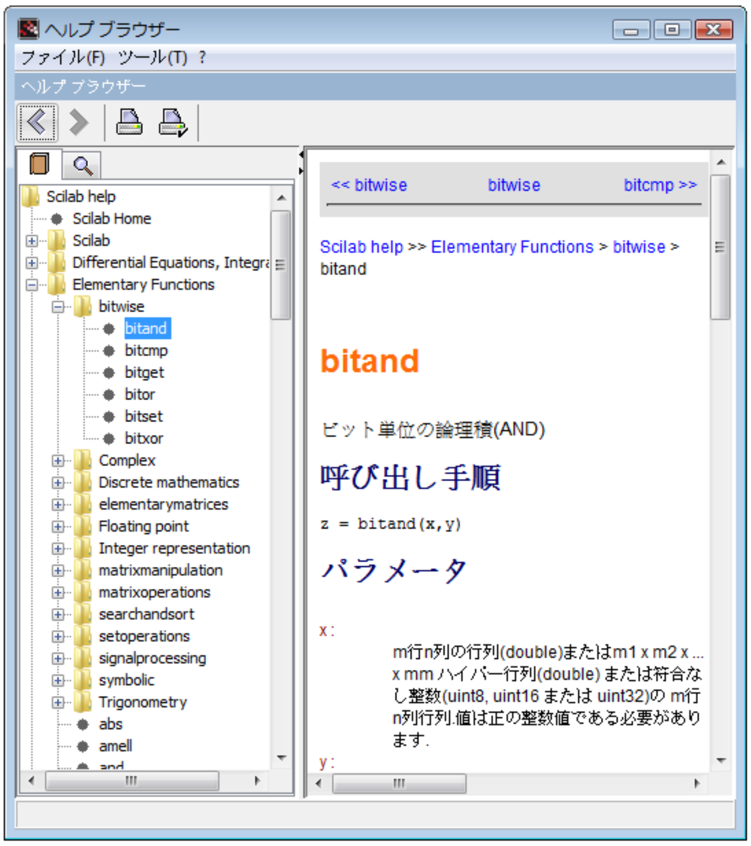
When we restart Scilab, the error messages are provided in Portuguese:

Messages	
ca_ES	Catalan - Spain
de_DE	German - Germany
en_US	English - United States
es_ES	Spanish - Castilian - Spain
fr_FR	French - France (with help pages)
it_IT	Italian - Italy
ja_JP	Japanese - Japan (with help pages)
pl_PL	Polish - Poland
pt_BR	Portuguese - Brazil (with help pages)
ru_RU	Russian - Russian Federation
uk_UA	Ukrainian - Ukraine
zh_CN	Simplified Chinese
zh_TW	Chinese Traditional

Figure 15: Languages supported by Scilab.

`getdefaultlanguage`	Returns the default language used by Scilab.
`getlanguage`	Returns current language used by Scilab.
`setdefaultlanguage`	Sets and saves the internal LANGUAGE value.
`setlanguage`	Sets the internal LANGUAGE value.
`dgettext`	Get text translated into the current locale and a specific domain.
`gettext`	Get text translated into the current locale and domain.
`_`	Get text translated into the current locale and domain.

Figure 16: Functions related to localization.



Scilab help >> Elementary Functions > bitwise > bitand

bitand

ビット単位の論理積(AND)

呼び出し手順

```
z = bitand(x,y)
```

パラメータ

x :

m行n列の行列(double)またはm1 x m2 x ... x mm ハイパー行列(double) または符合なし整数(uint8, uint16 または uint32)の m行n列行列.値は正の整数値である必要があります.

y :

Figure 17: The help page of `bitand` in Japanese.

```
-->1+"foo"
     !--error 144
OperaÃğÃčo indefinida para os dados operandos.
Verifique ou defina a funÃğÃčo %s_a_c para overloading.
```

The figure 17 presents the help page of the `bitand` function in Japanese.

Under GNU/Linux, Scilab uses the language of the operating system, so that most users should get Scilab in their own language without configuring Scilab. For example, in Ubuntu, installing and configuring languages can be done in the *System > Administration > Language Support* menu.

Under GNU/Linux or Mac OS X, an other way to start Scilab in an other language is to set the *LANG* environment variable. For example, the following command in the Linux terminal runs Scilab in Japanese:

```
# Starts Scilab in Japanese
LANG=ja_JP scilab
```

Still, there might be differences between the language provided by GNU/Linux, and the language used by Scilab. These differences may come from a wrong definition of the language, where Scilab cannot find the language which corresponds to the expected one. When we run Scilab from a Linux terminal, the following message may appear:

```
$ Warning: Localization issue.
```

```
Does not support the locale '' (null) C.
(process:1516): Gtk-WARNING **:
Locale not supported by C library.
        Using the fallback 'C' locale.
```

One common reason for this error is the various ways to define a language. Notice, for example, that there is a difference between the "fr" language (French) and the "fr_FR" language (French in France). In this case, we must configure the language to "fr_FR", which is the language detected by Scilab. Another reason, especially on the Debian GNU/Linux distribution, is that the locale might not have been compiled. In this case, we can use the "*dpkg-reconfigure locales*" command in the Linux terminal.

More information on Scilab's localization is provided at [8].

2.8 ATOMS, the packaging system of Scilab

In this section, we present ATOMS, which is a set of tools designed to install pre-built toolboxes.

Scilab is designed to be extended by users, who can create new functions and use them as if they were distributed with Scilab. These extensions are called "toolboxes" or "external modules". The creation of a new module, with its associated help pages and unit tests, is relatively simple and this is a part of the success of Scilab.

However, most modules cannot be used directly in source form: the module has to be compiled so that the binary files can be loaded into Scilab. This compilation step is not straightforward and may be even impossible for users who want to use a module based on C or Fortran source code and who do not have a compiler. This is one of the issues that ATOMS has solved: modules are provided in binary form, allowing the user to install a module without any compilation phase and without Scilab compabilities issues.

An extra feature for the user is that most modules are available on all platforms: the developper benefits from the compilation farm of the Scilab Consortium, and the user benefits from a module which is, most of the time, guaranteed to be cross-platform.

ATOMS is the packaging system of Scilab external modules. With this tool, pre-built (i.e. pre-compiled) Scilab modules, can be downloaded, installed and loaded. The dependencies are managed, so that if a module A depends on a module B, the installation of the module A automatically installs the module B. This is similar to the packaging system available in most GNU/Linux/BSD distributions. ATOMS modules are available on all operating systems on which Scilab is available, that is, on Microsoft Windows, GNU/Linux and Mac OS X. For example, when a ATOMS module is installed on Scilab running on a MS Windows operating system, the pre-built module corresponding to the MS Windows version of the module is automatically installed. The web portal for ATOMS is:

http://atoms.scilab.org

This portal presents the complete list of ATOMS modules and let the developpers of the modules upload their new modules. The figure 18 presents the 10 most

Image Processing Design Toolbox (`"IPD"`)	*Functions for object detection*
Scilab Image and Video Processing toolbox (`"SIVP"`)	*Image and video processing*
Plotting library (`"plotlib"`)	*"Matlab-like" Plotting library for Scilab*
Scilab 2 C (`"scilab2c"`)	*Translate Scilab code into C code*
Apifun (`"apifun"`)	*Check input arguments in macros*
Module Lycée (`"module_lycee"`)	*Scilab pour les lycées*
Guimaker (`"guimaker"`)	*Create GUIs with a minimum of programming*
Make Matrix (`"makematrix"`)	*A collection of test matrices*
GUI Builder (`"guibuilder"`)	*A Graphic User Interface Builder*
Scilab_XLL (`"Scilab_XLL"`)	*Scilab Xll add in for Excel*

Figure 18: The 10 most downloaded ATOMS modules.

downloaded ATOMS modules. This is just an arbitrary set of modules: more than 100 modules are currently available on ATOMS.

There are two ways to install an ATOMS module. The first way is to use the `atomsGui` function, which opens a Graphical User Interface (GUI) which lets the user browse through all the available ATOMS modules. This tool is also available from the menu "Applications > Module manager - ATOMS" of the Scilab console. Within the GUI, we can read the description of the module and simply click on the "Install" button. The figure 19 presents the ATOMS GUI.

The second way is to use the `atomsInstall` function, which takes the name of a module as input argument. For example, to install the `makematrix` module, the following statement should be executed:

```
atomsInstall("makematrix")
```

Then Scilab should be restarted and the `makematrix` module (and, if any, its dependencies) are automatically loaded.

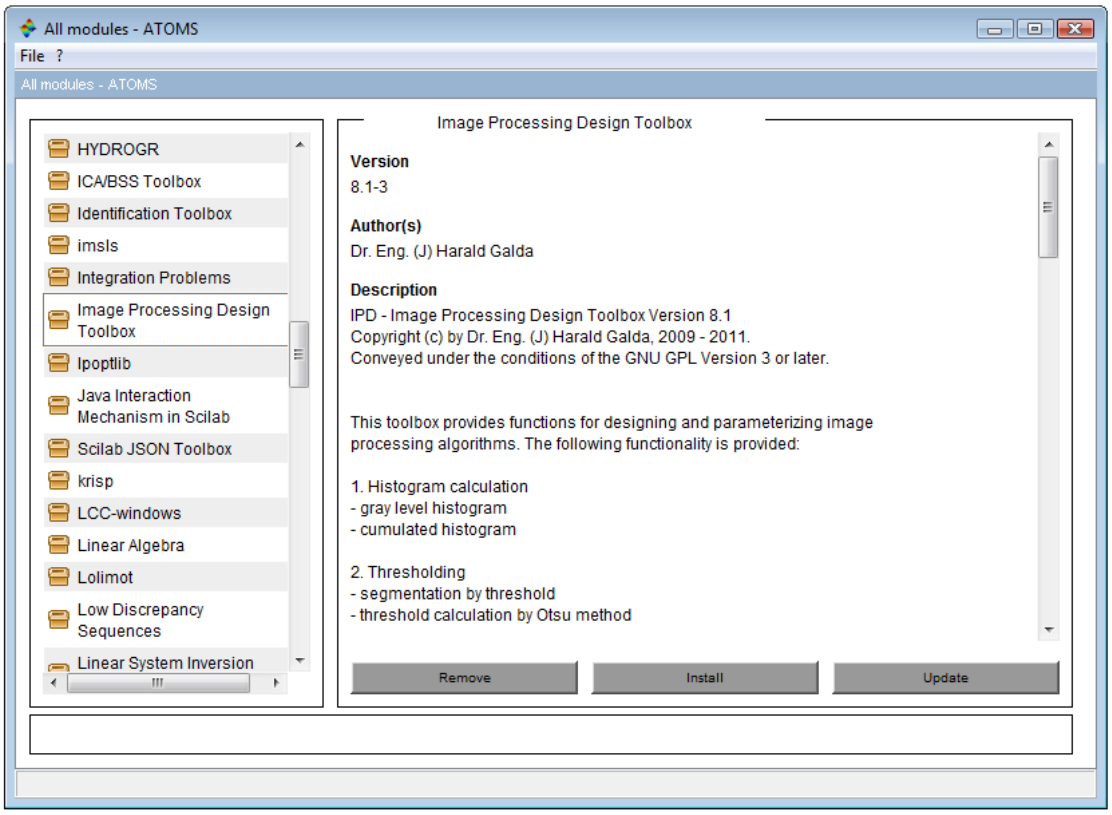
More details on ATOMS are available at [10].

2.9 Exercises

Exercise 2.1 (***The console***) Type the following statement in the console.

```
atoms
```

Now type on the <Tab> key. What happens? Now type the "I" letter, and type again on <Tab>. What happens?



Version
8.1-3

Author(s)
Dr. Eng. (J) Harald Galda

Description
IPD - Image Processing Design Toolbox Version 8.1
Copyright (c) by Dr. Eng. (J) Harald Galda, 2009 - 2011.
Conveyed under the conditions of the GNU GPL Version 3 or later.

This toolbox provides functions for designing and parameterizing image processing algorithms. The following functionality is provided:

1. Histogram calculation
- gray level histogram
- cumulated histogram

2. Thresholding
- segmentation by threshold
- threshold calculation by Otsu method

Figure 19: The ATOMS graphical user interface.

Exercise 2.2 (***Using exec***) When we develop a Scilab script, we often use the `exec` function in combination with the `ls` function, which displays the list of files and directories in the current directory. We can also use the `pwd`, which displays the current directory. The `SCI` variable contains the name of the directory of the current Scilab installation. We use it very often to execute the scripts which are provided in Scilab. Type the following statements in the console and see what happens.

```
pwd
SCI
ls(SCI+"/modules")
ls(SCI+"/modules/graphics/demos")
ls(SCI+"/modules/graphics/demos/2d_3d_plots")
dname = SCI+"/modules/graphics/demos/2d_3d_plots";
filename = fullfile(dname,"contourf.dem.sce");
exec(filename)
exec(filename);
```

3 Basic elements of the language

Scilab is an interpreted language, which means that we can 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.

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

Figure 20: Scilab elementary mathematical operators.

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 with the right hand side (it is not the comparison operator, which syntax is associated with 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 20 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. It will be reviewed again in section 4.12, which deals with the conjugate transpose of a matrix.

3.2 Variable names

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 digits from "`0`" to "`9`" are allowed, with the additional characters "`%`", "`_`", "`#`", "`!`", "`$`", "`?`". Notice though that variable names, whose first letter is "`%`", have a special meaning in Scilab, as we will see in section 3.5, which presents the pre-defined mathematical 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 and subsequent lines 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..
-->+4
 x  =
    10.
```

See the section 3.13 for more details on this topic.

3.4 Elementary mathematical functions

Tables 21 and 22 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. As a consequence, we can 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
```

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 21: Scilab elementary mathematical functions: trigonometry.

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

Figure 22: Scilab elementary mathematical functions: other functions.

```
 ans  =
    1.
```

3.5 Pre-defined mathematical variables

In Scilab, several mathematical variables are pre-defined variables, whose names begin with a percent "%" character. The variables which have a mathematical meaning are summarized in figure 23.

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

```
-->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.

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

Figure 23: 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 24: Comparison operators.

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`. Figure 24 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). The comparison operators are reviewed in section 4.14, where the emphasis is made on comparison of matrices.

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

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

3.7 Complex numbers

Scilab provides complex numbers, which are stored as pairs of floating point numbers. The pre-defined 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. Figure 25 presents functions which 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
```

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

Figure 25: Scilab complex numbers elementary functions.

`int8`	`int16`	`int32`
`uint8`	`uint16`	`uint32`

Figure 26: Scilab integer data types.

```
-->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 Integers

We can create various types of integer variables with Scilab. The functions that create such integers are presented in figure 26.

In this section, we first review the basic features of integers, which are associated with a particular range of values. Then we analyze the conversion between integers. In the final section, we consider the behaviour of integers at the boundaries and focus on portability issues.

3.8.1 Overview of integers

There is a direct link between the number of bits used to store an integer and the range of values that the integer can manage. The range of an integer variable depends on the number of its bits.

- An n-bit signed integer takes its values from the range $[-2^{n-1}, 2^{n-1}-1]$.

`y=int8(x)`	a 8-bit signed integer in $[-2^7, 2^7 - 1] = [-128, 127]$
`y=uint8(x)`	a 8-bit unsigned integer in $[0, 2^8 - 1] = [0, 255]$
`y=int16(x)`	a 16-bit signed integer in $[-2^{15}, 2^{15} - 1] = [-32768, 32767]$
`y=uint16(x)`	a 16-bit unsigned integer in $[0, 2^{16} - 1] = [0, 65535]$
`y=int32(x)`	a 32-bit signed integer in $[-2^{31}, 2^{31} - 1]$
`y=uint32(x)`	a 32-bit unsigned integer in $[0, 2^{32} - 1] = [0, 4294967295]$

Figure 27: Scilab integer functions.

iconvert	conversion to integer representation
inttype	type of integers

Figure 28: Scilab integer conversion functions.

- An n-bit unsigned integer takes its values from the range $[0, 2^n - 1]$.

For example, an 8-bit signed integer, as created by the `int8` function, can store values in the range $[-2^7, 2^7 - 1]$, which simplifies to $[-128, 127]$. The map from the type of integer to the corresponding range of values is presented in figure 27.

In the following session, we check that an unsigned 32-bit integer has values inside the range $[0, 2^{32} - 1]$, which simplifies to $[0, 4294967295]$.

```
-->format(25)
-->n=32
 n  =
    32.
-->2^n - 1
 ans  =
    4294967295.
-->i = uint32(0)
 i  =
  0
-->j=i-1
 j  =
  4294967295
-->k = j+1
 k  =
  0
```

3.8.2 Conversions between integers

There are functions which convert to and from integer data types. These functions are presented in figure 28.

The `inttype` function inquires about the type of an integer variable. Depending on the type, the function returns a corresponding value, as summarised in table 29.

When two integers are added, the types of the operands are analyzed: the resulting integer type is the larger, so that the result can be stored. In the following script, we create an 8-bit integer `i` (which is associated with `inttype`=1) and a

inttype(x)	Type
1	8-bit signed integer
2	16-bit signed integer
4	32-bit signed integer
11	8-bit unsigned integer
12	16-bit unsigned integer
14	32-bit unsigned integer

Figure 29: Types of integers returned by the `inttype` function.

16-bit integer `j` (which is associated with `inttype`=2). The result is stored in `k`, a 16-bit signed integer.

```
-->i=int8(1)
 i  =
  1
-->inttype(i)
 ans  =
    1.
-->j=int16(2)
 j  =
  2
-->inttype(j)
 ans  =
    2.
-->k=i+j
 k  =
  3
-->inttype(k)
 ans  =
    2.
```

3.8.3 Circular integers and portability issues

The behaviour of integers at the range boundaries deserves a particular analysis, since it is different from software to software. In Scilab, the behaviour is *circular*, that is, if an integer at the upper limit is incremented, the next value is at the lower limit. An example of circular behaviour is given in the following session, where

```
-->uint8(0+(-4:4))
 ans  =
  252  253  254  255  0  1  2  3  4
-->uint8(2^8+(-4:4))
 ans  =
  252  253  254  255  0  1  2  3  4
-->int8(2^7+(-4:4))
 ans  =
  124  125  126  127 -128 -127 -126 -125 -124
```

This is in contrast with other mathematical packages, such as Octave or Matlab. In these packages, if an integer is at the upper limit, the next integer stays at the upper limit. In the following Octave session, we execute the same computations as previously.

```
octave-3.2.4.exe:1> uint8(0+(-4:4))
ans =
    0   0   0   0   0   1   2   3   4
octave-3.2.4.exe:5> uint8(2^8+(-4:4))
ans =
   252   253   254   255   255   255   255   255   255
octave-3.2.4.exe:2> int8(2^7+(-4:4))
ans =
   124   125   126   127   127   127   127   127   127
```

The Scilab circular way gives a greater flexibility in the processing of integers, since we can write algorithms with fewer `if` statements. But these algorithms must be checked, particularly if they involve the boundaries of the integer range. Moreover, translating a script from another computation system into Scilab may lead to different results.

3.9 Floating point integers

In Scilab, the default numerical variable is the double, that is the 64-bit floating point number. This is true even if we write what is mathematically an integer. In [12], Cleve Moler call this number a "flint", a short for floating point integer. In practice, we can safely store integers in the interval $[-2^{52}, 2^{52}]$ into doubles. We emphasize that, provided that all input, intermediate and output integer values are strictly inside the $[-2^{52}, 2^{52}]$ interval, the integer computations are exact. For example, in the following example, we perform the exact addition of two large integers which remain in the "safe" interval.

```
-->format(25)
-->a=  2^40 - 12
 a  =
    1099511627764.
-->b=  2^45 + 3
 b  =
    35184372088835.
-->c = a + b
 c  =
    36283883716599.
```

Instead, when we perform computations outside this interval, we may have unexpected results. For example, in the following session, we see that additions involving terms slightly greater than 2^{53} produce only even values.

```
-->format(25)
-->(2^53 + (1:10))'
 ans  =
    9007199254740992.
    9007199254740994.
    9007199254740996.
    9007199254740996.
    9007199254740996.
    9007199254740998.
```

```
      9007199254741000.
      9007199254741000.
      9007199254741000.
      9007199254741002.
```

In the following session, we compute 2^{52} using the floating point integer 2 in the first case, and using the 16-bit integer 2 in the second case. In the first case, no overflow occurs, even if the number is at the limit of 64-bit floating point numbers. In the second case, the result is completely wrong, because the number 2^{52} cannot be represented as a 16-bit integer.

```
-->2^52
 ans  =
    4503599627370496.
-->uint16(2^52)
 ans  =
  0
```

In section 4.15, we analyze the issues which arise when indexes involved to access the elements of a matrix are doubles.

3.10 The ans variable

Whenever we make a computation and do not store the result into an output variable, the result is stored in the default `ans` variable. Once it is defined, we can use this variable as any other Scilab variable.

In the following session, we compute exp(3) so that the result is stored in the `ans` variable. Then we use its content as a regular variable.

```
-->exp(3)
 ans  =
    20.085536923187666792368
-->t = log(ans)
 t  =
    3.
```

In general, the `ans` variable should be used only in an interactive session, in order to progress in the computation without defining a new variable. For example, we may have forgotten to store the result of an interesting computation and do not want to recompute the result. This might be the case after a long sequence of trials and errors, where we experimented several ways to get the result without taking care of actually storing the result. In this interactive case, using `ans` may save some human (or machine) time. Instead, if we are developing a script used in a non-interactive way, it is a bad practice to rely on the `ans` variable and we should store the results in regular variables.

3.11 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
```

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

3.12 Dynamic type of variables

When we create and manage variables, Scilab changes the variable type 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.

3.13 Notes and references

The coding style that we presented in this section and that we will use in the remaining of the document is standard in the context of Scilab. This style is based on common practices and on the document [9], which defines code conventions for the Scilab programming language. The convention defines the way of defining new functions (e.g. their names, the name of the arguments, etc...), the indentation style, the quotes, the line length, the loop indices, the number of statements per line, and many other details.

In the section 3.3, we presented the way to define comments and continuation lines and stated that any line which ends with two dots is a continuation line. In fact, the interpreter considers that any line which ends with more that two dots is a continuation line. This implies that we can use continuation lines with three dots, or more. This feature is maintained only for backward compatibility, does not correspond to the Scilab coding style and should not be used in new Scilab scripts.

In the section 3.11, we presented the double quote "`"`" to defined strings. In fact, the interpreter can also use strings defined with single quotes. But this may lead to bugs, because there might be a confusion with the transpose operator (see section 4.12). This feature is maintained for backward compatibility and should not be used in new scripts.

3.14 Exercises

Exercise 3.1 (*Precedence of operators*) What are the results of the following computations (think about it before trying in Scilab) ?

```
2 * 3 + 4
2 + 3 * 4
2 / 3 + 4
2 + 3 / 4
```

Exercise 3.2 (*Parentheses*) What are the results of the following computations (think about it before trying in Scilab) ?

```
2 * (3 + 4)
(2 + 3) * 4
(2 + 3) / 4
3 / (2 + 4)
```

Exercise 3.3 (*Exponents*) What are the results of the following computations (think about it before trying in Scilab) ?

```
1.23456789d10
1.23456789e10
1.23456789e-5
```

Exercise 3.4 (*Functions*) What are the results of the following computations (think about it before trying in Scilab) ?

```
sqrt(4)
sqrt(9)
sqrt(-1)
sqrt(-2)
exp(1)
log(exp(2))
exp(log(2))
10^2
log10(10^2)
10^log10(2)
sign(2)
sign(-2)
sign(0)
```

Exercise 3.5 (*Trigonometry*) What are the results of the following computations (think about it before trying in Scilab) ?

```
cos(0)
sin(0)
cos(%pi)
sin(%pi)
cos(%pi/4) - sin(%pi/4)
```

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 the 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, with the goal of performing linear algebra operations within a high-level language.

By design, Scilab was created to be able to perform matrix 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, subtraction, 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 developer should always use high-level operations, so that each statement processes 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.

`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 30: Functions which generate matrices.

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 in 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 "`...`" denotes intermediate values:

```
A = [a11 , a12 , ... , a1n; ... ; 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:

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

This lightens 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 create matrices from a given size, i.e. from a given number of rows and columns. These functions are presented in figure 30. The most commonly used are `eye`, `zeros` and `ones`. These commands take 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 deletes the content of a matrix, so that the associated memory is freed.

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

4.4 Query matrices

The functions in figure 31 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 used depending on if the matrix is 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 $nr, nc > 1$.

The `size` function has also the following syntax

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

which gets 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,

`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 31: Functions which query or modify matrices.

- `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 subscript indices 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 subscripts.

We emphasize that, by default, the first subscript of a matrix is 1. This contrasts with other languages, such as the C language for instance, where the first subscript 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 subscript i and j satisfy $1 \leq i \leq nr$ and $1 \leq j \leq nc$. If the subscript 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 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 subscript and `j` is the ending subscript with $i \leq j$. This creates the vector $v = (i, i+1, \ldots, j)$. In the following session, we create a vector of subscripts from `2` to `4` in one statement.

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

The complete syntax configures the increment used when generating the subscripts, i.e. the *step*. The complete syntax for the colon operator is

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

where `i` is the starting subscript, `j` is the ending subscript 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 subscript in the vector of subscripts 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 subscripts 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 than $j = 10$.

In the following session, we present two examples where the step is negative. In the first case, the colon operator generates decreasing subscripts 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 at the same time.

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

With a vector of subscripts, we can access 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 subscripts. 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 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 subscripts are the result of a computation. For example, the algorithm may be based on a loop where the subscripts are updated regularly. In these cases, the syntax

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

where vi,vj are vectors of subscripts, 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  =
```

`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 32: Access to a matrix with the colon ":" operator.

```
      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 figure 32 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([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])`.

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

4.7 The `eye` matrix

The `eye` function creates the identity matrix with the size which depends on the context. Its name has been chosen in place of `I` in order to avoid the confusion with a subscript or with the imaginary number.

In the following session, we add 3 to the diagonal elements of the matrix `A`.

```
-->A = ones(3,3)
 A  =
    1.    1.    1.
    1.    1.    1.
    1.    1.    1.
-->B = A + 3*eye()
 B  =
    4.    1.    1.
    1.    4.    1.
    1.    1.    4.
```

In the following session, we define an identity matrix `B` with the `eye` function depending on the size of a given matrix `A`.

```
-->A = ones(2,2)
 A  =
    1.    1.
    1.    1.
-->B = eye(A)
 B  =
    1.    0.
    0.    1.
```

Finally, we can use the `eye(m,n)` syntax in order to create an identity matrix with `m` rows and `n` columns.

4.8 Matrices are dynamic

The size of a matrix can grow or reduce dynamically. This adapts the size of the matrix to the data it contains.

Consider the following session where we define a 2×3 matrix.

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

In the following session, we insert the value 7 at the indices $(3,1)$. This creates the third row in the matrix, sets the $A(3,1)$ entry to 7 and fills the other values of the newly created row with zeros.

```
-->A(3,1) = 7
 A  =
    1.     2.     3.
    4.     5.     6.
    7.     0.     0.
```

The previous example showed that matrices can grow. In the following session, we see that we can also reduce the size of a matrix. This is done by using the empty matrix "`[]`" operator in order to delete the third column.

```
-->A(:,3) = []
 A  =
    1.     2.
    4.     5.
    7.     0.
```

We can also change the shape of the matrix with the `matrix` function. The `matrix` function reshapes a source matrix into a target matrix with a different size. The transformation is performed column by column, by stacking the elements of the source matrix. In the following session, we reshape the matrix `A`, which has $3 \times 2 = 6$ elements into a row vector with 6 columns.

```
-->B = matrix(A,1,6)
 B  =
    1.     4.     7.     2.     5.     0.
```

4.9 The dollar "$" operator

Usually, we make use of subscript to make reference *from the start* of a matrix. By opposition, the dollar "`$`" operator references elements *from the end* of the matrix. The "`$`" operator signifies "the subscript corresponding to the last" row or column, depending on the context. This syntax is associated with an algebra, so that the subscript `$-i` corresponds to the subscript $\ell - i$, where ℓ is the number of corresponding rows or columns. Various uses of the dollar operator are presented in figure 33.

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

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

Figure 33: Access to a matrix with the dollar "$" operator. The "$" operator signifies "the last subscript".

The "$" operator is used most of the time in the context of the "$+1" statement, which *adds to the end* of a matrix. This can be convenient, since it avoids the need of updating the number of rows or columns continuously; it should be used with care, only in situations where the number of rows or columns cannot be known in advance. The reason is that the interpreter has to internally re-allocate memory for the entire matrix and to copy the old values to the new destination. This can lead to performance penalties and this is why we should be warned against bad uses of this operator. All in all, the only good use of the "$+1" statement is when we do not know in advance the final number of rows or columns.

4.10 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 value of this scalar is added to each element of the second matrix. This feature is shown in the following session.

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

+	addition	.+	elementwise addition
-	subtraction	.-	elementwise subtraction
*	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 34: Matrix operators and elementwise operators.

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

The addition is possible only if the two matrices are conformable to addition. In the following session, we try to 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.
-->A+B
    !--error 8
Inconsistent addition.
```

Elementary operators which are available for matrices are presented in figure 34. 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.

Figure 34 separates the operators which treat the matrices as a whole and the elementwise operators, which are presented in the next section.

4.11 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 results are 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.
```

4.12 Conjugate transpose and non-conjugate transpose

There might be some confusion when the elementwise single quote " `.'` " and the regular single quote " `'` " operators are used without a careful knowledge of their exact definitions. With a matrix of doubles containing real values, the single quote " `'` " operator only transposes the matrix. Instead, when a matrix of doubles containing complex values is used, the single quote " `'` " operator transposes *and conjugates* the matrix. Hence, the operation `A=Z'` produces a matrix with entries $A_{jk} = X_{kj} - iY_{kj}$, where i is the imaginary number such that $i^2 = -1$ and X and Y are the real and imaginary parts of the matrix Z. The elementwise single quote " `.'` " always transposes without conjugating the matrix, be it real or complex. Hence, the operation `A=Z.'` produces a matrix with entries $A_{jk} = X_{kj} + iY_{kj}$.

In the following session, an non-symetric matrix of doubles containing complex values is used, so that the difference between the two operators is obvious.

```
-->A = [1 2;3 4] + %i * [5 6;7 8]
 A  =
    1. + 5.i    2. + 6.i
    3. + 7.i    4. + 8.i
-->A'
 ans  =
    1. - 5.i    3. - 7.i
    2. - 6.i    4. - 8.i
-->A.'
 ans  =
    1. + 5.i    3. + 7.i
    2. + 6.i    4. + 8.i
```

In the following session, we define an non-symetric matrix of doubles containing real values and see that the results of the " `'` " and " `.'` " are the same in this particular case.

```
-->B = [1 2;3 4]
 B  =
    1.    2.
    3.    4.
```

```
-->B'
 ans  =
    1.    3.
    2.    4.
-->B.'
 ans  =
    1.    3.
    2.    4.
```

Many bugs are created due to this confusion, so that it is mandatory to ask yourself the following question: what happens if my matrix is complex? If the answer is "I want to transpose only", then the elementwise quote " `.'` " operator is to be used.

4.13 Multiplication of two vectors

Let $\mathbf{u} \in \mathbb{R}^n$ be a column vector and $\mathbf{v}^T \in \mathbb{R}^n$ be a column vector. The matrix $A = \mathbf{u}\mathbf{v}^T$ has entries $A_{ij} = u_i v_j$. In the following Scilab session, we multiply the column vector `u` by the row vector `v` and store the result in the variable `A`.

```
-->u = [1
-->2
-->3]
 u  =
    1.
    2.
    3.
-->v = [4 5 6]
 v  =
    4.    5.    6.
-->u*v
 ans  =
    4.     5.     6.
    8.     10.    12.
    12.    15.    18.
```

This might lead to some confusion because linear algebra textbooks consider column vectors only. We usually denote by $\mathbf{u} \in \mathbb{R}^n$ a column vector, so that the corresponding row vector is denoted by $\mathbf{u}^T$. In the associated Scilab implementation, a row vector can be directly stored in the variable `u`. It might also be a source of bugs, if the expected vector is expected to be a row vector and is, in fact, a column vector. This is why any algorithm which works only on a particular type of matrix (row vector or column vector) should check that the input vector has indeed the corresponding shape and generate an error if not.

4.14 Comparing two real matrices

Comparison of two matrices is only possible when the matrices have the same shape. The comparison operators presented in figure 24 are indeed performed when the input arguments `A` and `B` are matrices. When two matrices are compared, the result is a matrix of booleans. This matrix can then be combined with operators such as `and`, `or`, which are presented in figure 35. The usual operators "`&`", "`|`" are also

`and(A,"r")`	rowwise ”and”
`and(A,"c")`	columnwise ”and”
`or(A,"r")`	rowwise ”or”
`or(A,"c")`	columnwise ”or”

Figure 35: Special comparison operators for matrices. The usual operators ”<”, ”&”, ”|” are also available for matrices, but the `and` and `or` functions allow performing rowwise and columnwise operations.

available for matrices, but the `and` and `or` allow performing rowwise and columnwise operations.

In the following Scilab session, we create a matrix `A` and compare it against the number `3`. Notice that this comparison is valid because the number `3` is compared element by element against `A`. We then create a matrix `B` and compare the two matrices `A` and `B`. Finally, the `or` function is used to perform a rowwise comparison so that we get the columns where one value in the column of the matrix `A` is greater than one value in the column of the matrix `B`.

```
-->A = [1 2 7
-->6 9 8]
 A  =
    1.     2.     7.
    6.     9.     8.
-->A>3
 ans  =
  F F T
  T T T
-->B=[4 5 6
-->7 8 9]
 B  =
    4.     5.     6.
    7.     8.     9.
-->A>B
 ans  =
  F F T
  F T F
-->or(A>B,"r")
 ans  =
  F T T
```

4.15 Issues with floating point integers

In this section, we analyze the problems which arise when we use integers which are stored as floating point numbers. If used without caution, these numbers can lead to disastrous results, as we are going to see.

Assume that the matrix `A` is a square 2×2 matrix.

```
-->A = testmatrix("hilb",2)
 A  =
    4.   - 6.
  - 6.     12.
```

In order to access the element `(2,1)` of this matrix, we can use a constant subscript, such as `A(2,1)`, which is safe. In order to access the element of the matrix, we can use variables `i` and `j` and use the statement `A(i,j)`, as in the following session.

```
-->i = 2
 i  =
    2.
-->j = 1
 j  =
    1.
-->A(i,j)
 ans  =
  - 6.
```

In the previous session, we emphasize that the variables `i` and `j` are doubles, that is, binary floating point values. This is why the following statement is valid.

```
-->A( 2 , [1.0 1.1 1.5 1.9] )
 ans  =
  - 6.  - 6.  - 6.  - 6.
```

The previous session shows that the floating point values 1.0, 1.1, 1.5 and 1.9 are all converted to the integer 1, as if the `floor` function had been used to convert the floating point number into an integer. Indeed, the `floor` function returns the floating point number storing the integer part of the given floating point number: in some sense, it rounds toward zero. For example, `floor(1.0)`, `floor(1.1)`, `floor(1.5)` and `floor(1.9)` all return 1.

This is what makes this language both simple and efficient. But it can also have unfortunate consequences, sometimes leading to unexpected results. For example, consider the following session.

```
-->ones(1,1)
 ans  =
    1.
-->ones(1,(1-0.9)*10)
 ans  =
     []
```

If the computations were performed in exact arithmetic, the result of $(1-0.9)*10$ is equal to 1. Instead, the statement `ones(1,(1-0.9)*10)` creates an empty matrix, because the floating point result of the expression `(1-0.9)*10` is not exactly equal to 1. In the following session, we check that the integer part of `(1-0.9)*10` is 0.

```
-->floor((1-0.9)*10)
 ans  =
    0.
```

Indeed, the decimal number 0.9 cannot be exactly represented as a double precision floating point number. This leads to a rounding, so that the floating point representation of 1-0.9 is slightly smaller than 0.1. When the multiplication `(1-0.9)*10` is performed, the floating point result is therefore slightly smaller than 1, as presented in the following session.

```
-->format(25)
-->1-0.9
 ans  =
    0.09999999999999997779554
-->(1-0.9)*10
 ans  =
    0.99999999999999997779554
```

Then the floating point number `0.999999999999999` is considered as the integer zero, which makes the `ones` function return an empty matrix. The origin of this issue is the use of the binary floating point number representing 0.1, which must be used with caution.

There is a way to fix this issue, by forcing the way that the expression is rounded to the integer value. For example, we can use the `round` function prior to the call to the `ones` function.

```
-->n=round((1-0.9)*10);
-->ones(1,n)
 ans  =
    1.
```

Indeed, the `round` function rounds to the nearest integer. In the following session, we check that `n` is exactly equal to 1.

```
-->n==1
 ans  =
  T
```

It is an unfortunate effect of the use of doubles which creates this kind of issues. On the other hand, this simplifies most expressions, so that this turns out to be a major advantage in most situations. We could as well blame the interpretor for silently converting a double with a fractional content into an integer, without generating (at least) a warning. In practice, it is a safe practice to round a double prior to the call of a function which actually expects an integer value.

4.16 More on elementary functions

In this section, we analyse several elementary functions, especially degree-based trigonometry functions, logarithm functions and matrix-based elementary functions.

Trigonometry functions such as `sin` and `cos` are provided with the classical input argument in radian. But some other trigonometry functions, such as the `cosd` function for example, are taking an input argument in degree. This means that, in the mathematical sense $\text{tand}(x) = \tan(x\pi/180)$. These functions can be easily identified because their name ends with the letter "`d`", e.g. `cosd`, `sind` among others. The key advantage for the degree-based elementary functions is that they provide exact results when their argument has special mathematical values, such as multiples of $90°$. Indeed, the implementation of the degree-based functions is based on an argument reduction which is exact for integer values. This gets exact floating point results for special cases.

In the following session, we compute $\sin(\pi)$ and $\text{sind}(180)$, which are mathematically equal, but are associated with different floating point results.

```
-->sin(%pi)
 ans  =
    1.225D-16
```

```
-->sind(180)
 ans  =
    0.
```

The fact that $\sin(\pi)$ is not exactly zero is associated with the limited precision of floating point numbers. Indeed, the argument π is stored in memory with a limited number of significant digits, which leads to rounding. Instead, the argument 180 is represented exactly as a floating point number, because it is a small integer. Hence, the value of `sind(180)` is computed by the `sind` function as `sin(0)`. Once again, the number zero is exactly represented by a floating point number. Moreover, the sin function is represented in the $[-\pi/2, \pi/2]$ interval by a polynomial of the form $p(x) = x + x^3 q(x^2)$ where q is a low degree polynomial. Hence, we get `sind(180)=sin(0)=0`, which is the exact result.

The `log` function computes the natural logarithm of the input argument, that is, the inverse of the function `exp`$= e^x$, where e is Euler's number. In order to compute the logarithm function for other bases, we can use the functions `log10` and `log2`, associated with bases 10 and 2 respectively. In the following session, we compute the values of the `log`, `log10` and `log2` functions for some specific values of x.

```
-->x = [exp(1) exp(2) 1 10 2^1 2^10]
 x  =
    2.7182818    7.3890561    1.    10.    2.    1024.
-->[x' log(x') log10(x') log2(x')]
 ans  =
    2.7182818    1.           0.4342945    1.442695
    7.3890561    2.           0.8685890    2.8853901
    1.           0.           0.           0.
    10.          2.3025851    1.           3.3219281
    2.           0.6931472    0.30103      1.
    1024.        6.9314718    3.0103       10.
```

The first column in the previous table contains various values of x. The column number 2 contains various values of `log(x)`, while the columns 3 and 4 contains various values of `log10(x)` and `log2(x)`.

Most functions are elementwise, that is, given an input matrix, apply the same function for each entry of the matrix. Still, some functions have a special meaning with respect to linear algebra. For example, the matrix exponential of a function is defined by $e^X = \sum_{k=0,\infty} \frac{1}{k!} X^k$, where X is a square $n \times n$ matrix. In order to compute the exponential of a matrix, we can use the `expm` function. Obviously, the elementwise exponential function `exp` does not return the same result. More generally, the functions which have a special meaning with respect to matrices have a name which ends with the letter "`m`", e.g. `expm`, `sinm`, among others. In the following session, we define a 2×2 matrix containing specific multiples of $\pi/2$ and use the `sin` and `sinm` functions.

```
-->A = [%pi/2 %pi; 2*%pi 3*%pi/2]
 A  =
    1.5707963    3.1415927
    6.2831853    4.712389
-->sin(A)
 ans  =
    1.           1.225D-16
```

`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 36: Some common functions for linear algebra.

```
  - 2.449D-16   - 1.
-->sinm(A)
 ans  =
  - 0.3333333    0.6666667
    1.3333333    0.3333333
```

4.17 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. Figure 36 presents a list of the most common linear algebra functions.

4.18 Exercises

Exercise 4.1 (***Plus one***) Create the vector $(x_1+1, x_2+1, x_3+1, x_4+1)$ with the following x.

```
x = 1:4;
```

Exercise 4.2 (***Vectorized multiplication***) Create the vector $(x_1y_1, x_2y_2, x_3y_3, x_4y_4)$ with the following x and y.

```
x = 1:4;
y = 5:8;
```

Exercise 4.3 (***The infamous dot***) Analyze the following session and explain why we might not get the expected result.

```
-->expected=[1/2 1/3 1/4]
 expected  =
    0.5    0.3333333    0.25
-->1./[2 3 4]
```

```
 ans  =
    0.0689655
    0.1034483
    0.1379310
```

Exercise 4.4 (***Vectorized invert***) Create the vector $\left(\frac{1}{x_1}, \frac{1}{x_2}, \frac{1}{x_3}, \frac{1}{x_4}\right)$ with the following x.

```
x = 1:4;
```

Exercise 4.5 (***Vectorized division***) Create the vector $\left(\frac{x_1}{y_1}, \frac{x_2}{y_2}, \frac{x_3}{y_3}, \frac{x_4}{y_4}\right)$ with the following x and y.

```
x = 12*(6:9);
y = 1:4;
```

Exercise 4.6 (***Vectorized squaring***) Create the vector $\left(x_1^2, x_2^2, x_3^2, x_4^2\right)$ with $x = 1, 2, 3, 4$.

Exercise 4.7 (***Vectorized sinus***) Create the vector $(sin(x_1), sin(x_2), \ldots, sin(x_{10}))$ with x is a vector of 10 values linearly chosen in the interval $[0, \pi]$.

Exercise 4.8 (***Vectorized function***) Compute the $y = f(x)$ values of the function f defined by the equation

$$f(x) = \log_{10}\left(r/10^x + 10^x\right) \qquad (1)$$

with $r = 2.220.10^{-16}$ and x a vector of 100 values linearly chosen in the interval $[-16, 0]$.

5 Looping and branching

In this section, we describe how to make conditional statements, that is, we present the `if` statement. We present the `select` statement, which creates more complex selections. We present Scilab loops, that is, we present the `for` and `while` statements. We finally present two main tools to manage loops, that is, the `break` and `continue` statements.

5.1 The `if` statement

The `if` statement performs 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 performs 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 "=" operator sets 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 as needed, and this creates complicated branches as required. But if there are many `elseif` statements required, this may imply that a `select` statement should be used instead.

5.2 The `select` statement

The `select` statement combines several branches in a clear and simple way. Depending on the value of a variable, it performs 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.

The `else` statement is optional, but is considered a good programming practice. Indeed, even if the programmer thinks that the associated case cannot happen, there may still exist a bug in the logic, so that all the conditions are false while they should not. In this case, if the `else` statement does not interrupt the execution, the remaining statements in the script will be executed. This can lead to unexpected results. In the worst scenario, the script *still works* but with inconsistent results. Debugging such scripts is extremely difficult and may lead to a massive loss of time.

Therefore, the `else` statement should be included in most `select` sequences. In order to manage these unexpected events, we often combine a `select` statement with the `error` function.

The `error` function generates an error associated with the given message. When an error is generated, the execution is interrupted and the interpreter quits all the functions. The call stack is therefore cleared and the script stops.

In the following script, we display a message depending on the the value of the positive variable `i`. If that variable is negative, we generate an error.

```
i = -5;
select i
case 1
  disp("One")
case 2
  disp("Two")
case 3
  disp("Three")
else
  error ( "Unexpected value of the parameter i" )
end
```

The previous script produces the following output.

```
-->i = -5;
-->select i
```

```
-->case 1
-->  disp("One")
-->case 2
-->  disp("Two")
-->case 3
-->  disp("Three")
-->else
-->  error ( "Unexpected value of the parameter i" )
Unexpected value of the parameter i
```

In practice, when we see a `select` statement without the corresponding `else`, we may wonder if the developer wrote this on purpose or based on the assumption that *it will never happen*. Most of the time, this assumption can be discussed.

5.3 The `for` statement

The `for` statement performs loops, that is, it performs a given action several times. Most of the time, a loop is performed over integer values, which go from a starting to an ending subscript. We will see, at the end of this section, that the `for` statement is in fact much more general, as it can allow looping 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 subscripts `[1 2 3 4 5]`. The following session shows that the statement `1:5` produces all the required integer values in 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 display 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 than what we have previously used in this section. Indeed, it browses 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 performs 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 as 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 interrupts a loop. Usually, we use this statement in loops where, once some condition is satisfied, the loops should not be continued.

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 makes the interpreter go 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 goes on to the next loop.

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

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

```
-->s
 s  =
    25.
-->i
 i  =
    10.
```

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 the same 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. We also present how to manage input and output arguments. Finally, we present how to debug a function using the `pause` statement.

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 parentheses, while the output arguments are surrounded by opening and closing *square* brackets.

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 37: 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` checks 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 37. 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`, multiplies it by 2, and returns the value in the output argument `y`.

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

The statement `function y = myfunction ( x )` is the *header* of the function while the *body* of the function is made of the statement `y = 2 * x`. The body of a function may contain one, two or more statements.

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 "`examples-functions.sce`", in the "`C:\myscripts`" directory. 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 character "`;`" 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.

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 makes us master the latter. Modules will not be described in this document. Still, in many practical situations, function libraries allow efficient management of 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 additional indexing files.

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

Before analyzing an example, let us consider some general rules which must be followed 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 (implicitly) private functions.

The name of the `.sci` file 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 manage libraries are presented in figure 38.

`genlib`	build library from functions in a given directory
`lib`	library definition

Figure 38: Scilab commands to manage functions.

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 every time 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 inconvenient to manually load 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-5.3.1
```

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 zero or more 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 follows:

```
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

`whereami`	display current instruction calling tree
`where`	get current instruction calling tree

Figure 39: Scilab commands associated with the call stack.

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 more advanced topic will not be detailed in this document.

6.5 Levels in the call stack

Obviously, function calls can be nested, i.e. a function `f` can call a function `g`, which in turn calls a function `h` and so forth. When Scilab starts, the variables which are defined are at the *global* scope. When we are in a function which is called from the global scope, we are one level down in the call stack. When nested function calls occur, the current level in the call stack is equal to the number of previously nested calls. The functions presented in figure 39 inquire about the state of the call stack.

In the following session, we define 3 functions which are calling one another and we use the function `whereami` to display the current instruction calling tree.

```
function y = fmain ( x )
  y = 2 * flevel1 ( x )
endfunction
function y = flevel1 ( x )
  y = 2 * flevel2 ( x )
endfunction
function y = flevel2 ( x )
  y = 2 * x
  whereami()
endfunction
```

When we call the function `fmain`, the following output is produced. As we can see, the 3 levels in the call stack are displayed, associated with the corresponding function.

```
-->fmain(1)
whereami  called at line 3 of macro flevel2
flevel2   called at line 2 of macro flevel1
flevel1   called at line 2 of macro fmain
 ans  =
    8.
```

In the previous example, the various calling levels are the following:

- level 0 : the global level,
- level -1 : the body of the `fmain` function,
- level -2 : the body of the `flevel1` function,
- level -3 : the body of the `flevel2` function.

These calling levels are displayed in the prompt of the console when we interactively debug a function with the `pause` statement or with breakpoints.

6.6 The `return` statement

Inside the body of a function, the `return` statement immediately stops the function, i.e. it immediately quits the current function. This statement can be used in cases where the remaining of the algorithm is not necessary.

The following function computes the sum of integers from `istart` to `iend`. In regular situations, it uses the `sum` function to perform its job. But if the `istart` variable is negative or if the `istart<=iend` condition is not satisfied, the output variable `y` is set to 0 and the function immediately returns.

```
function y = mysum ( istart , iend )
  if ( istart < 0 ) then
    y = 0
    return
  end
  if ( iend < istart ) then
    y = 0
    return
  end
  y = sum ( istart : iend )
endfunction
```

The following session checks that the `return` statement is correctly used by the `mysum` function.

```
-->mysum ( 1 , 5 )
 ans  =
    15.
-->mysum ( -1 , 5 )
 ans  =
    0.
-->mysum ( 2 , 1 )
 ans  =
    0.
```

`pause`	wait for interactive user input
`resume`	resume execution and copy some local variables
`abort`	interrupt evaluation

Figure 40: Scilab functions to debug manually a function.

Some developers state that using several `return` statements in a function is generally a bad practice. Indeed, we must take into account the increased difficulty of debugging such a function, because the algorithm may suddenly quit the body of the function. The user may get confused about what exactly caused the function to return.

This is why, in practice, the `return` statement should be used with care, and certainly not in every function. The rule to follow is that the function should return only at its very last line. Still, in particular situations, using `return` can actually greatly simplify the algorithm, while avoiding `return` would require writing a lot of unnecessary source code.

6.7 Debugging functions with `pause`

In this section, we present simple debugging methods that fix most simple bugs in a convenient and efficient way. More specifically, we present the `pause`, `resume` and `abort` statements, which are presented in figure 40.

A Scilab session usually consists in the definition of new algorithms by the creation of new functions. It often happens that a syntax error or an error in the algorithm produces a wrong result.

Consider the problem, the sum of integers from `istart` to `iend`. Again, this simple example is chosen for demonstration purposes, since the `sum` function performs it directly.

The following function `mysum` contains a bug: the second argument "`foo`" passed to the `sum` function has no meaning in this context.

```
function y = mysum ( istart , iend )
  y = sum ( iend : istart , "foo" )
endfunction
```

The following session shows what happens when we use the `mysum` function.

```
-->mysum ( 1 , 10 )
 !--error 44
Wrong argument 2.
at line       2 of function mysum called by :
mysum ( 1 , 10 )
```

In order to interactively find the problem, we place a `pause` statement inside the body of the function.

```
function y = mysum ( istart , iend )
  pause
  y = sum ( iend : istart , "foo" )
endfunction
```

We now call the function `mysum` again with the same input arguments.

```
-->mysum ( 1 , 10 )
Type 'resume' or 'abort' to return to standard level prompt.
-1->
```

We are now interactively located *in the body* of the `mysum` function. The prompt "`-1->`" indicates that the current call stack is at level -1. We can check the value of the variables `istart` and `iend` by simply typing their names in the console.

```
-1->istart
 istart  =
    1.
-1->iend
 iend  =
    10.
```

In order to progress in our function, we can copy and paste the statements and see what happens interactively, as in the following session.

```
-1->y = sum ( iend : istart , "foo" )
y = sum ( iend : istart , "foo" )
                                    !--error 44
Wrong argument 2.
```

We can see that the call to the `sum` function does not behave how we might expect. The "`foo`" input argument is definitely a bug: we remove it.

```
-1->y = sum ( iend : istart )
 y  =
    0.
```

After the first revision, the call to the `sum` function is now syntactically correct. But the result is still wrong, since the expected result in this case is 55. We see that the `istart` and `iend` variables have been *swapped.* We correct the function call and check that the fixed version behaves as expected

```
-1->y = sum ( istart : iend )
 y  =
    55.
```

The result is now correct. In order to get back to the zero level, we now use the `abort` statement, which interrupts the sequence and immediately returns to the global level.

```
-1->abort
-->
```

The "`-->`" prompt confirms that we are now back at the zero level in the call stack.
We fix the function definition, which becomes:

```
function y = mysum ( istart , iend )
  pause
  y = sum ( istart : iend )
endfunction
```

In order to check our bugfix, we call the function again.

```
-->mysum ( 1 , 10 )
Type 'resume' or 'abort' to return to standard level prompt.
-1->
```

We are now confident about our code, so that we use the resume statement, which lets Scilab execute the code as usual.

```
-->mysum ( 1 , 10 )
-1->resume
 ans  =
    55.
```

The result is correct. All we have to do is to remove the pause statement from the function definition.

```
function y = mysum ( istart , iend )
  y = sum ( istart : iend )
endfunction
```

In this section, we have seen that, used in combination, the pause, resume and abort statements are a very effective way to interactively debug a function. In fact, our example is very simple and the method we presented may appear to be too simple to be convenient. This is not the case. In practice, the pause statement has proved to be a very fast way to find and fix bugs, even in very complex situations.

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. It can create x-y plots with the plot function, contour plots with the contour function, 3D plots with the surf function, histograms with the histplot function and many other types of plots. The most commonly used plot functions are presented in figure 41.

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

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

During the creation of a plot, we use several functions in order to create the data or to configure the plot. The functions presented in figure 42 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 produce matrices of data in one function call.

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

`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 41: Scilab plot functions

`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 42: Scilab functions used when creating a plot.

```
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 an x-y plot.

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

Figure 43 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, using functions directly saves significant amount of memory space, since it avoids generating the intermediate vector `ydata`.

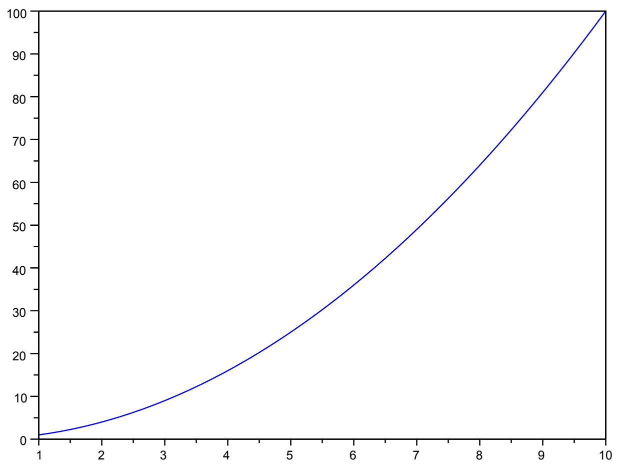


Figure 43: A simple x-y plot.

7.3 Contour plots

In this section, we present the contour plots of a multivariate function and make use of the `contour` function. This type of graphic is often used in the context of numerical optimization because it draws functions of two variables in a way that shows the location of the optimum.

Assume that we are given function f with n variables $f(\mathbf{x}) = f(x_1, \ldots, x_n)$ and $\mathbf{x} \in \mathbb{R}^n$. For a given $\alpha \in \mathbb{R}$, the equation

$$f(\mathbf{x}) = \alpha, \tag{2}$$

defines a surface in the $(n+1)$-dimensional space $\mathbb{R}^{n+1}$.

When $n = 2$, the points $z = f(x_1, x_2)$ represent a surface in the three-dimensional space $(x_1, x_2, z) \in \mathbb{R}^3$. This draws *contour* plots of the cost function, as we are going to see. For $n > 3$, though, these plots are not available. One possible solution in this case is to select two significant parameters and to draw a contour plot with these parameters varying (only).

The Scilab function `contour` plots contours of a function f. The `contour` function has the following syntax

```
contour(x,y,z,nz)
```

where

- `x` (resp. `y`) is a row vector of x (resp. y) values with size `n1` (resp. `n2`),
- `z` is a real matrix of size `(n1,n2)`, containing the values of the function or a Scilab function which defines the surface `z=f(x,y)`,
- `nz` the level values or the number of levels.

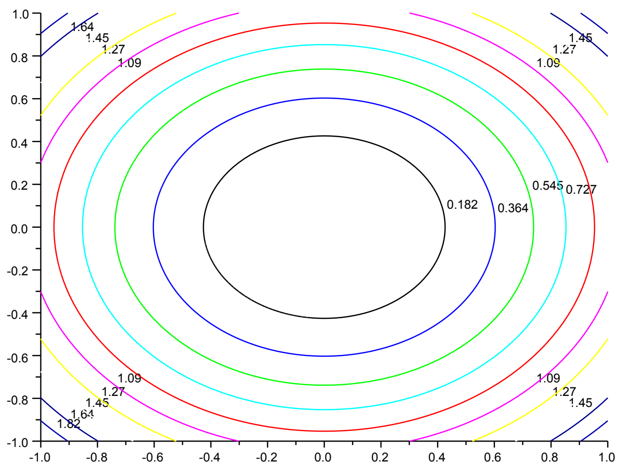


Figure 44: Contour plot of the function $f(x_1, x_2) = x_1^2 + x_2^2$.

In the following Scilab session, we use a simple form of the `contour` function, where the function `myquadratic` is passed as an input argument. The `myquadratic` function takes two input arguments x_1 and x_2 and returns $f(x_1, x_2) = x_1^2 + x_2^2$. The `linspace` function is used to generate vectors of data so that the function is analyzed in the range $[-1, 1]^2$.

```
function f = myquadratic2arg ( x1 , x2 )
  f = x1**2 + x2**2;
endfunction
xdata = linspace ( -1 , 1 , 100 );
ydata = linspace ( -1 , 1 , 100 );
contour ( xdata , ydata , myquadratic2arg , 10)
```

This produces the contour plot presented in figure 44.

In practice, it may happen that our function has the header `z = myfunction ( x )`, where the input variable `x` is a row vector. The problem is that there is only one single input argument, instead of the two arguments required by the `contour` function. There are two possibilities to solve this little problem:

- provide the data to the `contour` function by making two nested loops,
- provide the data to the `contour` function by using `feval`,
- define a new function which calls the first one.

These three solutions are presented in this section. The first goal is to let the reader choose the method which best fits the situation. The second goal is to show that performances issues can be avoided if a consistent use of the functions provided by Scilab is done.

In the following Scilab naive session, we define the function `myquadratic1arg`, which takes one vector as its single input argument. Then we perform two nested

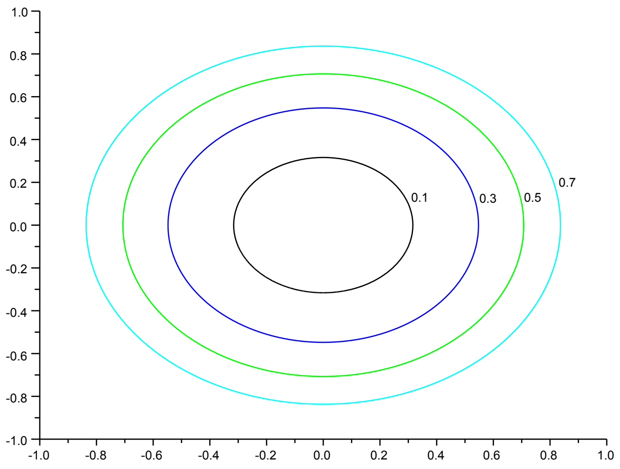


Figure 45: Contour plot of the function $f(x_1, x_2) = x_1^2 + x_2^2$ – The levels are explicitly configured.

loops to compute the `zdata` matrix, which contains the z values. The z values are computed for all the combinations of points $(x(i), y(j)) \in \mathbb{R}^2$, for $i = 1, 2, \ldots, n_x$ and $j = 1, 2, \ldots, n_y$, where n_x and n_y are the number of points in the x and y coordinates. In the end, we call the `contour` function, with the list of required levels (instead of the previous number of levels). This gets exactly the levels we want, instead of letting Scilab compute the levels automatically.

```
function f = myquadratic1arg ( x )
  f = x(1)**2 + x(2)**2;
endfunction
xdata = linspace ( -1 , 1 , 100 );
ydata = linspace ( -1 , 1 , 100 );
// Caution ! Two nested loops, this is bad.
for i = 1:length(xdata)
    for j = 1:length(ydata)
      x = [xdata(i) ydata(j)].';
      zdata ( i , j ) = myquadratic1arg ( x );
    end
end
contour ( xdata , ydata , zdata , [0.1 0.3 0.5 0.7])
```

The contour plot is presented in figure 45.

The previous script works perfectly. Still, it is not efficient because it uses two nested loops and this should be avoided in Scilab for performance reasons. Another issue is that we had to store the `zdata` matrix, which might consume a lot of memory space when the number of points is large. This method should be avoided since it is a bad use of the features provided by Scilab.

In the following script, we use the `feval` function, which evaluates a function on a grid of values and returns the computed data. The generated grid is made of all the combinations of points $(x(i), y(j)) \in \mathbb{R}^2$. We assume here that there is no possibility to modify the function myquadratic1arg, which takes one input argument. Therefore, we create an intermediate function myquadratic3, which takes 2 input arguments. Once done, we pass the myquadratic3 argument to the feval function and generate the zdata matrix.

```
function f = myquadratic1arg ( x )
  f = x(1)**2 + x(2)**2;
endfunction
function f = myquadratic3 ( x1 , x2 )
  f = myquadratic1arg ( [x1 x2] )
endfunction
xdata = linspace ( -1 , 1 , 100 );
ydata = linspace ( -1 , 1 , 100 );
zdata = feval ( xdata , ydata , myquadratic3 );
contour ( xdata , ydata , zdata , [0.1 0.3 0.5 0.7])
```

The previous script produces, of course, exactly the same plot as previously. This method should be avoided when possible, since it requires storing the zdata matrix, which has size 100×100.

Finally, there is a third way of creating the plot. In the following Scilab session, we use the same intermediate function myquadratic3 as previously, but we pass it directly to the contour function.

```
function f = myquadratic1arg ( x )
  f = x(1)**2 + x(2)**2;
endfunction
function f = myquadratic3 ( x1 , x2 )
  f = myquadratic1arg ( [x1 x2] )
endfunction
xdata = linspace ( -1 , 1 , 100 );
ydata = linspace ( -1 , 1 , 100 );
contour ( xdata , ydata , myquadratic3 , [0.1 0.3 0.5 0.7])
```

The previous script produces, of course, exactly the same plot as previously. The major advantage is that we did not produce the zdata matrix.

We have briefly outlined how to produce simple 2D plots. We are now interested in the configuration of the plot, so that the titles, axis and legends corresponds to our data.

7.4 Titles, axes and legends

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

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

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

We now have the plot which is presented in figure 43.

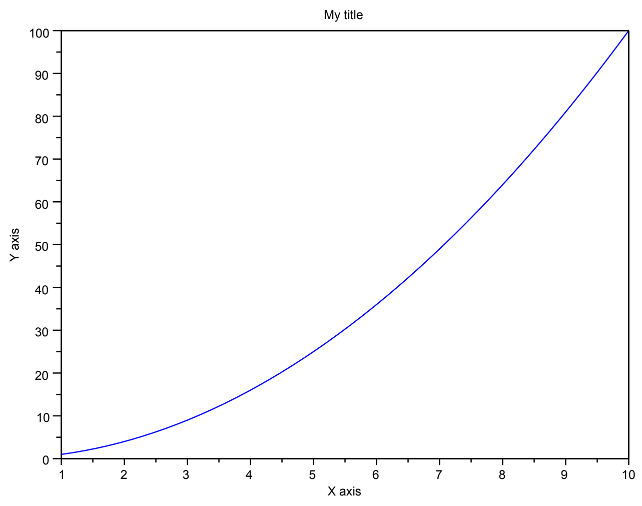


Figure 46: The x-y plot of a quadratic function – This is the same plot as in figure 43, with title and x-y axes configured.

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 axes, 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. These properties can be queried and configured simply by getting or setting their values, like any other Scilab variables. 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 axes of our plot as well. For this purpose, we use the `xtitle` function in the following script.

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

Figure 46 presents the produced plot.

It may happen that we want to compare two sets of data in the same 2D plot, that is, one set of x data and two sets of y data. 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 additionally use the ”`+-`” and ”`o-`” options of the `plot` function, so that we can distinguish the two curves $f(x) = x^2$ and $f(x) = 2x^2$.

```
function f = myquadratic ( x )
  f = x^2
endfunction
function f = myquadratic2 ( x )
```

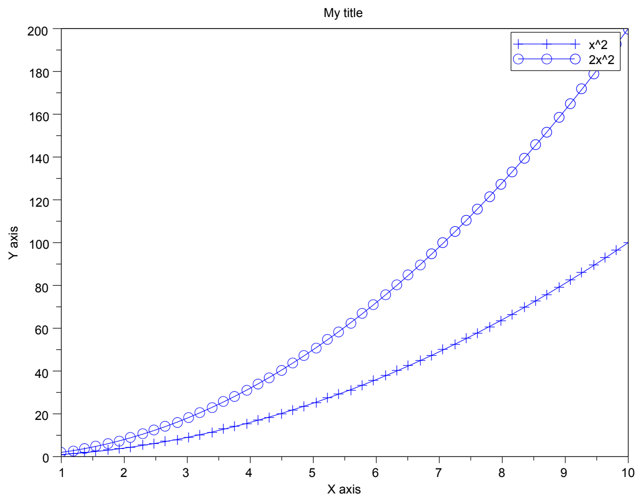


Figure 47: The x-y plot of two quadratic functions – We have configured the legend so that we can distinguish the two functions $f(x) = x^2$ and $f(x) = 2x^2$.

```
  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" );
```

Figure 47 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 putting it into a report. To do so, we can export the plot into a file, which is the subject of the next section.

7.5 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 48. Once a plot is produced, we can export its content into a file, by interactively using the *File > Export to...* menu of the graphics window. We can then set the name of the file and its type.

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

Figure 48: Export functions.

We can alternatively use the `xs2*` functions, presented in figure 48. All these functions are based on the same 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 preferred. 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 beginning of this document.

French readers may be interested by [5], where a good introduction is given about how to create and interface to an existing library, how to use Scilab to compute the solution of an Ordinary Differential Equation, how to use Scicos and many other subjects. The same content is presented in English in [3]. English readers should be interested by [4], which gives a deeper overview of Scicos. These books are of great interest, but are rather obsolete since they were written mainly for older version of Scilab.

Further reading may be obtained from the Scilab web cite [14], in the documentation section.

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 who helped me during the writing of this document. Many thanks to Sylvestre Ledru, who clarified many points on Scilab installation processes, and the ATOMS system. Thanks are also expressed to Artem Glebov and Jason Nicholson for proofreading this document. I thank Ihor Rokach for his comments on 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] C Bunks, J.-P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, and S. Steer. *Engineering and Scientific Computing With Scilab*. Birkhauser Boston, 1999.

[4] Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos*. Springer, 2006.

[5] J.-P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, and S. Steer. *Introduction à Scilab, Deuxième Édition*. Springer, 2007.

[6] Intel. Intel Math Kernel Library. http://software.intel.com/en-us/intel-mkl/.

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

[8] Sylvestre Ledru and Yung-Jang Lee. Localization. http://wiki.scilab.org/Localization.

[9] Sylvestre Ledru, Pierre Maréchal, and Clément David. Code conventions for the Scilab programming language. http://wiki.scilab.org/Code%20Conventions%20for%20the%20Scilab%20Programming%20Language.

[10] Sylvestre Ledru, Pierre Maréchal, and Simon Gareste. Atoms. http://wiki.scilab.org/ATOMS.

[11] The Scilab Consortium Michael Baudin. The scilab forge, 2011. http://wiki.scilab.org/Scilab%20forge.

[12] Cleve Moler. Numerical computing with Matlab.

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

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