An Introduction to

Scilab

from a Matlab User's Point of View

Version 5.2 [1]

Eike Rietsch

May 2, 2010

[1] FileScilab4Matlab 5.2.tex

To Antje

Preface

Scilab has progressed significantly since I wrote the previous version of this manual (Scilab, then, was in Version 2.7). The most important improvement, in my mind, is the new graphic system; then it just became available as an alternative, now it is the only one supported. It appears to provide all the features that I missed in the original version. Nevertheless, a description of Scilab's graphic capabilities is not included here; I continue to believe that they deserve a manual of their own.

Another improvement is the built-in editor. It works nicely, and it eliminates an annoyance: one can load a specific function (or all functions) in the editor into Scilab and it is automatically compiled. No more forgetting to compile a function after it has been changed.

Overall, Scilab functions are even closer to those of Matlab. For example, function "xgetfile" is now obsolete; its replacement, function "uigetfile" has the same name as the corresponding Matlab function. This also hints at a change in attitude. Initially, Scilab appeared to be geared towards Unix/X Windows with MS Windows just an afterthought. Now there are quite a few functions just for MS Windows. In addition, according to the Scilab team, Scilab runs "out of the box" on Mac OS X 10.5.5 (Leopard) and 10.6.x (Snow Leopard).

In this updated version of the manual I have removed outdated restrictions or caveats and included features that have been added to Scilab since Version 2.7. I am grateful to Sylvestre Ledru and Vincent Couvert, both members of the Scilab Team, for reading the manuscript and providing feedback that not only improved this manual but also my understanding of the new capabilities of Scilab.

May 2010

From the Introduction to Version 2.7 of this manual

This year the Scilab Group officially released Scilab-2.7. Its most important new feature—in my mind—is the new object-based graphics system. From what I have gleaned from the help files and the examples I consider it quite impressive. It reminds me of Matlab's handle graphics with some nifty features. But as yet there is no manual for it, and this could be a major impediment for its adoption. Furthermore, a number of features that I deem important are still missing. This manual, like the previous edition, does not discuss Scilab graphics either.

However, many other things have changed as well; these changes made some of the statements in the previous version obsolete. New functions, that I found useful, were added. They were already available in the CVS version of Scilab-2.7 and lead to continuous updates of the manual. As a result this new version of the manual is slightly longer and has a more extensive index. I have also fixed a number of typos that I found, and new ones are likely to have crept in.

September 2003

From the Introduction to Version 2.6 of this manual

Since 1993 I have been a heavy user of Matlab; this manual is the result of my effort to learn Scilab. Consequently, it is written from the point of view of someone who is familiar with Matlab and wants to use this knowledge to ease his entry into Scilab. Hence, this manual explains Scilab's functionality by drawing on the experience and expectations of a Matlab user. Thus, features that are the same in both systems are "glossed over" to some degree and more space is devoted to those features where the two differ; examples are operator overloading and lists of all types. Overall, this manual is not tailored to the needs of someone who is not already somewhat familiar with either Matlab or Scilab. Nevertheless, quite a number of chapters, which do not refer to Matlab analogs, would be useful for Scilab users without Matlab background.

To aid in the conversion of Matlab macros Table A.1 lists Matlab functions and their functional equivalents. Furthermore, the index includes an even more extensive lists of those Matlab functions and Scilab functions that are mentioned in the Manual. So one can look up quite a number of Matlab functions to find out what means there are in Scilab to achieve the same end. A user trying to figure out how to implement, say, a Matlab structure will be directed to Scilab lists. Someone who wants to understand the difference in the definition of workspace—which has the potential to trip up the unsuspecting—will need to look in the index which points to those pages that describe this difference.

Incidentally, one branch of the Scilab directory tree is a directory with Scilab functions that "emulate" Matlab functions. As explained more fully in Section 1.4 I do not advocate their use. Using such emulations deprives the user of the flexibility and power Scilab offers. In most cases it is a concept one needs to emulate not a function.

This manual is organized in a number of chapters, sections, and subsections. Obviously, this is arbitrary and reflects my own choices. Several sections have tables of functions or operators pertinent to the subject matter discussed. Due to some overlap one and the same function may show up in several different tables.

It was tempting to use unadulterated screen dumps as examples. However, Scilab wastes screen real estate the same way *`format loose`* does in Matlab — except, in Scilab, there is no equivalent to *`format compact`*, which suppresses the extra line-feeds. Hence, to conserve space, most examples are reproduced without some of these extra empty lines.

In compiling this manual I used Scilab 2.6 and the standard Scilab documentation:
Introduction To Scilab - Users Guide by the Scilab Group
Une Introduction à Scilab by Bruno Pinçon

Scilab Bag of Tricks by Lydia E. van Dijk and Christoph L. Spiel
All three can be downloaded from the INRIA web site (http://www.scilab.org), which also has manuals in languages other than English and French. I also drew freely on newsgroup discussions (comp.soft-sys.math.scilab), in particular contributions by Bruno Pinçon, Alexander Vigodner, Enrico Segre, Lydia van Dijk, and Helmut Jarausch.

From newsgroup discussions I got the impression that most users run Scilab on Unix (particularly Linux) machines. I, on the other hand, use Matlab and Scilab on Windows PCs. I do have a Scilab installation on a Sun workstation running Solaris, but use it only occasionally for quick calculations in a Unix environment. While I do not expect significant differences in the use of Scilab on different platforms, my pattern of use does color this manual. However, I am not completely Windows-centric: affected by many years of Unix use, I tend to favor the Unix term "directory" over the PC term "folder".

Every now and then this manual contains judgements regarding the relative merits of features in Matlab and Scilab. They represent my personal biases, experiences, and — presumably — a lack of knowledge as well.

Obviously, I cannot claim to cover every Matlab function or Scilab function. The selection is largely based on the subset of functions and syntactical features that I have been using over the years. But among all the omissions one is glaring. I do not discuss plotting. Were I unaware of Matlab, I would consider Scilab's plotting facility superb. But now I am spoiled. However, I understand that a new object-oriented plot library is under development, and I am looking forward to its release. Furthermore, plotting is such an important and extensive subject that it deserves a manual of its own (as is the case for Matlab).

Finally, the typographic conventions used are:
`Red typewriter font is used for Scilab commands, functions, variables, ...`
`Blue slanted typewriter font is used for Matlab commands, functions, variables, ...`
`Black typewriter font is used for general operating system-related terms and filenames outside of code fragments.`
Keyboard keys, such as the Enter key, are written with the name enclosed in angle brackets: <ENTER>. In the section on operator overloading angle brackets are also used to enclose operand types and operator codes.

Acknowledgment

Special thanks go to Glenn Fulford who was kind enough to review this manuscript and offer suggestions and critique and, in particular, to Serge Steer who not only provided a list of corrections but also an extensive compilation of the differences between Scilab and Matlab; I used it for my own education and included what I learned.

May 2002

Contents

1 Preliminaries **1**

1.1 Customizing Scilab for Windows . . . 1

1.1.1 Start-up File . . . 1

1.1.2 Fonts . . . 1

1.1.3 Number Formats . . . 1

1.1.4 Paging . . . 2

1.2 Interruption/Termination of Scripts and Scilab Session . . . 2

1.3 Help . . . 3

1.4 Emulated Matlab functions . . . 3

2 Syntax **4**

2.1 Arithmetic Statements . . . 4

2.2 Built-in Constants . . . 6

2.3 Predefined Global and Environmental Variables . . . 7

2.4 Comparison Operators . . . 8

2.5 Flow Control . . . 9

2.6 General Functions . . . 12

3 Variable Types **16**

3.1 Numeric Variables — Scalars and Matrices . . . 17

3.2 Special Matrices . . . 21

3.3 Character String Variables . . . 22

3.3.1 Creation and Manipulation of Character Strings . . . 22

3.3.2 Strings with Scilab Expressions . . . 30

3.3.3 Symbolic String Manipulation . . . 32

3.4 Boolean Variables . . . 34

3.5 Cell arrays . . . 39

3.6 Structures . . . 42

3.7 Lists . . . 46

3.7.1 Ordinary lists (list) . . . 46

3.7.2 Typed lists (tlist) . . . 50

3.7.3 Matrix-oriented typed lists (mlist) . . . 55

3.8 Polynomials . . . 56

4 Functions **60**

4.1 General . . . 60

4.2 Functions that Operate on Scalars and Matrices . . . 61

4.2.1 Basic Functions . . . 61

4.2.2 Elementary Mathematical Functions . . . 65

4.2.3 Special Functions . . . 67

4.2.4 Linear Algebra . . . 68

4.2.5 Signal-Processing Functions . . . 69

4.3 File Input and Output . . . 71

4.3.1 Opening and Closing of Files . . . 71

4.3.2 Functions `mgetl` and `mputl` . . . 72

4.3.3 Functions `read` and `write` . . . 75

4.3.4 Functions `load` and `save` . . . 78

4.3.5 Functions `loadmatfile` and `savematfile` . . . 79

4.3.6 Functions `mput` and `mget/mgeti` . . . 79

4.3.7 Functions `input` and `disp` . . . 79

4.3.8 Function `uigetfile` . . . 80

4.4 Utility Functions . . . 80

5 Scripts **84**

6 User Functions **90**

6.1 Functions in Files . . . 95

6.2 In-line Functions . . . 97

6.3 Functions for operator overloading . . . 98

6.4 Profiling of functions . . . 103

6.5 Translation of Matlab m-files to Scilab Format . . . 104

7 Function Libraries and the Start-up File **106**

7.1 Creating function libraries . . . 106

7.2 Start-up file . . . 108

7.3 User-supplied Libraries . . . 109

8 Error Messages and Gotchas **111**

8.1 Scilab error messages . . . 111

8.1.1 !–error 4: undefined variable: . . . 111

8.1.2 !–error 66: Too many files opened! . . . 112

8.2 Gotchas . . . 112

A Matlab functions and their Scilab Equivalents **114**

Index **121**

List of Tables

?? List of arithmetic operators 5
2.2 Built-in constants 7
2.3 Global and environmental variables 7
?? Comparison operators 8
?? Flow control 9
2.6 General functions 15

3.1 Variable types 17
3.2 Utility functions for managing/classifying of variables 17
3.3 Integer conversion functions 19
3.4 Special matrices 22
3.5 Functions that manipulate strings 24
3.6 Functions that evaluate strings with Scilab expressions 30
?? Comparison of the use of boolean variables as array indices in Scilab and Matlab . . 34
3.8 Boolean operators 35
3.9 Boolean variables and functions that operate on, or output, boolean variables 37
3.10 Functions that create, or operate on, cell arrays 40
3.11 Functions that create, or operate on, structures 45
3.12 Functions that create, or operate on, lists 47
3.13 Functions related to polynomials and rational functions 57

4.1 Basic arithmetic functions 63
4.2 Elementary transcendental functions 66
4.3 Matrix functions 67
4.4 Special functions 67
4.5 Linear algebra 68
4.6 Functions for sparse matrices 69
4.7 Functions for signal processing 69
4.8 Functions that manipulate file names and open, query, and close files 72
4.9 Functions that input data from files or the keyboard 73
4.10 Functions that output data to files or to the Scilab window 74
4.11 Utility functions 81

6.1 Functions/commands/keywords relevant for user functions 91
6.2 Operator codes used to construct function names for operator overloading 100

7.1 Functions installing, managing and removing user-supplied Scilab modules 110

A.1 Matlab functions and their Scilab equivalents 114

Chapter 1

Preliminaries

1.1 Customizing Scilab for Windows

1.1.1 Start-up File

Commands that should be executed at the beginning of a Scilab session can be put into the start-up file `.scilab` (the dot "." as the first character of the file name betrays the Unix heritage of Scilab). An alternative name for the start-up file is `scilab.ini` which might sound more familiar to those running Scilab on a Windows PC.
On a PC running Windows this start-up file must be in the Scilab home directory (type `SCIHOME` in the Scilab Console window to find the Scilab home directory).
Scilab's own start-up file, `scilab.start`, is in the `etc` subdirectory of the Scilab directory (type `SCI` or `getenv('SCI')` in the Scilab Console window to find this directory). Function `scilab.start` actually calls `.scilab` first and then tries to run `scilab.ini`.

1.1.2 Fonts

Screen fonts can be set in two different ways: Either click on the `Preference` menu button and then on `Font...` on the drop-down menu. Alternatively, click the font icon, **A**, on the icon bar. Both actions produce a window that allows selection of font and font size.

1.1.3 Number Formats

The way numbers are printed to the screen (e.g. number of digits) can be set with the command `format`. It has one or two arguments. The first argument, if given, is a string and describes the variable type; the second argument is the total number of positions used for digits, decimal point, sign, etc.

```
-->x=rand(1,3)
 x  =
    0.2113249     0.7560439     0.0002211    [1]

-->format(20);
--> x
```

```
 x  =
    0.21132486546412110     0.7560438541695476     0.00022113462910056

-->format('e',10);

--> x
 x  =
   2.113E-01    7.560E-01    2.211E-04

-->format('v',10);

--> x
 x  =
    0.2113249     0.7560439     0.0002211     1a
```

Comparison of 1 and 1a shows that the default is `format('v',10)`.

1.1.4 Paging

By default, display of a long array or vector is halted after a number of lines have been printed to the screen, and the message `[More (y or n ) ?]` is displayed. The number of lines displayed can be controlled via the `lines` command. Paging (and that message) can be suppressed by means of the command `lines(0)`. If this appears desirable, this command can be put in the start-up file `.scilab` to be run at start-up (see Section 7.2).

1.2 Interruption/Termination of Scripts and Scilab Session

Scilab has a feature that is sorely missed in Matlab: a reliable facility to interrupt or terminate a running program. The command `abort` allows one to terminate execution of a function or script, e. g. in debugging mode after a `pause` has been executed and continuation of the execution is not desired. In Matlab the usual way to achieve this goal is to clear all variables and thus to force a fatal error with the ***`return`*** command — and even this does not work every time. The `abort` command can also be invoked from the Control menu and does what it says: it aborts any running program. A less drastic intervention is `Interrupt`, also available from the Control menu. It interrupts a running program to allow keyboard entry (note that a program interruption in Scilab creates a new workspace; what this means is explained on page 12). Execution of the program can be resumed by typing `resume` or `return`. The same objective can be achieved by means of the `Resume` menu item in the Control menu or its keyboard shortcut <Alt c> followed by <Alt e> (press down the <Alt> key and hit <c> and then <e>).

The commands `quit` and `exit` can be used to terminate a Scilab session. Both commands exist in Matlab as well, and `exit` behaves like its Matlab counterpart. The `quit` command is somewhat different. If called from a higher workspace level it reduces the level by 1 (see the discussion of `pause` on page 12). If called from level 0 it terminates Scilab. In this case, `quit` also executes a termination script, `scilab.quit`, located in the `etc` subdirectory of the Scilab root directory (type

SCI or getenv('SCI') in the Scilab Console to find this directory). This script can be modified by the user and is comparable to *finish.m* in Matlab. Of course, one can also terminate Scilab by clicking on Exit in the File menu or the close box in the right upper corner of the Scilab window.

1.3 Help

The command-line help facility is similar to Matlab's. Typing help sin, for example, brings up a separate help window with information about sin. Typing help symbols brings up a table of symbols and the corresponding alphabetic equivalent to be used in the help command. For example, to find out what .* does type help star. Unfortunately, one still has to type in a misspelled word, "tilda", for "tilde"; help tilde gets the help file for gtild).
The command apropos, somewhat equivalent to Matlab's *lookfor*, performs a string search and lists the result in a separate window. Clicking a command in this list brings up the help window for that command. A search argument consisting of more than one word must be enclosed in single or double quotes (e.g. apropos "matrix pencil")
The Help Menu on the menu bar is pretty self-explanatory. It provides a nice overview of the commands available in more than 50 categories and is a very convenient way to get started with Scilab.

1.4 Emulated Matlab functions

The Scilab distribution comes with a directory,
SCIDIR\modules\compatibility_functions\macros,
where SCIDIR denotes the Scilab root directory (in Windows something like C:\Program Files\Scilab). In this directory there are more than 100 function that emulate Matlab functions; they appear to be mainly intended for automatic translations of Matlab functions to Scilab. For several reasons I do not advocate their use since this kind of "translation" of a Matlab object to Scilab may prevent a user from fully exploiting powerful features Scilab offers. An example are cell arrays. If all cell entries are strings then a string matrix is the appropriate "translation" to Scilab. Converting them into Scilab cell arrays instead deprives one of the benefits Scilab's string matrices offer (such as the overloaded + operator and the functionality of length). In other situations an ordinary list or a list of lists may be more appropriate.

Chapter 2

Syntax

2.1 Arithmetic Statements

Scilab syntax is generally quite like Matlab syntax. This means that someone familiar with Matlab knows how to write basic Scilab commands such as

```
// These are simple examples
-->a = 3; b = 7.2;

-->c = a + b^2 - sin(3.1415926/2)
  c  =
    53.84
```

As shown in these examples the Scilab prompt is `-->`, and any statements following it represent user input.

Comments are indicated by two slashes (`//`): everything to the right of the slashes is ignored by the interpreter/compiler. There are no means to define a block of comments similar to Matlab's "`%{`" "`%}`" syntax. However, the built-in editor editor has a "`Comment selection`" and "`Uncomment selection`" option in the `Edit` drop-down menu.

Like in Matlab, several statements can be on one line as long as they are separated by commas or semicolons. Semicolons suppress the display of results, commas do not.

Names of Scilab variables and functions must begin with a letter or one of the following special characters `%`, `#`, `!`, `$`, `?`, and the underscore `_`. Subsequent characters may be alphanumeric or the special characters `#`, `!`, `$`, `?`, and `_`. Thus `%` is only allowed as the first character of a variable name. Variables starting with `%` generally represent built-in constants or functions that overload operators. Variable names may be of arbitrary length, but all except the first 24 characters are disregarded (Matlab, since version 6.5, uses the first 63 characters).

```
-->a1234567890123456789012345678901234567890 = 34
a12345678901234567890123   =
    34.
```

Variable names are case-sensitive (i. e. Scilab distinguishes between upper-case and lower-case letters). A semicolon (`;`) terminating a statement indicates that the result should not be displayed whereas a comma or <ENTER> prompts a display of the result.

To create expressions, Scilab uses the same basic arithmetic operators Matlab does, but with two options for exponentiation.

+	Addition
−	Subtraction
*	Matrix multiplication
.*	Array multiplication
.*.	Kronecker multiplication
/	Division
\	Left matrix division
./	Array division
.\	Left array division
./.	Kronecker division
.\.	Kronecker left division
^ or **	Matrix exponentiation
.^	Array exponentiation
'	Matrix complex transposition
.'	Array transposition

Table 2.1: List of arithmetic operators

Statements can continue over several lines. Similar to Matlab's syntax, continuation of a statement is indicated by two or more dots, `..` (Matlab requires at least three dots).
Numbers can be used with and without decimal point. Thus the numbers 1, 1., and 1.0 are equivalent. However, in both Scilab and Matlab, the decimal points does double duty. In conjunction with the operators *, /, and ^ it indicates that operations on vectors and matrices are to performed on an element-by-element basis. This leads to ambiguities that can cause problems for the unsuspecting.

```
-->x = 1/[1 2 3]          2a
 x  =
    0.0714286
    0.1428571
    0.2142857

-->x = 1./[1 2 3]         2b
 x  =
    0.0714286
    0.1428571
    0.2142857

-->x = 1 ./[1 2 3]        2c
 x  =
   1.      0.5      0.3333333
```

Statements $\boxed{\text{2b}}$ and $\boxed{\text{2c}}$ look very similar and yet they produce quite different results. The reason for the difference is the space between the zero and the dot in $\boxed{\text{2c}}$ where the dot is interpreted as part of the operator `./` whereas in $\boxed{\text{2b}}$ it is interpreted as the decimal point of the number 1. In Matlab, on the other hand, statements $\boxed{\text{2b}}$ and $\boxed{\text{2c}}$ produce the same result (the one produced in Scilab by $\boxed{\text{2c}}$), and $\boxed{\text{2a}}$ causes an error message. In Scilab, however, `a` is a solution of `a'*[1 2 3]' = 1`. More generally, if

$$x^T A^T = b^T \tag{2.1}$$

where the superscripted T denotes transposition, then `x = b'/A'` computes the unknown x^T. Since Eq. 2.1 is equivalent to

$$Ax = b \tag{2.2}$$

x can also be computed from `x = A\b`. Hence, `(b'/A')'` is equivalent to `A\b`. The latter is also Matlab syntax. Thus

```
-->x = [1 2 3]'\ 1
 x  =
    0.0714286      0.1428571      0.2142857
```

In addition to single-variable assignments, Scilab has tuple assignments in which multiple variables are assigned values in a single statement. An example is

```
--> [u,v,w] = (1,2,3)
 w  =
    3.
 v  =
    2.
 u  =
    1.
```

Note that the commas on the right-hand and the left-hand side are required; they cannot be replaced by blanks). This construct bears some similarity with Matlab's *`deal`* function, but it is less powerful. For example, the number of Scilab objects on the right-hand side must equal that on the left hand side. Even if one wanted to assign the same value to all three variables one would still have to write it out three times; thus `[u,v,w] = (1)` is not allowed.
A feature peculiar to Scilab is the order (from right to left) in which variables in a left-hand bracketed expression are displayed; as shown in the example above the rightmost variable, `w`, is displayed first, followed by `u` and, finally, `v`.
A handy use of the tuple assignment is swapping of values. With variables `u` and `v` defined above

```
-->[u,v] = (v,u)
 v  =
    1.
 u  =
    2.
```

Scilab	Matlab	Description
%i	*i, j*	Imaginary unit ($\sqrt{-1}$)
%e	*e*	Euler's constant ($e = 2.7182818\cdots$)
%pi	*pi*	Ratio of circumference to diameter of a circle; ($\pi = 3.14159\cdots$)
%eps	*eps*	Machine ϵ ($\approx 2.2 \cdot 10^{-16}$); smallest number such that $1 + \epsilon > 1$
%inf	*inf*	Infinity (∞)
%nan	*NaN*	Not a number
%s		Polynomial s=poly(0,'s')
%z		Polynomial z=poly(0,'z')
%t	*true*	Boolean variable: logical true
%f	*false*	Boolean variable: logical false
%io		Two-element vector with file identifiers for standard I/O

Table 2.2: Built-in constants

2.2 Built-in Constants

Table 2.2 lists Scilab's special, built-in constants together with their Matlab equivalents (where they exist). Unlike constants in Matlab they are protected and cannot be overwritten. This has benefits; in Matlab a variable such as *i* can be overwritten inadvertently if it is redefined, for example, by its use as an index.
In many respects, keyboard (standard input) and Scilab window (standard output) are treated like files, and %io(1) (usually 5) is the file identifier for the keyboard and %io(2) (usually 6) is the file identifier for the Scilab window.

2.3 Predefined Global and Environmental Variables

Scilab	Matlab	Description
HOME		User's home directory (environmental variable)
home		User's home directory (global variable)
OS	*OS*	Operating system (environmental variable)
PWD	*pwd*	Working directory (global variable)
SCI	*matlabroot*	Scilab root directory (global/environmental variable)
SCIHOME	*prefdir*	Directory containing preferences (global/environmental variable)
TMPDIR	*tempdir*	Name of directory for temporary files (global variable)

Table 2.3: Global and environmental variables

The previously available environmental variable LANGUAGE is now obsolete; its replacement are functions getlanguage and setlanguage, respectively. These two functions can be used to identify (or set) the language used for menu buttons, Scilab help, etc. While setlanguage can take any variable, the only presently supported values are 'en_US' and 'fr_FR' for English and French, respectively. Obviously, this function can be used to create language-specific user interfaces, help files, etc.

The global variable MSDOS has been deprecated and will be removed in Scilab Version 5.3. Its functionality is provided by function getos which outputs a string with the name of the operating system.

```
 -->getos()
 ans =
Windows
```

Global variables can be accessed like actual variables. Thus

```
-->PWD
PWD  =
 C:\Users\user\Desktop
```

Environmental variables, on the other hand, need to be accessed via getenv and set via setenv. For example,

```
-->getenv('SCI')
ans  =
 D:/Program Files/Scilab-5.1.1
```

Note the quotes; the argument of getenv (and of setenv) must be a string. Also note that slashes (/) are used — rather than backslashes (\) to separate directory and file name in spite of the fact that the function was run on a PC.

2.4 Comparison Operators

Scilab uses the same comparison operators Matlab does, but with two choices for the "not equal" operator.

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

Table 2.4: Comparison operators

The result of a valid expression involving any of these operators — such as `a > 0` — is a boolean variable (%t or %f) or a matrix of boolean variables. These boolean variables are discussed later in Section 3.4.
In Scilab the first four operators are only defined for real numbers; in Matlab complex numbers are allowed but only the real part is used for the comparison.
The last two operators compare objects. Examples are

```
-->[1 2 3] == 1          3a
ans  =
```

```
  T F F

-->[1 2 3] == [1 2]      3b
 ans  =
  F

-->[1 2] == ['1','2']    3c
 ans  =
  F
```

In Matlab 3a would produce the same result, 3b abort with an error message, and 3c create the boolean vector [0 0].

2.5 Flow Control

Scilab's flow control syntax mirrors that used by Matlab.

Scilab	*Matlab*	
break	*break*	Force exit from a loop
case	*case*	Start clause within a select block
catch	*catch*	Start of the error-catching code in a try / catchidstry block
elseif	*elseif*	Start a conditional alternative in an if block
else	*else*	Start the alternative in an if block
else	*otherwise*	Start the alternative in an select/*switch* block
end	*end*	Terminate for, if, select, while, and try/catch blocks
errcatch	*try/catch*	Traps error with several possible actions
for	*for*	Start a loop with a generally known number of repetitions
if	*if*	Start a conditionally executed block of statements
select	*switch*	Start a multi-branch block of statements
try	*try*	Start of a try / catch block
while	*while*	Start repeated conditional execution of a block

Table 2.5: Flow control

However, there is more than the semantic difference between keywords *switch* and *otherwise* in Matlab and select and else, respectively, in Scilab. The following comparison illustrates this difference. With function foobar defined as:

```
function foobar(a)
//   Scilab
 select a
 case ['foo','pipo']
   disp('ok')
 case 'foo'
   disp('not ok')
```

```
 else
   disp('invalid case')
 end
endfunction
```

one gets

```
-->foobar(['foo','pipo'])
 ok

-->foobar('foo')
 not ok

-->foobar('pipo')
invalid case
```

The variable `a` following the keyword `select` can be any Scilab data object.
The analogous Matlab function

```
function foobar(a)
%   Matlab
switch a
case {'foo','pipo'}
  disp('ok')
case 'foo'
  disp('not ok')
otherwise
  disp('invalid case')
end
```

on the other hand, leads to

```
>>foobar({'foo','pipo'})
 ??? SWITCH expression must result in a scalar or string constant.

>>foobar('foo')
 ok

>>foobar('pipo')
 ok
```

The variable `a` following the keyword `switch` can only be a scalar or string constant. On the other hand, a `case` can represent more than one value of the variable. The strings `'foo'` and `'pipo'` satisfy the first case and so the second case is never reached.
In an `if` clause Scilab has the optional keywords `then` and `do` as in

```
-->if a >= 0 then a=sqrt(a); end

-->if a >= 0 do a=sqrt(a); end
```

but `then` and `do` can be replaced by a comma, a semicolon, or pressing the <ENTER> key. Hence, both statements are equivalent to

```
-->if a >= 0, a=sqrt(a); end
```

Likewise, the `for` loop can be written with the optional keyword `do` as in

```
for i = 1:n do a(i)=asin(2*%pi*i); end
```

and again `do` can be replaced by a comma, a semicolon, or pressing the <ENTER> key. The same is true for the `while` clause.
Matlab uses the *`try`*/*`catch`* syntax to trap errors. Its functionality is now also available in Scilab. For the sake of clarity it is shown here the way it would look in a file.:

```
try      4
   a=1/0;
   b=0;
catch
   disp('New definition of ""a"":')     5
   a=1.0e10;
end
disp(a)
disp(b)
```

The result looks like this:

```
New definition of "a":
   1.000D+10
disp(b)
     !--error 4
Undefined variable:  b
at line       8 of exec file called by :
exec("C:/Users/user/AppData/Local/Temp/SCI_TMP_4264_/Untitled1.sce");
while executing a callback
```

Statement 4 starts trapping errors. Any error found between the `try` and the `catch` statements is caught and control is transferred to statement 5, the one following the `catch` statement.
Probably of more historical interest is the approach to error trapping by means the combination of `errcatch` and `iserror`. This is illustrated in the following code fragment.

```
errcatch(-1,'continue','nomessage'); // Start error trapping   6
a=1/0                                                          7a
if iserror()      // Check for error
  disp('A division by zero has occurred')
  errclear(-1)
end

a=1/0                                                          7b
```

```
b=1
errclear(-1)
errcatch(-1)      // Error trapping toggled off
a=1/0                                                             7c
```

Statement 6 starts error trapping with the system error message suppressed. Statements 7a, 7b, and 7c represent errors. Execution of these statements produces the following result:

```
-->errcatch(-1,'continue','nomessage'); // Start error trapping   6

-->a=1/0                                                            7a

-->if iserror()      // Check for error
-->  disp('A division by zero has occurred')

 A division by zero has occurred
-->  errclear(-1)
-->end

-->a=1/0                                                            7b

-->b=1
 b  =
    1.

-->errclear(-1)

-->errcatch(-1)      // Error trapping toggled off

-->a=1/0                                                            7c
      !--error    27
division by zero...
```

Clearly, the three identical "division by zero" errors are treated differently. The first one, 7a, is trapped and the user-supplied message is printed; the second, 7b, is trapped and ignored; the third division by zero, 7c, occurs after error trapping has been turned off and creates the standard system error message.

Other commands can be considered as at least related to flow control. They include `pause` which interrupts execution similar to Scilab's ***keyboard*** command, but with some important differences explained in Section 2.6 beginning on page 12.

Another function, `halt`, can be used to temporarily interrupt a program or script. Execution of a function or script will stop upon encountering `halt()` and wait for a key press before continuing.

2.6 General Functions

Table 2.6 lists Scilab's low-level functions and commands (commands are actually functions used with command-style syntax; see Section 4.1). Some, like `date`, are essentially the same as in Matlab, others have slightly different names (`exists` vs. *`exist`*), some may have the same name but may give slightly different output (in Scilab `length` with a numeric-matrix argument returns the product of the number of rows and columns, in Matlab *`length`* returns the larger of the number of rows and columns), and many are quite different.
In this list of functions the command `pause` deserves special attention. It is equivalent to Matlab's *`keyboard`* command in that it interrupts the flow of a script or function and returns control to the keyboard. However, a Matlab function/script stays in the workspace of the function. In Scilab the `pause` command creates a new workspace. The prompt changes from, say, `-->` to `-1->` where the number 1 indicates the workspace level. All variables of all lower workspace are available at this new workspace as long as they are not shadowed (a variable in a lower workspace is shadowed if a variable with the same name is defined in a higher workspace). This is an example:

```
-->a = 1, b = 2    // Variables in original workspace
 a  =
    1.
 b  =
    2.

-->pause  // Creates new workspace (level 1)

-1->disp([a,b])
   1.    2.

-1->c = a+b
 c  =
    3.

-1->a = 0
 a  =
    0.

-1->return  // Return to original workspace 8a

-->a, c
 a  =
    1.
  !--error     4
undefined variable : c
```

The command `pause` creates a new workspace (the level of this workspace becomes part of the prompt); the display function `disp` shows that the variables `a` and `b` are available in this new

workspace, and the new variable `c` is computed correctly. However, upon returning to the original workspace we find that `a` still has the value 1 (in spite of being changed in the level-1 workspace) and that the variable `c` is no longer available. This is not what one would have found with Matlab's *`keyboard`* command.
In order to get these new values to the original workspace they have to be returned by the `return` command. In Scilab the `return` command can have input and output arguments!

```
-->a = 1, b = 2
a =
1.
b =
2.

-->pause  // Create new workspace (level 1)

-1->disp([a,b])
  1.  2.

-1->c = a+b
c =
3.

-1->a = 0
a =
0.

-1->[aa,c] = return(a,c)  // Return to original workspace 8b

-->aa,c
aa =
0.
c =
3.
```

The above two code fragments are identical except for the `return` statements $\boxed{8a}$ and $\boxed{8b}$. Statement $\boxed{8b}$ returns variables `a` and `c` created in the level-1 workspace, renaming `a` to `aa`. Without this name change, the existing variable `a` would have been overwritten by the value of `a` created in the level-1 workspace. A more complicated use of the `return` function is illustrated in statement $\boxed{35}$ on page 88. It is used there to return variables from a function. The number of variables and their names are generally not known at the time the function is called.
The command `resume` is equivalent to the `return` command (one could have used `resume` instead of `return` in the two examples above). Like `return` it can also be used to return variables to the level-1 workspace.

Scilab	Description
`$`	Index of the last element of matrix or (row/column) vector
`abort`	Interrupts current evaluation and return to prompt
`apropos`	Keyword search for a function
`clear`	Clear unprotected variables and functions from memory
`clearglobal`	Clear global variables from memory
`date`	Current date as string
`disp`	Display input argument
`getdate`	Get date and time in an 8-element numeric vector
`global`	Define variables as global
`halt`	Stop execution and wait for a key press
`help`	On-line help
`inttype(a)`	Output numeric code representing type of integer `a`
`pause`	Interrupt execution of function or script
`resume`	Return from a function or resume execution after a `pause`
`return`	Return from a function or resume execution after a `pause`
`tic`	Start a stopwatch timer
`timer`	Outputs time elapsed since the preceding call to `timer()`
`toc`	Read the stopwatch timer initiated via `tic`
`type(a)`	Output numeric code representing type of variable `a`
`typeof(a)`	Output string with type of variable `a`
`whereis`	Display name of library containing a specific function
`who`	Displays/outputs names of current variables
`who_user`	Displays/outputs names of current user-defined variables
`whos`	Displays/outputs names and specifics of current variables

Table 2.6: General functions

Unlike its Matlab counterpart, the display function `disp`, which has already been used above, allows more than one input parameter:

```
-->disp(123,'The result is:')
 The result is:
    123.
```

It shows the same behavior noted previously: the input arguments are printed to the screen beginning with the last.

The pair `tic` and `toc`, familiar from Matlab, perform similarly to `timer()`. They measure the wall-clock time required for the execution of one or more statements.

```
-->clear a;
-->tic; a=rand(1000,1000); toc
 ans =
     0.109

-->clear a;
-->timer(); a=rand(1000,1000); timer()
 ans =
     0.1092007
```

Note that function `timer` displays the CPU time, using more digits than the `tic`/`toc` pair. The latter, like in Matlab, measure the elapsed time (wall-clock time).

Chapter 3

Variable Types

The only two variable types a casual user is likely to define and use are numeric variables and strings; but Scilab has many more data types — in fact, it has more than Matlab. Hence, it is important to be able to identify them. Unlike Matlab, which uses specific functions with boolean output for each variable type (e. g. *iscell*, *ischar*, *isnumeric*, *issparse*, *isstruct*), Scilab uses essentially two functions, `type` and `typeof`: the former has numeric output, the latter outputs a character string. Table 3.1 lists variable types and the output of functions `type` and `typeof`. The last column of this table, with heading "Op-type", defines the abbreviation used to specify the operand type for operator overloading (see Section 6.3).
In addition, there is a special function `inttype` (see Table 3.2) to identify variables of type integer (see Table 3.3). Also, for variables of type integer, the function `typeof` outputs a character string identical to the name of the function that creates it (see Table 3.3). Thus

```
-->i8=uint8(16)      // i8 is an unsigned 8-bit integer
 i8  =
  16

-->typeof(i8)
 ans  =
uint8
```

The output of `typeof` for typed lists (`tlist`) and matrix-oriented typed lists (`mlist`) is discussed in Section 3.7.
Function `typeof` affords a straightforward simulation of Matlab function *isa*:

```
-->i8=uint8(11);
-->typeof(i8) == 'uint8'
 ans =
  T
```

and

```
>> i8=uint8(11);
>> isa(i8,'uint8')
ans =
     1
```

are equivalent.

Type of variable	`type`	`typeof`	Op-type
real or complex constant matrix	1	'constant'	s
polynomial matrix	2	'polynomial'	p
boolean matrix	4	'boolean'	b
sparse matrix	5	'sparse'	sp
sparse boolean matrix	6	'boolean sparse'	spb
Matlab sparse matrix	7	?	msp
matrix of integers stored in 1, 2, or 4 bytes	8	Depends on type of integer	i
handle of a graphic entity	9	'handle'	h
matrix of character strings	10	'string'	c
function (un-compiled)	11	'function'	m
function (compiled)	13	'function'	mc
function library	14	'library'	f
list	15	'list'	l
typed list (`tlist`)	16	Depends on type of list	tl
matrix-oriented typed list (`mlist`)	17	Depends on type of list	ml, ce, st
pointer	128	?	ptr
index vector with implicit size	129	'size implicit'	ip
intrinsic function, primitive	130	'fptr'	fptr

Table 3.1: Variable types

`Scilab`	Description
`inttype(a)`	Output numeric code representing type of integer `a`
`type(a)`	Output numeric code representing type of variable `a`
`typename`	Associate a variable type name with a numeric avariable type
`typeof(a)`	Output string with type of variable `a`
`who`	Displays/outputs names of current variables
`who_user`	Displays/outputs names of current user-defined variables
`whos`	Displays/outputs names and specifics of current variables

Table 3.2: Utility functions for managing/classifying of variables

3.1 Numeric Variables — Scalars and Matrices

Scilab knows matrices. This term includes scalars and vectors. Scalars and the elements of vectors and matrices can be real or complex. The statements

```
-->a = 1.2;
-->b = 1.0e3;
-->cx = a+%i*b
  cx   =
      1.2 + 1000.i
-->cy = complex(a,b)
  cy   =
      1.2 + 1000.i
```

define four 1×1 matrices, i.e. scalars. Complex numbers, such as `cx` and `cy` above, can be defined via an arithmetic statement or by means of function `complex`.
Vectors and matrices can be entered and accessed in much the same way as in Matlab.

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

-->mat2=[mat;mat+6]
 mat2  =
   1.     2.     3.
   4.     5.     6.
   7.     8.     9.
   10.    11.    12.

-->mat(2,3)
 ans  =
    6.

-->mat(2,:)
 ans  =
   4.    5.    6.

-->mat($,$)
 ans  =
    6.

-->mat($)
 ans  =     6.
```

There is a difference in the way the last element of a vector or matrix is accessed. Scilab uses the `$` sign as indicator of the last element whereas Matlab uses *`end`*. The `$` represents, in fact, a somewhat more powerful concept and can be used to create an implied-size vector, a new variable of type `'size implicit'`.

```
-->index=2:$
```

```
 index  =
 2:1:$

-->type(index)
 ans  =
    129.

-->typeof(index)
 ans  =
 size implicit

-->mat2(2,index)
 ans  =
   5.    6.
```

There is no equivalent in Matlab for this kind of addressing of matrix elements.
By default, Scilab variables are double-precision floating point numbers. There are no single-precision floating-point numbers. However, like Matlab, Scilab knows integers Conversion functions are shown in Table 3.3. Function `iconvert`, which takes two input arguments, does essentially what the seven other conversion functions listed in this table do.

Scilab	Description
`double`	Convert integer array of any type/length to floating point array
`iconvert`	Convert numeric array to any integer or floating point format
`int8(a)`	Convert `a` to an 8-bit signed integer
`int16(a)`	Convert `a` to a 16-bit signed integer
`int32(a)`	Convert `a` to a 32-bit signed integer
`uint8(a)`	Convert `a` to an 8-bit unsigned integer
`uint16(a)`	Convert `a` to a 16-bit unsigned integer
`uint32(a)`	Convert `a` to a 32-bit unsigned integer

Table 3.3: Integer conversion functions

Both, Matlab and Scilab, allow mathematical operations for such integers. However, the results are different if the number of digits is insufficient to hold the result.

```
-->u = int8(100), v = int8(2)
 u  =
 100
 v  =
 2

-->u*v
 ans  =
 -56
```

The result is wrapped (200-256).

Matlab, on the other hand,

```
>> u=int8(100), v=int8(2)
u =
   100
v =
   2

>> u*v
ans =
   127
```

Unsigned integers give an analogous result.

```
-->x = uint8(100), y = uint8(2), z= uint8(3)
 x  =
 100
 y  =
 2
 z  =
 3

-->x*y
 ans  =
 200

-->x*z
 ans  =  44
```

Again, the last result is wrapped (300-256).

The same code for Matlab produces

```
>> x = uint8(100), y = uint8(2), z = uint8(3)
x =
   100
y =
   2
z =
   3

>> x*y
ans =
   200

>> x*z
```

```
ans =
   255
```

In both cases Matlab does not wrap around but outputs the largest number that can be represented with the given number of digits.
Operations between integers of different type are not allowed, but those between integers and standard (double precision) floating point numbers are, and the result is a floating point number.

```
-->typeof(z)
 ans  =
 uint8

-->typeof(2*z)
 ans  =
 constant
```

The variable `z`, defined in the previous example, is an unsigned one-byte integer. Multiply it by 2 and the result is a regular floating point number for which `typeof` returns the value `constant`.

3.2 Special Matrices

Like Matlab, Scilab has a number of functions that create "standard" matrices or matrices of random numbers. Many of them have the same or a very similar names. The arguments or the output may be slightly different.
The empty matrix [] in Scilab behaves slightly different than the corresponding [] in Matlab; in Scilab, for example,

```
-->a = []+3 9a
 a  =
    3.
```

whereas in Matlab

```
>>a = []+3  9b
a  =
    [].
```

While 9a is the default result, Scilab's behavior in this situation can be changed by invoking Matlab-mode.

```
-->mtlb_mode(%t)
-->a = []+3 9c
 a  =
     []
```

Scilab	Description
[]	Empty matrix
companion	Companion matrix
diag	Create diagonal matrix or extract diagonal from matrix
eye	Identity matrix (or its generalization)
grand	Create random numbers drawn from various distributions
ones	Matrix of ones
rand	Matrix of random numbers with uniform or normal distribution
sparse	Sparse matrix
sylm	Sylvester matrix (input two polynomials, output numeric)
testmatrix	Test matrices: magic square, Franck matrix, inverse of Hilbert matrix
toeplitz	Toeplitz matrix
zeros	Matrix of zeros

Table 3.4: Special matrices

With Matlab-mode true, Scilab $\boxed{\text{9c}}$ produces the same result Matlab $\boxed{\text{9b}}$ does.
The standard syntax with two arguments to define dimensions works for functions zeros, ones, rand, eye the same way it does for Matlab.

```
-->rand_mat = rand(2,3)
 rand_mat  =
    0.2113249     0.0002211     0.6653811
    0.7560439     0.3303271     0.6283918
```

However, the syntax used in the following example

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

-->rand_mat = rand(a)      10
 rand_mat  =
    0.02113249    0.0002211     0.6653811
    0.7560439     0.3303271     0.6283918
```

has been deprecated in Matlab; Matlab expects $\boxed{10}$ written as *rand_mat = rand(size(a))*; this also works in Scilab (in the sense that it does not throw an error), but produces a different result.

```
-->rand_mat1=rand(size(a))
 randmat1 =
    0.0002211    0.3303271
```

It creates a random matrix with the size of size(a), a 1×2 matrix.

Function grand is a "generalized" version of rand; it allows one to create random numbers drawn from a variety of distributions and even provides means to choose between several random-number generators.

3.3 Character String Variables

3.3.1 Creation and Manipulation of Character Strings

Character strings can be defined with single quotes or double quotes, but the opening quote must match the closing quote. Thus 11a and 11b below are equivalent.

```
-->test = "This is a test";     11a

-->test = 'This is a test';     11b
```

Function `length` produces a familiar result—the number of characters in the string.

```
-->length(test)
 ans  =
    14.
```

However, a character string in Scilab is not a vector but rather akin to a Matlab cell. Thus

```
-->size(test)
 ans  =
   1.    1.
```

This is the same result *size* would give in Matlab if `test` were a Matlab cell. Thus it is not surprising that strings can be elements of matrices.

```
-->sm = ['This is','a','matrix';
-->     'each element','is a','string']
 sm  =
  !This is       a     matrix  !
  !                            !
  !each element  is a  string  !

-->size(sm)
 ans  =
    2.    3.
```

Function `size` again gives the same result *size* would give for a Matlab cell array. In other words: in order to get an analogous representation in Matlab one would have to use a cell array. However, there is no analog in Matlab for the behavior of `length`; nevertheless, it is a straight-forward generalization of its behavior for an individual string.

```
-->length(sm)
 ans  =
   7.      1.    6.
   12.     4.    6.
```

The output of `length` is a matrix with the same dimension as `sm`; each matrix entry is the number of characters of the corresponding entry of `sm`. For many purposes this output is so useful that one gets to miss it in Matlab.
Function `emptystr()` returns an empty string or string matrix very similar to the function *cell* in Matlab.

```
-->length(emptystr(2,5))
ans =
  0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.
```

A handy function for many string manipulations is `ascii` which converts character strings into their ASCII equivalent (e.g. 'A' to 65, 'a' to 97) and vice versa. Function `ascii` does in Scilab what the pair *`char`* and *`double`* does in Matlab.
Strings can be concatenated by means of the + sign (this is an example of operator overloading)

```
-->s1 = 'Example';

-->s2 = 'of ';

-->s3 = 'concatenation';

-->ss1 = s1+' '+s2+s3'          12a
 ss1  =
 Example of concatenation      13a

-->length(ss1)
 ans  =
    24.
```

The following is also a legal Scilab statement; it creates a string matrix.

```
-->ss2 = [s1,' ',s2,s3]         12b
 ss2  =
 Example     of    concatenation      13b

-->length(ss2)
 ans  =
   7.     1.     3.     13.
```

In Scilab, statement $\boxed{\text{12a}}$ does what $\boxed{\text{12b}}$ would do in Matlab; in Scilab the variable `ss2` is a 4-element string vector—note the difference in the display of `ss1` $\boxed{\text{13a}}$ and `ss2` $\boxed{\text{13b}}$ where the exclamation marks in $\boxed{\text{13b}}$ indicate that `ss2` is a string vector. In Matlab, strings in cell arrays are in quotes.
Scilab's built-in function `strcat` can be used to transform string vector `ss2` into string `ss1` (compare $\boxed{\text{13c}}$ with $\boxed{\text{13a}}$).

```
-->ss1a = strcat(ss2)
 ss1a  =
 Example of concatenation      13c
```

Scilab	Description
ascii	Convert ASCII codes to equivalent string and vice versa
basename	Strip directory and file extension from a file name
blanks	Create string of blank characters
convstr	Convert string to lower or upper case
date	Current date as string
emptystr	Create a matrix of empty strings
grep	Find matches of strings in a vector of strings
gsort(a)	Sort elements/rows/columns of a
intersect(str1,str2)	Returns elements common to two vectors str1 and str2
isalphanum	Test if characters of a string are alphanumeric
isascii	Test if characters of a string are represented by 7-bit ASCII code
isdigit	Test if characters of a string are digits between 0 and 9
isletter	Test if characters of a string are letters
isnum	Test if a string represents a number
length	Matrix of lengths of the strings in a string matrix
msprintf	Convert, format, and write data to a string
msscanf	Read variables from a string under format control
part	Extract substrings from a string or string vector
pathconvert	Convert file path from Unix to Windows and vice versa
pol2str	Convert polynomial to a string
regexp	Find substring matching regular-expression string
sci2exp	Convert a variable into a string representing an expression
size	Size/dimensions of a Scilab object
strcat	Concatenate character strings
strcmp	Compare character strings
strcmpi	Case-insensitive comparison of character strings
strcspn(str1,str2)	Get number of characters in str1 before finding one of the characters in str2
strindex(str1,str2)	Return starting index/indices where string str2 occurs in str1
string	Convert number(s) into string(s)
stripblanks	Remove leading and trailing blanks from a string
strncpy(str,num)	Copy the first num characters from each entry of string matrix str
strrchr(str1,str2)	Copy the characters of the entries of string matrix str1 from the last occurrence of the corresponding entry ins string matrix str2
strrev	Reverse the order of the characters in the input string
strspn(str1,str2)	Get characters in str1 before finding one not in the characters in str2
strsubst(s1,s2,s3)	Substitute string s3 for s2 in s1
strtod	Convert string to numeric value (double)
tokens	Split string into substrings based on one or more “separators”
union(a,b)	Extract the unique common elements of a and b
unique(a)	Return the unique elements of string vector a in ascending order

Table 3.5: Functions that manipulate strings

As shown below, strcat can do even more. It has a second argument that allows one to concatenate elements of a string vector but insert a string between the individual elements. One can, for example, create a string containing the comma-separated elements of a string vector.

```
-->s=['a','b','c']
s  =


 a  b  c

-->strcat(s,'+')
 ans  =


 a+b+c
```

It does not matter if the strings are arranged in a row vector, column vector or a matrix.

```
-->norns = ['Urd';'Werdandi';'Skuld']
 norns  =
!Urd       !
!          !
!Werdandi  !
!          !
!Skuld     !

-->Nornen = strcat(norns,', ')
 Nornen  =
 Urd, Werdandi, Skuld
```

The string inserted between individual elements by strcat can be as long as desired. In this specific instance, where it consists of characters (comma and space) not present in the original strings, function tokens can be used to “decompose” Nornen to obtain the original strings.

```
-->norns1 = tokens(Nornen,[',',' '])
 norns1  =
!Urd       !
!          !
!Werdandi  !
!          !
!Skuld     !
```

Function tokens, which includes the functionality of Matlab's *strtok*, decomposes a string into substrings separated by one or more “tokens”, single characters in a vector. In the example above these characters are a comma and a space (arranged in string vector `[',',' ']` and not simply as string `', '`). Thus, in certain circumstances, tokens is the inverse of strcat.

The operator +, used above for strings, works for string matrices the same way it works for numeric matrices. As illustrated below, a single string “added” to a string matrix is prepended (or appended) to every matrix element.

```
-->cost = ['10' '100' '1000'; '1'  '13'  '-22']
 cost  =
!10  100  1000  !
!               !
!1   13   -22   !

-->new_cost= '$'+cost+'.00'
 new_cost  =
!$10.00  $100.00  $1000.00  !
!                           !
!$1.00   $13.00   $-22.00   !
```

Since indexing is used to identify elements of string vectors and string matrices the question is how one would access individual characters in a string. As far as extracting characters is concerned this can be done with function part.

```
-->test = 'This is a test';
-->part(test,1)     // Select the first character
 ans  =
 T

-->part(test,1:4)   // Select the first 4 characters
 ans  =
 This
```

The second argument of part is a vector of indices. For every index that exceeds the length of the string a blank is appended to the output of part. This is illustrated in 14a.

```
-->str1 = part(test,11:20) 14a
str1  =
 test

-->length(str1)
 ans  =
    10.
```

The string str1 consists of the requested 10 characters, and 14b below shows that the last six characters of str1 are indeed blanks (ASCII code 32).

```
-->ascii(str1) 14b
 ans  =
116.   101.   115.   116.   32.   32.   32.   32.    32.    32.
```

A more general approach to creating string matrices with equal elements uses emptystr together with the + operator

```
-->const=emptystr(5,3)+'constant'
 const  =
```

```
!constant  constant  constant  !
!                              !
!constant  constant  constant  !
!                              !
!constant  constant  constant  !
!                              !
!constant  constant  constant  !
!                              !
!constant  constant  constant  !
```

It is worth reviewing a few more of the functions shown in Table 3.5.

`grep(vstr1,str2)` searches for occurrence of string `str2` in string vector `vstr1`; returns a vector of indices of those elements of `vstr1` where `str2` has been found or an empty vector if `str2` does not exist in any element of `vstr1`.

`strindex(str1,vstr2)` looks for the position in string `str1` of the character string(s) in string vector `vstr2`. Function `strindex` differs from `grep` in that its first input argument is a string whereas the first argument of `grep` can be a string vector. The index vector output by `grep` refers to elements of the string vector `vstr1` whereas the index vector output by `strindex` refers to the position of the elements of `vstr2` in string `str1`. This is illustrated by the following example.

```
-->str1 = 'abcdefghijkl';

-->idx1 = grep(str1,'jk')       //String 'jk' is in str1(1)
 idx1  =
    1.

-->idx2 = strindex(str1,'jk')   //String 'jk' is in str at position 10
 idx2  =
    10.

-->str2 = ['abcdefghijkl','xyz','jklm'];

-->idx3 = grep(str2,'jk')       //String 'jk' is in str2(1) and str2(3)
 idx3  =
   1.    3.

-->idx4 = grep(str2,['jk','y']) //Strings 'jk' or 'y' are in str2(1)
                                //str2(2), and str2(3)
 idx4  =
   1.    2.    3.
```

This means that `grep` with some additional checking can be used to emulate the Matlab function *`ismember`* for `string` arguments (a generally more useful implementation of `ismember` — it works for numeric and string vectors — is reproduced on pages 38 ff.).

```
function bool=ismember(strv1,strv2)
```

```
    // Function outputs a boolean vector the same size as strv1.
    // bool(i) is set to %t if the string strv1(i) is equal to
    // one of the strings in strv2

    bool = ~ones(strv1);          // Create a boolean vector %f
    [idx1,idx2]=grep(strv1,strv2); // Find indices of strv1 and strv2
                                   // for which there is a match
    if idx1 == []
      return
    end

    // Eliminate indices for which an element of strv2 is only
    // a substring in strv1

    temp1 = strv1(idx1);
    temp2 = strv2(idx2);
    bool(idx1(temp1(:)  == temp2(:))) = %t;
endfunction
```

Like its Matlab equivalent the function `msscanf` can be use to extract substrings separated by spaces and numbers from a string.

```
-->str='   Weight: 2.5 kg';
-->[a,b,c] = msscanf(str,'%s%f%s')
 c  =
 kg
 b  =
    2.5
 a  =
 Weight:

-->typeof(a)
 ans  =
 string

-->typeof(b)
 ans  =
 constant
```

The format types available are `%s` for strings, `%e`, `%f`, `%g` for floating-point numbers, `%d`, `%i` for decimal integers, `%u` for unsigned integers, `%o` for octal numbers, `%x` for hexadecimal numbers, and `%c` for a characters (white spaces are not skipped). For more details see the help file for `scanf_conversion`.
In the context of the next subsection the function `sci2exp` may prove useful. It converts a variable into a string representing a Scilab expression. An example is

```
-->a=[1 2 3 4]'
 a  =
!   1. !
!   2. !
!   3. !
!   4. !

-->stringa = sci2exp(a)
 stringa  =
 [1;2;3;4]
```

Regular expressions based on the PCRE syntax are now supported for functions `grep`, `strindex`, and `strsubst`. They require the `'r'` flag, an additional input argument, to be set. Function `regexp`, which corresponds to Matlab's *`regexp`*, is also available.

3.3.2 Strings with Scilab Expressions

Like Matlab, Scilab allows execution of strings with Scilab statements and expressions. There are three possible functions with slightly different features.

Scilab	*Matlab*	
eval	*eval*	Evaluate string vector with Scilab expressions
evstr	*eval*	Evaluate string vector with Scilab expressions
execstr	*eval*	Evaluate string vector with Scilab expressions or statements

Table 3.6: Functions that evaluate strings with Scilab expressions

While there is a Scilab function eval, the best functional equivalent to Matlab's *eval* is execstr. Thus

```
-->execstr('a=1+sin(1)')


-->a
 a  =
    1.841471
```

Note that the execstr does not echo the result even though there is no semicolon at the end of the statement. A more elaborate use of execstr is

```
-->ier = execstr(['a=2','b=3^a'],'errcatch','n')
 ier  =
    0.


-->a,b
 a  =
    2.
 b  =
    9.
```

This code fragment illustrates that the first input argument of execstr can be a string vector. Since the second input argument 'errcatch' is given, an error in one of the statements of the first argument does not issue an error message. Instead, execstr aborts execution at the point where the error occurred, and resumes with ier equal to the error number. In this case the display of the error message is controlled by the third input argument ('m' ⇒ error message, 'n' ⇒ no error message).

A practical example of the use of execstr is the implementation, on page 87, of the return command in the simulation of a search path for the execution of a Scilab script.

In Scilab eval evaluates a vector of Scilab expressions. Thus

```
-->c = eval(['1+sin(1)';'1+cos(1)'])   15a
 c  =
!   1.841471  !
!   1.5403023 !
```

Note that eval('a=1+sin(1)') produces the error message

```
 Warning: obsolete use of = instead of ==
%h = a=1+sin(1)
      !
at line       3 of function %eval                    called by :
line    17 of function eval                     called by :
eval('a=1+sin(1)')
 !--error     4
undefined variable : a
at line       2 of function %eval                    called by :
```

```
line    18 of function eval                    called by :
eval('a=1+sin(1)')
```

Scilab expects an expression and interprets the `=` as a typo, assumes that the user really means `==`, and then finds that `b` is undefined.
The Scilab command `evstr` is very similar to `eval`; it, too, works only with expressions. However, while `eval` has no provisions to allow user-defined error handling, `evstr` will trap errors if used with two output arguments.

```
-->[c,ier] = evstr(['1+sin(1)';'1+cos(1)'])  15b
 ier =
    0.
 c  =
!   1.841471  !
!   1.5403023 !
```

If an error occurs, `ier` is set to the error number, but the function does not abort. The following is an example where the second expression of the of the argument has a syntax error.

```
-->[c,ier] = evstr(['1+sin(1)';'1+-cos(1)'])
 ier  =
    2.
 c  =
     []
```

The function does not abort, but `ier` is set to 2.
Note: since all the variables of the whole workspace (that are not shadowed) are available to these three functions there is generally no need for an equivalent to Matlab function *`evalin`*.

3.3.3 Symbolic String Manipulation

Scilab has several function that treat strings as variables and perform symbolic operations. An examples is `trianfml` which converts a symbolic matrix to upper triangular form.

```
-->mat = ['a','b+c','d';'-b*a','0','a+b';'b','1','-1']
 mat  =
a      b+c   d

-b*a   0     a+b

b      1     -1

-->tri = trianfml(mat)
 tri  =
b  1    -1

0  b*a  b*(a+b)-b*a
```

```
0   0     b*a*(b*d+a)-(b*(b+c)-a)*(b*(a+b)-b*a)
```

A symbolic matrix can be evaluated by means of the function `evstr` discussed above.

```
-->a = 1,b = -1,c = 2,d = 0
 a  =
    1.
 b  =
  - 1.
 c  =
    2.
 d  =
    0.

-->nummat = evstr(tri)
 nummat  =
! - 1.    1.  - 1. !
!   0.  - 1.    1. !
!   0.    0.    1. !
```

There are several functions such as `solve` and `trisolve` that operate on symbolic matrices and `addf`, `subf`, `mulf`, `ldivf`, and `rdivf` that operate on symbols representing scalars. What exactly they do can be found by looking at their help files.

3.4 Boolean Variables

Boolean (logical) variables are represented by `%t` (true) and `%f` (false). Since Scilab's main initialization file `scilab.start` equates the corresponding upper-case and lower-case variables they can also be used with capital letters (`%T`, `%F`). This is similar to Matlab which, in Version 6.5, introduced boolean variables *true* and *false*[1]. While intrinsically different from numbers, they are displayed as 1 and 0. This analogy is illustrated in the following table which shows a line-by-line comparison of corresponding Scilab and Matlab statements.

<table>
<tr><th>Scilab</th><th>Matlab</th></tr>
<tr><td><pre>
-->index = [1 1]
 index =
   1.   1.

-->bool = [%t,%t]
 b =
 T T

-->a = [1 2]
 a =
   1.   2.

-->a(index)
 ans =
   1.   1.

-->a(bool)
 ans =
   1.   2.
</pre></td><td><pre>
>> index = [1 1]
index =
     1   1

>> bool = [true,true]
b =
     1   1

>> a = [1 2]
a =
     1   2

>> a(index)
ans =
     1   1

>> a(bool)
ans =
     1   2
</pre></td></tr>
</table>

Table 3.7: Comparison of the use of boolean variables as array indices in Scilab and Matlab

When displayed on the screen in Matlab, vectors *n* and *b* look exactly the same. Nevertheless, they are different

```
>> islogical(n)
 0
>> islogical(b)
 1
```

and, when used as indices for the vector `a`, they produce different results. But, fortunately, these results agree with those obtained with Scilab.
There are three operators, well known from Matlab, that operate on boolean variables. They are shown in Table 3.8. Scilab does not know the "short-circuit" logical AND (`&&`) and OR (`||`).

[1] Actually, *true* and *false* are functions analogous to *zeros*, *ones*, *NaN*, etc., and *true* is the same as *true(1)*.

&	logical AND
~	logical NOT
|	logical OR

Table 3.8: Boolean operators

In the proper context, both Scilab and Matlab treat numeric variables like logical (boolean) variables; any real numeric variable $\neq 0$ is interpreted as `true` and `0` is interpreted as `false`. Thus

```
-->if -1.34
-->  a=1;
-->else
-->  a=2;
-->end
-->a
 a  =
    1.
```

Interestingly, Scilab itself is not very consistent regarding the use of boolean variables . The two functions `exists` and `isdef` do the same thing: they check if a variable exists (actually, `isdef` is a script that calls `exists`). However, `isdef` outputs `T` if the variable exists and `F` otherwise, whereas `exists` outputs 1 and 0, respectively. In this sense the function `bool2s`, which converts a boolean or a numeric matrix to a matrix of 1s and 0s, can be considered as having boolean output. If `a` is a numeric matrix then `b = bool2s(a)` creates a matrix `b` where all non-zero entries of `a` are 1.
If `a` is a boolean matrix then `b = bool2s(a)` creates a matrix `b` where entries are 1 where those of `a` are `%t` and 0 where entries of `a` are `%f`. The same result — even without an execution-time penalty — can be achieved by adding 0 to the boolean matrix `a`.
Since there is no Scilab function analogous to *`false`*, *`true`*, or *`logical`* a similar trick can be used to create a boolean matrix or vector.

```
-->false = ~ones(1,10)
 false  =
 F F F F F F F F F F

-->true = ~zeros(1,10)
 true  =
 T T T T T T T T T T
```

Like in Matlab, boolean variables can be used in arithmetic expressions where they behave as if they were 1 and 0, respectively.

```
-->5*%t
ans  =
    5.
```

```
-->3*%f
ans  =
    0.
```

Table 3.9 lists a number of functions that output or use boolean variables.
Functions and and or are functional equivalents of Matlab's functions *all* and *any*, respectively.[2] Function and(a) returns the boolean variable %t if all entries of a are %t (for a boolean matrix) or non-zero (for a numeric matrix).

```
 a  =
!   0.    1. !
!   2.    3. !

-->and(a)
 ans  =
  F

-->and(a,'r')    16
 ans  =
 F T
```

In the example above a has a zero entry in the upper left corner; hence, the answer is `false`. With the optional second argument 'r' (line 16), and analyzes the columns of a and outputs a row vector. The first column contains a zero; hence the first element of the output vector is f. Matlab's *all* would output an analogous logical vector.
Function or(a) returns the boolean variable %t if at least one entry of a is %t (for a boolean matrix) or non-zero (for a numeric matrix). Hence, for the same matrix a

```
-->or(a)
 ans  =
  T

-->or(a,'c')
 ans  =
 T
 T
```

With the second argument 'c' function or analyzes the rows and puts out a column vector. Since each row has at least one non-zero element, all entries of the output are %t. The analog to Matlab's *any* is or with the second input argument 'r'.
Like Matlab, Scilab always evaluates all terms of a logical expression; it does not, say, evaluate an expression from left to right and stop as soon as the result is no longer in question. Thus the statement

```
bool = exists('a') & a > 0
```

[2]See also the discussion of max, min, etc. on page 61

Scilab	Description
and(a)	Output %t if all entries of the boolean matrix a are true
bool2s	Replace %t (or non-zero entry) in matrix by 1 and %f by zero
exists(a)	Test if variable a exists
find(a)	Find the indices for which boolean matrix a is true
isascii	Test if characters of a string are represented by 7-bit ASCII code
iscell(a)	Test if variable a is a cell array
iscellstr(a)	Test if variable a is a cell array of strings
isdef(a)	Test if variable a exists
isdigit	Test if characters of a string are digits between 0 and 9
isdir(a)	Test if variable a is a directory path
isempty(a)	Test if variable a is empty
iserror	Test if error has occurred
isglobal(a)	Test if a is a global variable
isfield(a,b)	Test if b is a field of structure a
isinf(a)	Test if a is infinite
isletter	Test if characters of a string are letters
isnan(a)	Output boolean vector with entries %t where a is %nan
isnum	Test if a string represents a number
isnum	Test if a string represents a number
isstruct(a)	Test if a is structure
mtlb_mode	Test for (or set) Matlab mode for empty matrices
or(a)	Output %t if at least one entry of the boolean matrix a is true
simp_mode	Test for (or set) simplification mode for rational expressions
with_texmacs	Test if Scilab as been called by TeXmacs

Table 3.9: Boolean variables and functions that operate on, or output, boolean variables

will fail with an error message if the variable a does not exist even though the fact that exists('a') is false also means that bool is false. This statement needs to be split up.

```
bool = exists('a')
if bool
   bool = a > 0;
end
```

Some of the constructs discussed above are used in the following example of an emulation of the Matlab function *ismember*. For example, the Matlab statements

```
>>vstr = {'abcd','abc','xyz','uvwx'};
>>str = 'abc';
>>index = ismember(vstr,str)
 index =
     0     1     0     0
```

produce the same result as the analogous Scilab statements

```
-->vstr = ['abcd','abc','xyz','uvwx'];
-->str = ['abc','xy'];
-->index = ismember(vstr,str)
 index  =
 F T F F
```

if the function ismember is defined as

```
function bool=ismember(a,b)
// Find elements in vector "a" that are also in vector "b".
// Return logical vector "bool" of the same length as "a". An element of
// "bool" is true if the corresponding element in "a" is also in "b".
// "a" and "b" can be numeric vectors or string vectors
// INPUT
// a     numeric or string vector
// b     numeric or string vector (same type as "a")
// OUTPUT
// bool  boolean vector with the same length as an element lv(k) is true
//       if a(lv(k)) exists in b.  Hence, a(lv) are the common elements

  if type(a) ~= type(b)
     disp('Input arguments must be of the same type')
     error(' Abnormal termination')
  end

  if type(a) == 1
     if a == []
        bool= [];
        return
     end
  elseif type(a) == 10
     if a == ''
        bool=[];
        return
     end
  else
     error(' This function does not work with variables of type ' ...
            +string(type(a))+' ('+typeof(a)+')')
  end


 [ua,index]=myunique(matrix(a,1,-1));
 [temp,idx]=gsort([ua,unique(matrix(b,1,-1))],'g','i');

idx1=temp(2:$) == temp(1:$-1);
```

```
if ~or(idx1)
   bool=~ones(a);
   return
end

idx2=find(idx1);
idx2=min(idx(idx2)',idx(idx2+1)');
bool=~ones(ua);
bool(idx2)=%t;
bool=bool(index)
endfunction
```

The two output arguments of Scilab's function `unique` correspond to the first two output arguments of Matlab's function *`unique`*. However, Matlab's *`unique`* has an optional third output argument which allows one to recover the original input vector from its sorted unique elements. Function `myunique`, used in `ismember` above and defined below, is similar to Scilab's function `unique`, but its second output argument corresponds to the third output argument of Matlab's *`unique`*.

```
function [u,index]=myunique(a)
// Sort input vector and eliminate duplicate elements
// An optional second output allows recreation of the original input vector
// INPUT
// a      numeric or string vector
// OUTPUT
// u      vector a sorted and without duplicate entries
// index  optional index vector such that a = u(index)

if a == []
   u=[];
   index=[];
   return
end

 [a,index]=gsort(a,'g','i')
num=[1:size(a,'*')];
bool=a(2:$) == a(1:$-1);
bool=[0,matrix(bool,1,-1)];
u=a(~bool)

if argn(1) > 1
   num=num-cumsum(bool);
   [dummy,index]=gsort(index,'g','i');
   index=num(index);
end
endfunction
```

3.5 Cell arrays

The entries of a numeric array are numbers. The cells of a cell array, on the other hand, can contain not only numbers but any other Scilab object such as strings, matrices, other cell arrays, etc. Scilab's cell arrays, however, are not quite like Matlab's cell arrays — in fact, if one needs to access the content of a cell, an ordinary Scilab `list` is closer to a Matlab cell array.[3] Scilab's cell arrays, on the other hand, are "typed matrix-oriented lists" discussed later in this manual.

Scilab	Description
`cell`	Create a cell array with empty cells
`cell2mat`	Convert a cell array into a matrix
`iscell(a)`	Test if variable `a` is a cell array
`makecell`	Create a cell array and initiate its cells

Table 3.10: Functions that create, or operate on, cell arrays

Table 3.10 shows functions related to cell arrays. There are two functions, `cell` and `makecell`, that create cell arrays. The former, of course, is known from Matlab; as shown in the example below, it works like its Matlab counterpart. The latter not only creates a cell array but also populates it with values. There must be a value for each entry, and the cells are filled one row after the other. In the example below, `makecell` creates a 2×3 cell array. The first two cells in the first row contain numeric values, the third contains a numeric array. The bottom row contains strings.

```
-->c = cell(2,3)
 c =
!{} {} {} !
!          !
!{} {} {} !

-->cc = makecell([2,3],1,2,[3,4],'one','two','three')
 cc =
!1      2     [3,4]   !
!                     !
!"one" "two" "three" !
```

The cell array `c` created by function `cell` is simply an empty container that can be filled with Scilab objects. In order to do that one needs to use the following syntax:

```
-->c(1,2).entries = 2;
-->c(2,1).entries = [1,2,3];
-->c(2,3).entries = %t
c =
!{}       2      {} !
```

[3]Ordinary lists are discussed in Section 3.7.1 beginning on page 46

```
!                   !
![1,2,3]  {}    %t !
```

The same syntax is used to retrieve objects from a cell of a cell array

```
-->bool=c(2,3).entries
 bool =
  T

-->type(bool)
 ans =
 4.

-->typeof(bool)
 ans =
 boolean
```

In contrast, `c23`, as computed below, is a cell (type `ce`).

```
-->c23=c(2,3)
 c23 =
 %t

-->type(c23)
 ans =
 17.

-->typeof(c23)
 ans =
 ce

->iscell(c23)
 ans =
 T

-->c23.entries
 ans =
 T
```

The meaning of the output of `typeof` and `type` can be found in Table 3.1 on page 17.
It one extracts the cell content from two or more cells the result is a list.

```
-->cc1to2=cc(1:2,1).entries;

-->typeof(cc1to2)
 ans =
 list
```

```
cc1to2(1)
 ans =
 1

cc1to2(2)
 ans =
 one
```

There are two ways to compute the dimensions of a cell array; they are illustrated in the following code fragments by means of cell array `cc` defined above:

```
-->cc.dims
 ans =  2 3

-->size(cc)
 ans =
 2.  3.
```

The output of the first one shows the dimensions of `cc` as 32-byte integers, the latter as doubles.

3.6 Structures

A structure is a convenient container for disparate data that can be stored and retrieved in a self-explanatory way. Let us assume we need to store, for Toy Store # 31, toys for sale, their price, and the quantities on hand. Toys are represented by strings with their name; price and quantities are numeric values. We could store each in its own vector, but it is more convenient to have them all in one place. This is where structures come in. In this specific case we create a structure, called `toys1`, with fields `'shop'`, `'currency'`, `'name'`, `'price'`, and `'quantity'`. This can be done in many different ways; for example, by means of the `struct` function.

```
-->toys1=struct('shop', 'Store # 31', ..
-->            'currency', '$', ..
-->            'name', ['doll','truck','game station'], ..
-->            'quantity', [75,102,7], ..
-->            'price', [39.90,12.99,299.95])
 toys1 =
     shop:  "Toy Store # 31"
  currency:  "$"
      name:  ["doll","truck","game station"]
  quantity:  [75,102,7]
     price:  [39.9,12.99,299.95]
```

Function `struct` must have an even number of arguments (zero is allowed). The odd-numbered arguments, the names of the "fields" of the structure, must be strings.
The above structure, `toys1`, has five fields with names `shop`, `currency`, `name`, `quantity`, and `price`. Each field name is followed by a valid Scilab object (the even-numbered arguments of

struct). In the example above they are two strings, a string matrix and, two numeric matrices. The first entry in field name is 'doll'. The first entry in field quantity is 75, the number of dolls available for sale, and the first entry in field price is the price of the doll, \$39.90 (since the currency is \$).
There are two ways to access fields of a structure. One is the "structure.fieldname" syntax familiar from Matlab.

```
-->toys1.shop
 ans =
 Toy Store # 31"
```

The other uses parentheses to access the content of a field (in this example the field shop).

```
-->toys1("shop")
 ans =
 Toy Store # 31"
```

Note that there is no dot (".") after the structure name and that the field name is quoted, i.e. it is a string. This latter form has two advantages: there are no restrictions on the field name (it can contain blanks, special characters, etc. and can be longer than 23 characters), and the field name can be a string variable. In this respect it resembles Matlab's syntax *toys1.('shop')* (notice the dot after the structure name *toys1*).
On the other hand, if one uses the "structure.fieldname" syntax, field names must satisfy the same restrictions as variable names (e.g. no blanks, see page 4); Function struct does not complain about illegal field names; however, an error will be thrown if one tries to access a field with an illegal name.
If we need to find the number of trucks in Store #31 we simply type

```
-->number = toys1.quantity(2)
 number =
     102.
```

since truck is the second entry in field name. This shows that one way to reference a field of a structure is to append the field name, preceded by a dot, to the structure name. Similarly, the price of the truck is

```
-->price = toys1.price(2)
 price =
     12.90
```

Thus we can write

```
-->disp('In shop ""' + toys1.shop + '"" the price of the ' + ..
-->toys1.name(3) + ' is '+toys1.currency + string(toys1.price(3)))

 In shop "Store No 31" the price of the game station is $299.95
```

Using function struct is but one way to define a structure. Another method is:

```
-->toys2.shop = 'Toy Store No 69';
-->toys2.currency = '$';
-->toys2.name =[ "doll","truck","game station"];
-->toys2.quantity = [14,49,3];
-->toys2.price=[34.90,11.99,279.95]
toys2 =
     shop:  "Toy Store No 69"
  currency:  "$"
      name:  ["doll","truck","game station"]
  quantity:  [14,49,3]
     price:  [34.9,11.99,279.95]
```

Since the two structures toys1 and toys2 have the same fields they can be concatenated to form a structure array.

```
-->toys=[toys1,toys2]
 toys =
1x2 struct array with fields:
    shop
    currency
    name
    quantity
    price

-->size(toys)
 ans =
    1.  2.
```

It is worth mentioning that the above example is but one way to store the information about the toys in a structure. Another approach could be

```
-->toysx=struct('shop', 'Toy Store # 31', ..
-->            'currency', '$', ..
-->            'doll', [75, 39.90], ..
-->            'truck', [201, 19.95], ..
-->            'game_station', [7, 299.95])
 toysx =
         shop:  "Toy Store # 31"
     currency:  "$"
         doll:  [75, 39.90]
        truck:  [201, 19.95]
 game_station:  [7, 299.95]
```

In this case information about each toy is stored in its own field. The toy name is the field name; note that “game station” needs an underscore to avoid an illegal blank space in the field name.

Scilab	Description
`getfield`	Get field names and values from a structure
`isfield(a,b)`	Test if `b` is a field of structure `a`
`isstruct(a)`	Test if `a` is a structure
`length(a)`	Number of user-created fields of `a` + 2
`null()`	Delete an element of a list
`setfield`	Set field names and values of a structure
`struct`	Create a structure
`size(a)`	Size of structure `a`

Table 3.11: Functions that create, or operate on, structures

While Scilab has functions `getfield` and `setfield` they work differently than the like-named functions in Matlab. In particular,

```
-->fields=getfield(1,toys)
 fields =
!st dims shop currency name quantity price !
```

outputs a seven-element string vector. Its first entry, `'st'` is the type of the matrix-oriented typed list (internally, a Scilab structure is an `mlist`). The second entry is a field that Scilab created: the dimension of the structure. The other entries are the fields of `toys` defined above. Hence, it could be used to emulate Matlab's function *`fieldnames`*.

```
-->temp=getfield(1,toys); fieldnames=temp(3:$)
 fieldnames =
!shop currency name quantity price !
```

The values associated with the user-created fields can be obtained by choosing the first argument of `getfields` ≥ 3.

```
-->value1=getfield(3,toys(2))
 value1 =
 Store No 69
```

Thus `value1` is a string representing the value of `toys(2).shop`, the field `shop` of the second entry of `toys`. If we drop the index we get

```
-->value1=getfield(3,toys)
 value1 =
    value1(1)
 Store No 31
    value1(2)
 Store No 69
```

In this case `value1` is a two-entry list with the names of the two shops. The problem with this approach is that one needs to know the sequence of the fields of the structure. The following sample function shows a quick and dirty emulation that works like Matlab's *`getfield`*.

```
function field=getfield4st(st,fieldname)
  //\Get the value of a field of structure "st"

  //INPUT
  //st          structure
  //fieldname   name of the field to retrieve
  //OUTPUT
  //field       value of the field

  //  Find the field names
  fields=getfield(1,st);

  //  Check if "fieldname" is one of the fields of structure "st"
  index=find(strcmp(fields(2:$),fieldname) == 0);
  if isempty(index)
    error('Field ""'+fieldname+'"" is not present in this structure.')
  end

  //  Extract the value of the field specified
  field=getfield(index+1,st);          17
 endfunction
```

A function providing Matlab's *setfield* functionality is is quite analogous. More interesting is an emulation of Matlab's *rmfield* which removes one or more fields from a structure. A simple version, which removes just one field, is exactly like function `setfield` above, except that statement $\boxed{17}$ needs to be replaced by the three statements

```
fields(index+1)=[];          // Remove the string "fieldname" from
                             // the list of fields
setfield(1,fields,st)
setfield(index+1,null(),st) // Remove the object associated with
                             // field "fieldname"
```

The last statement illustrates a use of function `null()`, which has no input arguments.

3.7 Lists

Lists are Scilab data objects; they come in three flavors: ordinary lists, `list`, which behave like Matlab cell vectors (one-dimensional cell arrays), typed lists, `tlist`, and matrix-oriented typed lists, `mlist`. The latter two can be used to emulate Matlab cell arrays and structures; in fact, the cell arrays and structures described above are matrix-oriented lists.

Someone coming from Matlab might be inclined to disregard lists; and for "quick and dirty" programming this appears reasonable. However, any serious user should avail himself of the power that typed lists and matrix-oriented typed lists offer. Operator overloading is just one example, and the flexibility offered by Scilab is unmatched by Matlab.

3.7.1 Ordinary lists (list)

A list is a collection of data objects. Its Matlab equivalent is a one-dimensional cell array. Like Matlab cell arrays these objects need not be of the same type. They can be scalars, matrices, character strings, string matrices, functions, as well as other lists. An example is (remember that both single quotes (') and double quotes (") can be used to denote strings in Scilab):

```
-->a_list=list('Test',[1 2; 3 4], ...
             ['This is an example'; 'of a list entry'])
a_list  =

      a_list(1)
Test

      a_list(2)
  1.   2.
  3.   4.

      a_list(3)
This is an example

of a list entry
```

Individual elements can be accessed with the usual index notation. Thus

```
-->a_list(1)
 ans  =
 Test
```

This is different from the way Matlab works. If *a_list* were a Matlab cell array the same result would be achieved by *a_list{1}* — note the curly brackets — whereas *a_list(1)* would be a one-element cell array which contains the string *'Test'* (note the quotes).

Scilab	Description
getfield	Get a data object from a list
length	Length of list
list	Create a list
lstcat	Concatenate lists
mlist	Create a matrix-oriented typed list
null	Delete an element of a list
setfield	Set a data object in a list
size	Size of a list or typed list (but not matrix-oriented typed list)
tlist	Create a typed list

Table 3.12: Functions that create, or operate on, lists

Using the index 0 one can prepend an element to the list

```
-->a_list(0)=%eps;
```

This pushes all elements of `a_list` back. Hence

```
-->a_list(2)
 ans  =
 Test
```

What used to be the first element is now the second one. The Matlab equivalent would be *a_list=[{eps},a_list]*; it is more flexible since any number of elements (not just one) could be prepended and the augmented cell array could be saved under a new name; e.g.

```
a_list1=[{eps},a_list]
```

However, in Scilab the same functionality could be created by overloading (see Section 6.3). Appending elements works the same way.

```
-->a_list(8)='final element';
```

assigns the string `'final element'` to element 8 of the list `a_list`. Elements 5 to 7 are undefined. Thus

```
-->a_list(5)
!--error   117  List element    5 is Undefined
```

The `null` function can be used to delete an elements of a list. For example,

```
-->aa=list(1,2,3,4,5);

-->aa(3)=null()
  aa  =
       aa(1)
    1.
       aa(2)
    2.
       aa(3)
    4.
       aa(4)
    5.
```

The third element has been removed from the list. The list has now only four elements. It is not possible to delete more than one element at a time in this way; e.g. the attempt to delete elements 2 and 4 via `aa([2,4])=null()` generates an error message.
Lists allow tuple assignments, i. e. more than one variable can be assigned a value in a single statement. With the list `aa` defined above

```
-->[u,v]=aa(2:3)
```

```
v =
4.
u =
2.
```

This kind of tuple assignment can also be used with typed lists.
The functions `size` and `length` have been appropriately overloaded for lists.

```
-->blist = list('abcd','efg',1.3,[1 2; 3 4],list('1',1))
 blist  =
       blist(1)
 abcd

       blist(2)
 efg

       blist(3)
    1.3

       blist(4)
!   1.    2. !
!   3.    4. !

       blist(5)

       blist(5)(1)
 1
       blist(5)(2)
    1.

-->length(blist)
 ans  =
    5.

-->size(blist)
 ans  =
    5.
```

Note that `length` and `size` give the same result — one number. Lists are inherently one-dimensional objects. But this last example illustrates how one can emulate a two-dimensional cell array, i. e. a multi-dimensional object where an element is defined by two indices (this may be desirable for tables where some columns have alphanumeric entries while others are purely numeric). One can write it as a list of lists. The following is an example.

```
-->cell=list(list(),list());
```

```
-->cell2d(1)=['first','second','third'];

-->cell2d(2)=[1,2,3];

-->cell2d(1)(3)
 ans  =
 third

-->cell2d(2)(3)
 ans  =
    3.
```

Nevertheless,

```
-->length(cell2d)
 ans  =
    2.
```

Thus `cell` is still a one-dimensional data object.

3.7.2 Typed lists (tlist)

Typed lists are lists with very useful properties. They allow the user to set up special kinds of data objects and to define operations on these objects (see Section 6.3 beginning on page 98). Examples are linear systems (type `'lls'`) or rational functions (type `'rational'`; see page 58).
The first element of a typed list must be a string (the type name or type) or a string vector. In the latter case the type name is the first element of the string vector; the other elements of this string vector are names (in the following called "fields") for the other entries of the typed list. An example is

```
-->my_tlist=tlist(['example','first','second'], 1.23,[1,2])
 my_tlist  =

       my_tlist(1)
!example  first  second  !

       my_tlist(2)
    1.23

       my_tlist(3)
!   1.    2. !

-->type(my_tlist)
ans =
16.
```

```
-->typeof(my_tlist)
ans =
example
```

Here the first element of the list is a three-element vector of character strings whose first element, `'example'`, identifies the type of list (type name). While this type name can consist of almost any number of characters (definitely more than 1024), it must not have more than 8 if one intends to overload operators for this typed list. The other elements of the first string vector, `first` and `second`, are the fields.

From a Matlab user's perspective the fact that typed lists can be used to represent Matlab structures is of greatest relevance here, and in this case the type name as represented by the first element of the first string vector can, in principle, be ignored[4]. The elements of `my_tlist` can be accessed in the usual way via indices.

```
-->my_tlist(1)
 ans  =
!example  first  second  !

-->my_tlist(2)
 ans  =
    1.23

-->my_tlist(3)
 ans  =
!   1.    2. !

-->my_tlist(1)(2)
 ans  =
 first
```

Displays of lists can become quite lengthy and confusing. Here, for display purposes, a function `show` is used (it is not part of the Scilab distribution, but too long to be reproduced here) which displays data objects in a more compact form and, for typed lists, is patterned after the format Matlab uses for structures. Thus

```
-->show(my_tlist)
LIST OF TYPE "example"
   first: 1.23
  second: 1  2
```

Section 6.3 shows how this kind of display can be made the default for displaying a typed list with a particular type name.

Elements of the typed list other than the first can be accessed in various ways. For example

[4] As shown above, the display of lists can be rather unwieldy. Fortunately, the way a typed list (or matrix-oriented typed list) is displayed can be overloaded to create, for example, a Matlab-like look. If this is desired then the type name plays a key role (see Section 6.3).

```
-->my_tlist('first')
 ans  =
   1.23

-->my_tlist('second')  18a
 ans  =
!   1.    2. !
```

This means that the second and subsequent elements of `my_tlist(1)` can be used as "names" for the second and subsequent elements, respectively, of `my_tlist`. But there is another way of using these names. It is a construct where a dot "." separates the name of a typed list and the name of the field—the representation of structures familiar to Matlab and C users.

```
-->my_tlist.first
 ans  =
    1.23

-->my_tlist.second  18b
 ans  =
   1.    2.
```

Thus a typed list can be accessed like a Matlab structure. Once a field is defined, different values can be assigned to it in the same way they would be assigned to a Matlab structure.

```
-->my_tlist.second = 'A new value';

-->my_tlist.second
 ans  =
 A new value
```

One advantage of 18a over 18b is that the field name need not satisfy requirements for a variable; it may contain white spaces and special characters not allowed for variable names. But more importantly, the field name may be computed by concatenating strings or it could be the element of a string vector.
In principle, the typed list `my_tlist` could have been defined as

```
-->my_tlist = tlist(['example','first','second']);

-->my_tlist.first = 1.23;

-->my_tlist.second = [1,2];
```

If `my_tlist` were to have one more element, it would have to be added first — e.g. via (remember that `$` means "last element" equivalent to *end* in Matlab);

```
-->my_tlist(1)($+1) = 'new';

-->my_tlist.new = 'value of new field';    19a;
```

```
-->show(my_tlist)
 LIST OF TYPE "example"
   first: 1.23
  second: 1  2
     new: value of new field
```

The statement 19a above could have been written as

```
my_tlist($+1) = 'value of new field';    19b
```

Generally speaking, the kth element of the first-element character string vector of a typed list is the field name of the kth element of the typed list.
Operator overloading—more specifically, insertion overloading (see page 102)—can be used to make adding new fields to a typed list as simple as adding fields to a structure in Matlab.
Lists can have other lists as elements. For example

```
-->record=tlist(['record','patient','invoice']);

-->record.patient=tlist(['patient','address','city','phone']);

-->record.patient.phone='123.456.7890';

-->record.invoice=1234.33;

-->record
 record  =

       record(1)
!record  patient  invoice  !

       record(2)

        record(2)(1)
!patient  address  city  phone  !

        record(2)(2)
    Undefined

        record(2)(3)
    Undefined

        record(2)(4)
 123.456.7890

       record(3)
```

```
        1234.33
```

With function `show` this typed list is displayed as:

```
-->show(record)
LIST OF TYPE "record"
  patient: LIST OF TYPE "patient"
      address: Undefined
         city: Undefined
        phone: 123.456.7890
  invoice:  1234.33
```

An element of a typed list can be removed the same way an element of an ordinary list is removed. However, the index or the name can be used. Thus, for the typed list `record` defined above the following four Scilab statements

```
-->record.patient.phone = null();
-->record.patient(4) = null();
-->record(2)(4) = null();
-->record(2).phone = null();
```

are equivalent.
A combination of `list` and `tlist` can be used to create a Scilab equivalent of a structure array.

```
-->seis1 =
tlist(['seismic','first','last','step','traces','units'],0,[],4,[],'ms');

-->seismic = list(seis1,seis1,seis1);

-->for ii=1:3
-->  seismic(ii).last=1000*ii;
-->  nsamp = (seismic(ii).last-seismic(ii).first)/seismic(ii).step+1;
-->  seismic(ii).traces=rand(nsamp,10);
-->end

-->show(seismic)
List element 1:
LIST OF TYPE "seismic"
   first: 0
    last: 1000
    step: 4
  traces: 251 by 10 matrix
   units: ms
List element 2:
LIST OF TYPE "seismic"
   first: 0
    last: 2000
```

```
     step: 4
   traces: 501 by 10 matrix
    units: ms
List element 3:
LIST OF TYPE "seismic"
    first: 0
     last: 3000
     step: 4
   traces: 751 by 10 matrix
    units: ms
```

Thus seismic is a list with three seismic data sets with the same start times but different end times, that can be individually addressed.

```
-->show(seismic(3))
LIST OF TYPE "seismic"
    first: 0
     last: 3000
     step: 4
   traces: 751 by 10 matrix
    units: ms
```

It is also straight forward to access fields of individual data sets. For example,

```
-->seismic(2).last
 ans  =
    2000.
```

3.7.3 Matrix-oriented typed lists (mlist)

A matrix-oriented typed list, mlist, is like a regular typed list discussed above — but with an important difference. This is illustrated by an example. The statement

```
-->an_mlist=mlist(['VVV','name','value'],['a','b','c'],[1 2 3])
 an_mlist  =

       an_mlist(1)
!VVV  name  value  !

       an_mlist(2)  
!a  b  c  !

       an_mlist(3)
!   1.    2.    3. !
```

20a

creates a matrix-oriented typed list, and the statements

```
-->an_mlist.name
```

```
 ans  = !a  b  c  !

-->an_mlist('name')
 ans  =
!a  b  c  !

-->an_mlist.value
 ans  =
!   1.    2.    3. !

-->an_mlist('value')
 ans  =
!   1.    2.    3. !
```

work as expected. However, elements cannot be accessed by index the way elements of a typed list can.

```
-->an_mlist(2)   20b
             !--error     4
undefined variable : %l_e
```

This is in spite of the fact that 20b looks exactly like 20a, the output created by function `mlist`. Also, the `size` function does not work with mlists. In practical terms, this means that matrix-oriented typed lists allow overloading the “extraction operator”. This appears to be the reason why structures and cell arrays in Scilab are implemented as matrix-oriented lists of types `st` and `ce`, respectively (see below).

```
-->ccc=cell(1,3)
 ccc =
!{}  {}  {}  !

-->type(ccc)
 ans =
17.

-->typeof(ccc)
 ans =
 ce
```

Numeric and ASCII type codes are listed in Table 3.1 on page 17.

3.8 Polynomials

If polynomials are a data type available with standard Matlab (there is, of course, the Symbolic Toolbox based on the Maple kernel) then, at least, I am not aware of them. In Scilab they can be created by means of function `poly`.

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

-->typeof(z)
 ans  =
 polynomial

-->typeof(p)
 ans  =
polynomial
```

In this example the first argument of `poly` is a vector of polynomial coefficients. Alternatively, it is also possible to define a polynomial via its roots.

```
-->p = poly([1 2 3],'z','roots')
 p  =
                2   3
  - 6 + 11z - 6z + z

-->roots(p)
 ans  =
!   1. !
!   2. !
!   3. !
```

The default for the third argument is actually `'roots'` and so it could have been omitted. It is also possible to define first the symbolic variable and then create polynomials via standard Scilab expressions.

```
-->s = poly(0,'s')  // This is a polynomial whose only zero is 0
 s  =
    s

-->p = 2 - 3*s + s^2
 p  =
              2
    2 - 3s + s

-->q = 1 - s
 q  =
    1 - s
```

Scilab	Description
bezout	Compute greatest common divisor of two polynomials
clean	Round to zero small entries of a polynomial matrix
cmndred	Create common-denominator form of two polynomial matrices
coeff	Compute coefficients of a polynomial matrix
coffg	Compute inverse of a polynomial matrix
colcompr	Column compression of polynomial matrix
degree	Compute degree of polynomial matrix
denom	Compute denominator of a rational matrix
derivat	Compute derivative of the elements of a polynomial matrix
det	Compute determinant of a polynomial or rational matrix
determ	Compute determinant of a polynomial matrix
detr	Compute determinant of a polynomial or rational matrix
diophant	Solve diophantine equation
factors	Compute factors of a polynomial
gcd	Compute greatest common divisor of elements of polynomial matrix
hermit	Convert polynomial matrix to triangular form
horner	Evaluate polynomial or rational matrix
hrmt	Compute greatest common divisor of polynomial row vector
inv	Invert rational or polynomial matrix
invr	Invert rational or polynomial matrix
lcm	Compute least common multiple elements of polynomial matrix
lcmdiag	Least common multiple diagonal factorization
ldiv	Polynomial matrix long division
pdiv	Elementwise polynomial division of one matrix by another
pol2str	Convert polynomial to a string
residu	Compute residues (e. g. for contour integration) of ratio of two polynomials
roots	Compute roots of a polynomial
rowcompr	Row compression of polynomial matrix
sfact	Spectral factorization of polynomial matrix
simp	Rational simplification of elements of rational polynomial matrix
simp_mode	Test for (or set) simplification mode for rational expressions
sylm	Sylvester matrix (input two polynomials, output numeric)

Table 3.13: Functions related to polynomials and rational functions

```
-->simp_mode(%f)    // Do not simplify ratios of polynomials

-->r = p/q
 r  =
                 2
    2 - 3s + s
    ----------
```

```
      1 - s

-->simp_mode(%t)    // Simplify ratios of polynomials

-->simp(r)
 ans  =
    2 - s
    -----
      1

-->type(r)
 ans  =
    16.

-->typeof(r)
 ans  =
 rational
```

The result of `type` indicates that `r` is a typed list and `typeof` tells us that it is a list of type `rational`.

Table 3.13 lists functions available in Scilab for manipulating polynomials and ratios of polynomials. One difference between computer algebra packages such as Mathematica, Maple, or Macsyma and this implementation of polynomial algebra is the precision. Scilab evaluates expression to its normal precision while the above packages maintain infinite precision unless requested to perform numerical evaluations.

Chapter 4

Functions

4.1 General

For someone coming from Matlab, Scilab functions are familiar entities. There are differences, though. One is that parentheses are generally required even if a function has no input arguments. There are two exceptions:

- The function is treated as a variable
- The function has at most one output argument and all input arguments are strings (command-style syntax).

Command-style Syntax: For any function that has at most one output argument and whose input arguments are character strings, the calling syntax may be simplified by dropping the parentheses. Thus

```
-->exec('fun1.sci')
-->exec 'fun1.sci'
-->exec fun1.sci     // Command-style syntax
```

are equivalent. The last form represents the command-style syntax (a command, possibly followed by one or more arguments; Matlab has a similar feature). More generally, if function `funct` accepts three string arguments then

```
funct('a','total','of three strings')
```

is equivalent to

```
funct a total 'of three strings'
```

Here the quotes around the last argument are required to prevent it from being interpreted as three individual strings. It even seems to work if the function accepts non-string arguments provided that these arguments are optional. In order to run a script, say `script.sce`, the command `exec('script.sce')` must be executed. The function `exec` has one required and two optional arguments (one of which is numeric). Nevertheless,

```
exec('script.sce')
exec 'script.sce'
exec script.sce
```

give all the same result.
Scilab provides one way of passing parameters to a function that is not available in Matlab: named arguments. This method of passing arguments is especially practical with function that have many input parameters with good default values — plot functions are typical examples. For example, the built-in function `plot2d` can be called as follows

```
plot2d([logflag],x,y,[style,strf,leg,rect,nax])
```

The first argument is an optional string that can be used to set axis graduation (linear or logarithmic). The next two arguments are the x-coordinates and y-coordinates of the function to be plotted. The last five arguments are optional again. Now suppose one wants to use the default values for all optional parameters except the curve legend (parameter `leg`). The parameter `logflag` is not a problem. If the first input argument is not a string the program knows it is not given as a positional parameter. But the defaults of `style` and `strf` would have to be given so that `leg` is at the correct position in the argument list. Hence, the statement would read as follows

```
-->plot2d(x,y,1,'161','Curve legend')
```

This, of course means that one has to figure out what the default values are. The simpler solution to this problem is to use named parameters

```
-->plot2d(x,y,leg='Curve legend')
```

Note that the name of the argument, `leg` is not quoted — it is not a string. The order of named parameters is arbitrary, but any positional parameters must come before named parameters. It is for example possible to specify the parameter `logflag` after all. For example,

```
-->plot2d(x,y,leg='Curve legend',logflag='ll')
```

creates the same plot, but with log-log axes. Of course, the same could be achieved by

```
-->plot2d('ll',x,y,leg='Curve legend')
```

In principle, any input argument could be supplied as a named parameter.

```
-->plot2d(x=x,y=y,leg='Curve legend')
```

but `plot2d` has internal checks that do not allow that. Also, named parameters are not compatible with variable-length input argument lists `varargin`.

4.2 Functions that Operate on Scalars and Matrices

4.2.1 Basic Functions

Quite a number of functions in Table 4.1 below, while having the same name, behave differently for matrices than their Matlab counterparts. The following example illustrates this difference. Scilab has an edge here.

```
-->mat = matrix([1:20],4,5)  // Create a matrix by rearranging a vector
 mat  =
   1.     5.     9.      13.     17.
   2.     6.     10.     14.     18.
   3.     7.     11.     15.     19.
   4.     8.     12.     16.     20.

-->[maxa,index] = max(mat)   // Find largest element and its location
 index  =
   4.     5.
 maxa  =
     20.

-->[maxr,idx] = max(mat,'r')
 idx  =
   4.     4.     4.     4.     4.
 maxr  =
   4.     8.     12.     16.     20.

-->maxc = max(mat,'c')
 maxc  =
   17.
   18.
   19.
   20.

-->[maxm,idxm] = max(mat,'m')
 idxm  =
   4.     4.     4.     4.     4.
 maxm  =
   4.     8.     12.     16.     20.
```

With a single argument, the Matlab version of *`max`* behaves just like `max(mat,'r')` does — it computes a row vector representing the maxima of every column. Likewise, `max(mat,'c')` computes a column vector with the maximum element in each row (equivalent to *`max(mat,[],2)`* in Matlab).
The last example uses parameter `'m'` to emulate the the output provided by Matlab's function *`max(mat)`*. In the same way, function `min` can be forced to emulate Matlab's *`min`*.
The functions `max` and `maxi` are equivalent as are `min` and `mini`.

There is no equivalent in Matlab for the behavior of `max(mat)` or `min(mat)` . It is particular the easy way of getting the indices of the largest element of a matrix that I consider extremely useful. The analogous behavior is found for Scilab functions `cumprod`, `cumsum`, `prod`, `sum`, and `st_deviation` .

Scilab	Description
`abs(a)`	Absolute value of `a`, $\|a\|$
`bool2s`	Replace `%t` (or non-zero entry) in matrix by 1 and `%f` by zero
`ceil(a)`	Round the elements of `a` to the nearest integers $\geq$`a`
`clean`	"Clean" matrices; i.e. small entries are set to zero
`conj`	Complex conjugate
`cumprod`	Cumulative product of all elements of a vector or array
`cumsum`	Cumulative sum of all elements of a vector or array
`fix(a)`	Rounds the elements of `a` to the nearest integers towards zero
`floor(a)`	Rounds the elements of `a` to the nearest integers $\geq$ `a`
`gsort(a)`	Sort elements/rows/columns of `a`
`imag`	Imaginary part of a matrix
`intersect(str1,str2)`	Returns elements common to two vectors `str1` and `str2`
`lex_sort`	Sort rows of matrices in lexicographic order
`linspace`	Create vector with linearly spaced elements
`logspace`	Create vector with logarithmically spaced elements
`max`	Maximum of all elements of a vector or array
`maxi`	Maximum of all elements of a vector or array
`mean`	Mean of all elements of a vector or array
`median`	Median of all elements of a vector or array
`min`	Minimum of all elements of a vector or array
`mini`	Minimum of all elements of a vector or array
`modulo(a,b)`	`a-b.*fix(a./b) if b~=0`; remainder of `a` divided by `b`
`pmodulo(a,b)`	`a-b.*floor(a./b) if b~=0`; remainder of `a` divided by `b`
`prod`	Product of the elements of a matrix
`real`	Real part of a matrix
`round(a)`	Round the elements of `a` to the nearest integers
`sign(a)`	Signum function, $a/\|a\|$ for $a \neq 0$
`sqrt(a)`	$\sqrt{a}$
`st_deviation`	Standard deviation
`sum`	Sum of all elements of a matrix
`union(a,b)`	Extract the unique common elements of `a` and `b`
`unique(a)`	Return the unique elements of `a` in ascending order

Table 4.1: Basic arithmetic functions

Beginning with Version 5.3, Scilab will have only one functions for sorting: `gsort`. Unlike its Matlab counterpart, `gsort` sorts in decreasing order by default. It also behaves differently for matrices. While Matlab sorts each column, Scilab sorts all elements and then stores them columnwise as shown in the example below.

```
-->mat = [-1 4 -2 2;1 0 -3 3]
 mat  =
 - 1.    4.  - 2.    2.
```

```
   1.    0.  - 3.    3.

-->smat = gsort(mat)
 smat  =
   4.    2.    0.  - 2.
   3.    1.  - 1.  - 3.

-->smatc = gsort(mat,'c') // Rows are sorted [21]
smatc =
  4.  2.  - 1.  - 2.
  3.  1.  0.  - 3.
```

In the help file [21] is called a "columnwise" sort; this appears to be somewhat misleading since — as described later in the help file — the rows are the ones that are being sorted. The first column contains the largest element of each row, the second column the second largest, etc. Thus `smatc(:,i)` $\geq$ `smatc(:,j) for i` $<$ `j`.
A third input parameter allows the user to select the sort direction (decreasing or increasing) of `gsort`. In order to get what Matlab's *`sort`* would do one needs to set it to increasing (`'i'`) and also choose "row sorting" (`'r'`).

```
-->smati = gsort(mat,'r','i')  // Matlab-like result
 smati  =
 - 1.    0.  - 3.    2.
   1.    4.  - 2.    3.
```

This way the elements of each column are sorted in increasing order.
Function `gsort` also has an option to perform a lexicographically increasing or decreasing sort. This corresponds to Matlab's *`sortrows`* command and is illustrated below for sorting of rows

```
-->mat1 = [3 4 1 4; 1 2 3 4; 3 3 2 1; 3 3 1 2]
 mat1  =
   3.    4.    1.    4.
   1.    2.    3.    4.
   3.    3.    2.    1.
   3.    3.    1.    2.

//        Lexicographically increasing sorting of rows
-->[smat1,index] = gsort(mat1,'lr','i')
 index  =
   2. !
   4.
   3. !
!   1.
   smat1  =
   1.    2.    3.    4.
   3.    3.    1.    2.
```

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

The first column is sorted first. Rows that have the same element in the first column are sorted by the entries of the second column. If two or more of those are the same as well the entries of the third column are used to determine the order, etc. The optional second output argument gives the sort order (thus `smat1 = mat1(index,:)`).
Changing input argument `'lr'` to `'lc'` changes row sorting to column sorting.
While shown here for numeric arrays, string arrays can be sorted the same way.

4.2.2 Elementary Mathematical Functions

Except for `cotg` the names of all the elementary transcendental functions listed in Table 4.2 agree with those of their Matlab counterparts. Furthermore, `atan` can be called with one or with two arguments. With one argument it equivalent to Matlab's *atan*; with two arguments it corresponds to Matlab's *atan2*: if `x > 0` then `atan(y,x) == atan(y/x)`.
If the argument of any of these functions is a matrix, the function is applied to each entry separately. Thus

```
-->a = [1 2; 3 4]
 a  =
   1.    2.
   3.    4.
-->b = sqrt(a)
 b  =
   1.            1.4142136
   1.7320508     2.          !
-->b.*b
 ans  =
   1.    2.
   3.    4.
```

22a

The functions listed in Table 4.3 are "true" matrix functions; they operate on a matrix as a whole. Thus the matrices have to satisfy certain requirement, the minimum being that they must be square. So, the example above, with same matrix `a` but for `sqrtm`, looks like this

```
-->b = sqrtm(a)
 b  =
    0.5536886 +  0.4643942i     0.8069607 -  0.2124265i
   1.2104411 -  0.3186397i    1.7641297 +  0.1457544i

-->b*b
```

22b

Scilab	Description
acos	Arc cosine
acosh	Inverse hyperbolic cosine
asin	Arc sine
asinh	Inverse hyperbolic sine
atan	Arc tangent
atanh	Inverse hyperbolic tangent
cos	Cosine
cosh	Hyperbolic cosine
cotg	Cotangent
coth	Hyperbolic cotangent
exp	Exponential function
log	Natural logarithm
log10	Base-10 logarithm
log2	Base-2 logarithm
sin	Sine
sinc	Sinc function, $\sin(x)/x$
sinh	Hyperbolic sine
tan	Tangent
tanh	Hyperbolic tangent

Table 4.2: Elementary transcendental functions

```
 ans  =
   1. + 5.551E-17i    2.
   3. + 2.776E-17i    4.

clean(b*b)          23
 ans  =
   1.     2.
   3.     4.
```

Obviously, the matrix b is complex and so rounding errors lead to small imaginary parts of some of the entries in the product b*b. Expression 23 illustrates how function clean can be used to remove such small matrix entries.
The important difference between these two examples is that in 22a corresponding entries of b are multiplied (the . in front of the *) whereas in 22b the matrices are multiplied.
The list of functions in Table 4.3 is longer than it would be in Matlab; on the other hand Scilab lacks an equivalent for Matlab's *funm* function which works for any user-specified functions; for good accuracy, matrices should be symmetric or Hermitian.

Scilab	Description
acoshm	Matrix inverse hyperbolic cosine
acosm	Matrix arc cosine
asinhm	Matrix inverse hyperbolic sine
atanhm	Matrix inverse hyperbolic tangent
atanhm	Matrix inverse hyperbolic tangent
coshm	Matrix hyperbolic cosine
cosm	Matrix cosine
expm	Matrix xponential function
logm	Matrix natural logarithm
sinhm	Matrix hyperbolic sine
sinm	Matrix sine
sqrtm	Matrix square root
tanhm	Matrix hyperbolic tangent
tanm	Matrix tangent

Table 4.3: Matrix functions

4.2.3 Special Functions

Table 4.4 lists so-called special functions of mathematical physics available in Scilab.

Scilab	Description
%asn	Jacobian elliptic function, $\mathrm{sn}(x,m)=\int_0^x dt/\sqrt{(1-t^2)(1-mt^2)}$
%k	Complete elliptic integral, $K(m)=\int_0^1 dt/\sqrt{(1-t^2)(1-mt^2)}$
%sn	Jacobian elliptic function, sn
amell	Jacobian function $\mathrm{am}(u,k)$
besseli	Modified Bessel function of the first kind, $I_\alpha(x)$
besselj	Bessel function of the first kind, $J_\alpha(x)$
besselk	Modified Bessel function of the second kind, $K_\alpha(x)$
bessely	Bessel function of the second kind, $Y_\alpha(x)$
calerf	Compute error functions $\mathrm{erf}(x)$, $\mathrm{erfc}(x)$, $\mathrm{erfcx}(x)$ (see definitions below)
delip	Elliptic integral, $\mathrm{u}(x,k)=\int_0^x dt/\sqrt{(1-t^2)(1-k^2)}$
dlgamma	Digamma function, $\psi(x)=d\ln(\Gamma(x))/dx$
erf	Error function, $\mathrm{erf}(x)=2/\sqrt{\pi}\int_0^x \exp(-t^2)dt$
erfc	Complementary error function, $\mathrm{erfc}(x)=2/\sqrt{\pi}\int_x^\infty \exp(-t^2)dt$
erfcx	Scaled complementary error function, $\mathrm{erfcx}(x)=\exp(x^2)\mathrm{erfc}(x)$
gamma	Gamma function, $\Gamma(x)=\int_0^\infty t^{x-1}\exp(-t)dt$
gammaln	Logarithm of the Gamma function, $\ln(\Gamma(x))$

Table 4.4: Special functions

4.2.4 Linear Algebra

Tables 4.6 and 4.5 list functions for linear-algebra operations. Functions for full matrices work on sparse matrices as well.

Scilab	Description
balanc	Balance matrix to improve condition number
bdiag	Block diagonalization of matrix
bdiag(M)	Block diagonalization/generalized eigenvectors of M
chol(M)	Choleski factorization; R'*R = M
colcomp(M)	Column compression of M
cond	Condition number of M
det	Determinant of a matrix
fullrf(M)	Full-rank factorization of M
fullrfk(M)	Full-rank factorization of M^K
givens	Given's rotation
hess(M)	Hessenberg form of M
householder	Householder orthogonal reflection matrix
inv(M)	Inverse of matrix M
kernel(M)	Nullspace of M
linsolve	Linear-equation solver
norm(M)	Norm of M (matrix or vector)
orth(M)	Orthogonal basis for the range of M
pinv(M)	Pseudoinverse of M
polar(M)	Polar form of M, M=R*expm(%i*Theta)
qr(M)	QR decomposition of M
range(M)	Range of M
rank(M)	Rank of M
rcond(M)	Reciprocal of the condition number of M; L-1 norm
schur(M)	Schur decomposition
spaninter(M,N)	Intersection of the span of M and N
spanplus(M,N)	Span of M and N
spec	Eigenvalues of matrix
sva(M)	Singular-value approximation of M for specified rank
svd(M)	Singular-value decomposition of M
trace(M)	Trace (sum of diagonal elements) of M

Table 4.5: Linear algebra

Scilab	Description
bandwr	Band-width reduction of a sparse matrix
chfact	Sparse Cholesky factorization
chsolve	Use sparse Cholesky factorization to solve linear system of equations
full	Convert sparse to full matrix
lufact	Sparse LU factorization
luget	Sparse LU factorization
lusolve	Solve sparse linear system of equations
nnz	Number of nonzero elements of a sparse matrix
sparse	Create sparse matrix
spchol	Sparse Cholesky factorization
speye	Sparse identity matrix
spget	Retrieve entries of a sparse matrix
spones	Replace non-zero elements in sparse matrix by ones
sprand	Create sparse random matrix
spzeros	Sparse zero matrix

Table 4.6: Functions for sparse matrices

4.2.5 Signal-Processing Functions

Scilab proper and the Signal-Processing Toolbox offer quite a number of functions for signal processing. The functions shown here in Table 4.7 have been chosen because they are frequently used and have Matlab equivalents. Furthermore, the Fast Fourier Transform (FFT) `fft` may need some explanation. After all, it does what *`fft`*, *`ifft`*, *`fft2`*, and *`ifft2`* do in Matlab.

Scilab	Description
convol	Convolution
corr	Convolution
fft	Forward and inverse Fast Fourier Transform
fftshift	Shift zero-frequency component to center of spectrum
mfft	Multidimensional Fast Fourier Transform
nextpow2	For argument x compute smallest integer n such that $2^n \geq x$

Table 4.7: Functions for signal processing

The basic Fourier transform is performed as shown in the example below

```
-->x = rand(100,1);

-->y = fft(x,-1);        24a
```

```
-->z = fft(y,1);            24b

-->norm(x-z)
ans =
1.532E-15
```

where

$$y_m = \sum_{n=1}^{N} x_n \, e^{-2\pi i(n-1)(m-1)/N} \quad \text{for } m = 1, \cdots, N \tag{4.1}$$

with N denoting the number of elements x_n. The second argument, -1, in 24a corresponds to the minus sign in front of the exponent in (4.1). The operation performed in 24b,

$$z_m = \frac{1}{N} \sum_{n=1}^{N} y_n \, e^{2\pi i(n-1)(m-1)/N} \quad \text{for } m = 1, \cdots, N,$$

is the inverse of 24a.

If `xx` is a matrix then `f(xx,-1)` performs the two-dimensional Fourier transform. It is thus equivalent to Matlab's *fft2*. Matlab's *fft*, on the other hand, performs a one-dimensional FFT on each column of a matrix. In order to achieve the same result with Scilab one has to write `fft` in the form shown in line 25 below.

```
-->n = 100; m = 20;

-->xx = rand(n,m);

-->yy1 = zeros(xx);

-->for i=1:m
-->  yy1(:,i) = fft(xx(:,i),-1);
-->end

-->yy2 = fft(xx,-1,n,1);           25

-->norm(yy1-yy2)                   26
ans =
0.

-->zz = fft(yy2,1,n,1);            27

-->norm(xx-zz)                     28
ans =
3.368E-15
```

Expression 26 shows that yy1 and yy2 are identical. Likewise, expression 28 shows that the inverse Fourier transform 27 works as expected with this syntax.
Furthermore, with yy2 computed in 25, statement 29 computes the two-dimensional FFT of xx:

```
-->uu1 = fft(xx,-1);    // Two-dimensional FFT of xx

-->uu2 = fft(yy2,-1,m,n);        29

-->norm(uu1-uu2)
 ans =
0.
```

Obviously, uu1 and uu2 are identically.

4.3 File Input and Output

There are quite a few functions for formatted and unformatted reading and writing of text and numeric data. Some have Matlab equivalents. They are summarized in tables 4.9 (reading), 4.10 (writing), and 4.8 (ancillary functions). Many I/O functions come in pairs — one is designed to read what the other one writes. The special-purpose routines for, say, writing and reading audio files fall into this category. Some of the following pairs represent my own way of grouping. This grouping does not imply that no other function can read what one of these functions writes and vice versa; rather, these pairs appear similar in terms of design philosophy and input arguments.

4.3.1 Opening and Closing of Files

Before one can read from, or write to, a file the file needs to be "opened". This is transparent for I/O functions, such as fprintfMat or fscanfMat, that only use a file name to specify which file to read from (write to). They open the requested file, read/write the data and close the file without the user being aware of it. But whenever there is a need to incrementally read or write data it is up to the user to open (and, eventually, close) files. A situation like that occurs, for example, with big data sets. One might read a piece of the data from file A, process it, and write it out to file B; then read in the next piece of data from file A, process it, and write it to file B, etc. In this case files A and B must be opened before anything can be read from respectively written to them. Scilab has two functions for opening a file, mopen and file, and Scilab functions that allow incremental I/O require one or the other. For this reason the subsequent discussion of specific I/O functions mentions, where appropriate, which one of the two functions needs to be used for opening a file. Function mopen is quite similar to Matlab's *fopen* whereas file reminds one of the Fortran equivalent.
Functions mopen and file output a file identifier (Matlab terminology). Scilab help files call it "file descriptor" or "logical unit descriptor"; in Fortran it is called "Logical Unit Number". It is this file identifier, and not the file name, that is then used to specify from which file to read (to which file to write). File identifiers are numbers which range from 1 to 19 in Scilab. File identifier 1 is used for the history file scilab.hist, file identifiers 5 and 6 (%io(1) and %io(2), respectively) are reserved for keyboard (input) and Scilab window (output), respectively. Hence, a maximum of

16 file identifiers are available to users; this limits to 16 the number of user files that can be open at any one time.
Files that have been opened with `mopen` must be closed with `mclose`, and `file` with the `'close'` option must be used to close files that have been opened with `file`. Examples of the use of `mopen`, `mclose`, and `file` are part of the discussion of specific I/O functions below.

Scilab	Description
`basename`	Strip directory and file extension from a file name
`dirname`	Get the directory from a filename
`dispfiles`	Display properties of opened files
`file`	Open/close a file, define file attributes
`fileinfo`	Get information about a file
`getio`	Get Scilab's standard logical input/output units
`isdir(a)`	Test if directory `a` exists
`listfiles`	Output string vector with names of files matching a pattern
`mclearerr`	Reset binary-file access errors
`mclose`	Close (all) open file(s)
`meof`	Check if end-of-file has been reached
`mopen`	Open a file
`mseek`	Set position in a binary file
`mtell`	Output the current position in a binary file
`newest`	Find newest of a set of files
`pathconvert`	Convert file path from Unix to Windows and vice versa
`uigetfile`	Open dialog box for file selection

Table 4.8: Functions that manipulate file names and open, query, and close files

4.3.2 Functions `mgetl` and `mputl`

Function `mputl` writes a vector of strings to an ASCII file in form of a sequence of lines, and `mgetl` can retrieve one or more of these lines. This is a simple example:

```
-->text = ['This is line 1';'Line 2 ';'Line 3 (last)']
 text  =
!This is line 1  !
!                !
!Line 2          !
!                !
!Line 3 (last)   !
-->mputl(text,'C:\temp\dummy.txt')
-->
-->all = mgetl('C:\temp\dummy.txt') // Get the whole file
```

```
 all  =
!This is line 1  !
!                !
!Line 2          !
!                !
!Line 3 (last)   !
```

With only one input argument, `mgetl` reads the whole file. If only the first few lines are required the number of lines can be specified via the second input parameter:

Scilab	Description
`auread`	Read a .au audio file from disk
`excel2sci`	Read ASCII file created by MS Excel
`fscanf`	Read numeric/string variables from ASCII file under format control
`fscanfMat`	Read matrix from ASCII file
`input`	Read from keyboard with a prompt message to Scilab window
`load`	Load variables previously saved with `save`
`loadmatfile`	Load variables previously saved in Matlab-readable format
`loadwave`	Read a .wav sound file
`mfscanf`	Read data from file (C-type format)
`mget`	Read numeric data (vector) from binary file (conversion format)
`mgeti`	Read data from binary file, converts to Scilab integer format
`mgetl`	Read a specified number of lines from ASCII file
`mgetstr`	Read bytes from binary or ASCII file and interpret as character string
`mscanf`	Read data from keyboard (C-type format)
`read`	Read matrix of strings/numbers from ASCII file under format control
`read4b`	Read Fortran binary file (4 bytes/word)
`readb`	Read Fortran binary file (8 byte/word)
`readc_`	Read a character string from a file/keyboard
`wavread`	Read a .wav sound file

Table 4.9: Functions that input data from files or the keyboard

```
-->only2 = mgetl('C:\temp\dummy.txt',2)  //  Read first 2 lines only
 only2  =
!This is line 1  !
!                !
!Line 2          !
```

If more lines are requested than are available, the function aborts with an error message. If the second argument is -1, all lines are read (equivalent to no second input argument).

Scilab	Description
auwrite	Write a .au audio file to disk
diary	Write screen output of a Scilab session to a file
disp	Write input argument to Scilab window
fprintf	Write formatted data to file (like C-language fprintf function)
fprintfMat	Write matrix to ASCII file under format control
mfprintf	Write data to ASCII file (C-type format)
mprintf	Writes data to Scilab window (C-type format)
mput	Write numeric data to file in user-specified binary representation
mputl	Write string vector to ASCII file (one line per vector element)
mputstr	Write character string to an ASCII file
print	Print variables to file in the format used for Scilab window
printf	Print to Scilab window (emulation of C-language printf function)
save	Write current Scilab variables to a binary file
savematfile	Write current Scilab variables to a Matlab-readable file
savewave	Write a .wav sound file
wavwrite	Write a .wav sound file
writb	Write matrix in to a Fortran binary file (4 bytes/word)
write	Write matrix of strings/numbers to ASCI file (Fortran-type format)
write4b	Write matrix in to a Fortran binary file (8 bytes/word)

Table 4.10: Functions that output data to files or to the Scilab window

In the examples above, the file to use is identified by its name. In a case like this `mgetl` does three things. It opens the file for reading, reads the lines requested, and closes the file again. This convenience comes at a price. It is not possible to read a file a few lines at a time. If this is necessary one must open the file oneself and use the file identifier created by `mopen` to "tell" `mgetl` from which file to read. Finally, once the file has been read, one needs to close it again.

```
fid = mopen('C:\temp\dummy.txt','r')     // Open file for reading
 fid  =
    3.
-->one = mgetl(fid,1)              // Read one line
 one  =
 This is line 1
-->twomore = mgetl(fid,2)          // Read two more lines
 twomore  =
!Line 2         !
!               !
!Line 3 (last)  !
-->mclose(fid)                     //  Close the file
 ans  =
    0.
```

An analogous procedure can be used to write a file one line (or several lines) at a time.

It is important to note that mputl puts each string in a string matrix in a separate line. Thus a string matrix with more than one column — when read in — will become a one-column matrix. This is illustrated in the next example. 30a

```
-->textlines = ['This is line 1a','Line 1b';
-->             'This is line 2a','Line 2b']
 textlines  =
!This is line 1a  Line 1b  !
!                          !
!This is line 2a  Line 2b  !

-->mputl(textlines,'C:\temp\dummy.txt')

-->allnow = mgetl('C:\temp\dummy.txt')   // Get the whole file
 allnow  =
!This is line 1a  !
!                 !
!This is line 2a  !
!                 !
!Line 1b          !
!                 !
!Line 2b          !
```

The function matrix can be used to reshape (no pun on Matlab intended) allnow into the original 2 by 2 string matrix.

```
-->matrix(allnow,2,2)    31
 ans  =
! This is line 1a   Line 1b  !
!                            !
! This is line 2a   Line 2b  !
```

Similar to Matlab's *reshape*, only one of the dimensions of matrix needs to be given; the other can be replaced by -1.[1] The parameter not specified is computed from the dimension of the matrix to be reshaped. Thus statement 31 is equivalent to either of the two statements

```
matrix(allnow,-1,2)
matrix(allnow,2,-1)
```

4.3.3 Functions read and write

Functions write and read do what mputl and mgetl do — and more. The following statements are equivalent to those in 30a above.

```
-->textlines = ['This is line 1a','Line 1b';
-->             'This is line 2a','Line 2b']
```

[1]With Matlab it is the empty matrix *[]*.

```
 textlines  =
!This is line 1a  Line 1b  !
!                          !
!This is line 2a  Line 2b  !
-->write('C:\temp\dummy.txt',textlines)

-->all = read('C:\temp\dummy.txt',-1,1,'(A)')  // Get the whole file
 all  =
! This is line 1a  !
!                  !
! This is line 2a  !
!                  !
! Line 1b          !
!                  !
! Line 2b          !
```

While the write statements only needs the file name and the data the read statement also wants the size of the array to read and a format in FORTRAN syntax. The dimension are in input arguments 2 and 3, the -1 simply instructs `read` to read the whole file; in this example it could have been replaced by 4 since there are 4 strings in the file. Like `mputl` function `write` writes a string array one column to a line.

Functions `read` and `write`, when used with a file name as first argument, open the file and close it again after the I/O operation. To read or write incrementally, one needs to open the file oneself. However, this cannot be done with function `mopen` used above. Rather, the file must be opened (and closed) with function `file`. This is illustrated in the example below where the file created above is read again.

```
-->fid = file('open','C:\temp\dummy.txt','unknown') // Open file
 fid  =
    4.

-->one = read(fid,1,1,'(A)')                    // Read one line
 one  =
  This is line 1a

-->twomore = read(fid,2,1,'(A)')                // Read two more lines
 twomore  =
! This is line 2a  !
!                  !
! Line 1b          !

-->file('close',fid)                            // Close the file
```

Function `file` above opens the file `C:\temp\dummy.txt` for read and write access. By default the file is a sequential-access file for ASCII data. Other file types can be chosen by setting the appropriate input parameters.

Sequentially writing to a file is completely analogous.
Functions `read` and `write` can also be used to read and write numeric data.

```
-->fid = file('open','C:\temp\numeric.txt','unknown'); // Open file

-->a = rand(3,5,'normal')
 a  =
 -  0.7460990     0.1023021  -  0.3778182  -  0.6453261    1.748736
 - 1.721103    - 1.2858605    2.5749104     0.0116391    0.1645912
 - 1.7157583     0.6107784  -  0.4575284  - 1.4344473    0.9182207

-->write(fid,a)

-->file('close',fid)                            // Close the file
```

The 3 by 5 matrix `a` is written to file in ASCII format (as a string) and unformatted and can be retrieved as shown below.

```
-->[fid,ierr] = file('open','C:\temp\numeric.txt','old') // Open file
 ierr  =
    0.
 fid  =
    4.

-->if ierr ~= 0 then error(' Problem opening file'), end

-->newa = read(fid,2,3)
 newa  =
  - 0.7460990     0.1023021  -  0.3778182
 - 1.721103    - 1.2858605    2.5749104

-->file('close',fid)                            // Close the file
```

Function `file` is used here with two output arguments; the second provides the error status. If an error occurs while opening a file function `file` does not abort but rather saves the error number in this second output argument and leaves it to the user to handle the error. Function `read` only requests the first two columns of the first two rows, and that is what is output. Furthermore, the file status is set to `'old'`. After all, the file must already exist in order to be read. Of course, `'unknown'` would have been an option too.
The next example shows how `a` can be written to a file under format control. It also shows that the "file" can be the Scilab window — as mentioned earlier, %io(2) is the file identifier for the Scilab window.

```
-->write(%io(2),a,'(5f10.5)')
    0.51633    0.64507    -.54852  -1.38505  -1.10499
   1.04225    -.44840    1.13162  -1.62805    0.76045
   2.49761    -.72190   -1.36674    0.77577    -.65881
```

Matrix a is written to the file in 5 columns, each of which is 10 characters wide, with 5 digits to the right of the decimal point. Incidentally, write can also be used to write a string vector (but not a matrix) to the Scilab window without the “almost blank” lines.

```
textlines(:)
 ans  =
!This is line 1a  !
!                 !
!This is line 2a  !
!                 !
!Line 1b          !
!                 !
!Line 2b          !

-->write(%io(2),textlines)
This is line 1a
This is line 2a
Line 1b
Line 2b

-->write(%io(2),textlines,'(a20)')
     This is line 1a
     This is line 2a
             Line 1b
             Line 2b
```

Without a format the strings are left-justified. With format a20 they are right justified; the total number of characters per line is 20.

4.3.4 Functions load and save

Functions save and load perform the same function they perform in Matlab: save writes one, or more, or even all variables of the workspace to a file. The file can be defined either by its name (in this case opening and closing is done automatically) or by a file identifier. In the latter case the file needs to be opened with mopen with parameter wb (write binary). But variables can be saved incrementally to the same file. An example is below.

```
-->a = 3;

-->fid = mopen('C:\Temp\saved.bin','wb');

-->save(fid,a)

-->b = 5; c = 'text';

-->save(fid,b,c)
```

```
-->mclose(fid);
```

Note that variable names in the save command are not in quotes. In Matlab they would be. To recall variables saved earlier, possibly in another session,

```
-->clear a, clear b

-->load('C:\Temp\saved.bin','a','b')

-->a,b
 a  =
    3.
 b  =
    5.
```

Here, as in Matlab's *load* function, the variable names must be in quotes.

4.3.5 Functions loadmatfile and savematfile

Function savematfile and loadmatfile are quite analogous to functions save and load, respectively. The difference is the format they use for saving variables. Function savematfile saves one, several, or all workspace variables to a file that Matlab can read. The Matlab format is version-specific, with later versions also supporting earlier formats. Scilab supports '-v4', '-v6', '-v7', and '-v7.3'. Function loadmatfile reads the variables from a file in Matlab formats '-v4', '-v6', '-v7', and '-v7.3'. Hence, exchange of data between Matlab and Scilab is quite simple.

4.3.6 Functions mput and mget/mgeti

The two input functions allow one to read blocks of 1, 2, 4, or 8 bytes from a binary file and convert them into either double-precision floating point numbers (mget) or into integers (mgeti) (see rightmost column of the table below). The type of conversion is controlled by a type parameter which can take the following values

Type	in file	in Scilab
c	8-bit integer	int8
s	16-bit integer	int16
i	32-bit integer	int32
l	64-bit integer	double
uc	Unsigned 8-bit integer	uint8
us	Unsigned 16-bit integer	uint16
ui	Unsigned 32-bit integer	uint32
ul	Unsigned 64-bit integer	double
f	32-bit floating-point number	double
d	64-bit floating-point number	double

The functions can incrementally read/write files that have been opened with mopen.

With binary files the questions of byte ordering has to be addressed. Intel CPU's, and thus PC's, use "little-endian" byte ordering whereas so-called workstations (Sun Sparc, SGI) use "big-endian" byte ordering. In Matlab byte ordering is specified when a file is opened with *fopen*. Function mopen in Scilab has no such option; instead, byte ordering is specified together with the variable type by appending a b or l to the type parameter. Thus the statement for reading 6 big-endian, 32-bit integers from a file with file identifier fid is

```
-->from_file = mget(6,'ib',fid)
```

Had the b been omitted the "natural" byte ordering of the computer on which the program runs would have been used (e.g. little-endian for a PC). As long as one reads files written on the same type of computer, byte ordering is generally not a problem. It needs attention when one reads a file created on a computer with different byte ordering.

4.3.7 Functions input and disp

Function input is completely equivalent to Matlab's *input*:

```
-->response = input('Prompt user for input')
Prompt user for input-->3
 response  =
    3.
```

The user response (3 in this example) can also be an expression involving variables in the workspace. Furthermore, by adding a second argument, 'string' or simply 's', the user's response can be interpreted as a string. There is no need to put it in quotes.
Function disp has a close Matlab analog as well. Unlike its Matlab counterpart it can take more than one argument. However, as illustrated out earlier (page 14) the arguments are displayed in reverse order.

4.3.8 Function uigetfile

Function uigetfile opens a dialog box for interactive file selection. It works essentially like Matlab's *uigetfile*. An example is

```
-->file_name = uigetfile(filemask='*.sgy',dir='D:\Data\Seismic', ...
          title='Read SEG-Y file');
```

which opens a file selection window with the title "Read SEG-Y file". The initial directory shown in the window is D:\Data\Seismic, and only files with file name extension .sgy are shown initially.

4.4 Utility Functions

This chapter describes some of the functions that are not directly necessary to run or debug Scilab functions, that are more peripheral to Scilab and may not fit well in any other topic discussed previously. Table 4.11 shows the functions I chose to put into this category.

The LaTeX code that function prettyprint generates requires the amsmath package (add \usepackage{amsmath} to the preamble):

```
-->mat = [.2113249 .3303271 .8497452 .0683740 .7263507;
        0.7560439 .6653811 .6857310 .5608486 .1985144;
        0.0002211 .6283918 .8782165 .6623569 .5442573];

-->prettyprint(mat)
 ans  =
${\begin{pmatrix}
0.2113249 &0.3303271 &0.8497452 &0.068374 &0.7263507 \cr
0.7560439 &0.6653811 &0.685731 &0.5608486 &0.1985144 \cr
0.0002211 &0.6283918 &0.8782165 &0.6623569 &0.5442573 \cr
\end{pmatrix}}$
```

Scilab	Description
basename	Strip directory and file extension from a file name
diary	Write screen output of a Scilab session to a file
dirname	Get the directory from a filename
findfiles	Find all files in a given directory (containing specific characters)
fun2string	Output string vector with a function's source code
getenv	Get value of an environment variable
getversion	Display version of Scilabversion of Scilab
host	Execute Unix/DOS command; outputs error code
lines	Specify number of lines to display and columns/line
listfiles	Output string vector with names of files matching a pattern
pathconvert	Convert file path from Unix to Windows and vice versa
stacksize	Determine/set the size of the stack
prettyprint	LaTeX representation of a Scilab object
tic	Start timer
timer	Output CPU time used since the preceding call to timer()
toc	Output elapsed time since the call to tic
unix	Execute Unix/DOS command; outputs error code
unix_g	Execute Unix/DOS command; output to variable
unix_s	Execute Unix/DOS command; no output (silent)
unix_w	Execute Unix/DOS command; output to Scilab window
unix_x	Execute Unix/DOS command; output to a new window

Table 4.11: Utility functions

The diary function causes a copy of all subsequent keyboard input and the resulting Scilab output to be copied to the file named in the argument. It now offers more flexibility than Matlab's *diary*,

which only creates a new file if the named file does not exist. Otherwise, it appends the output to the existing file. Also, diary recording can be turned off and on.
Quite a few functions are available to execute UNIX or DOS commands from the Scilab command line. Those accustomed to the ways of Matlab will not be surprised to use a function that begins with the four letters `unix` to execute DOS commands. The choice among the various functions beginning with `unix` depends on the desired output which is indicated in Table 4.11. Functions `host` and `unix` are interchangeable.
Since operating system commands are generally different for MS-DOS and UNIX, a function that is expected to run on both may need to have different branches for the two. Function `getos` can be used to determine the type of operating system.

```
select getos()
case 'Windows'
  files=unix_g('dir D:\MyScilab\Experimental\*.sci /B');  32a
case 'Unix'
  files=unix_g('ls -l ~/MyScilab/Experimental\*.sci');
else
  error('Unknown operating system')
end

write(%io(2),files)
s_test.sci
atest.sci
clean.sci
s_wplot.sci
```

Incidentally, statement 32a is equivalent to statement 32b below in that it also produces a string vector with the names of the files in a particular directory. The latter uses the function `listfiles` in combination with `basename`.

```
-->files=basename(listfiles('D:\MyScilab\Experimental\*.sci'))+'.sci';  32b

-->write(%io(2),files)
s_test.sci
atest.sci
clean.sci
s_wplot.sci
```

Function `listfiles` returns the files names including the directory path; `basename` strips off not only the path but also the file extension `.sci` which is then appended again.
The advantage of 32b over 32a is that, with the correct path, 32b can be used for either UNIX or Windows/MS-DOS.
Function `findfiles`, on the other hand, creates a string matrix with all the files in a director. By adding a search string as a second argument on can reduce the number of entries in the string matrix to those that include that string.

```
-->findfiles(SCI,'*exe')
 ans =
unins000.exe
```

Since the source code of Scilab is freely available it is, in principle, possible to inspect every function. However, this may require more effort than one is willing to expend. Fortunately, for macros, one can achieve the same objective with function fun2string. This function regenerates, from the pseudo-code of a compiled Scilab function, the original source code—essentially converting a *.bin function into a *.sci function. Thus, many of Scilab's functions (but not primitives, i.e. built-in functions) can be reviewed and possibly modified for a particular purpose. The following example shows the source code of help[2].

```
-->fct = fun2string(help);

-->show(fct)
function []=ans(key, flag)

change_old_man();
INDEX = make_index();
[lhs, rhs] = argn(0);

if rhs == 0 then
  browsehelp(INDEX, 'index');
  return,
end,

if rhs > 2 then   error(39);return,end,
if rhs == 2 then
  help_apropos(key);
  return,
end,

path = gethelpfile(key);
if path ~= [] then
  browsehelp(path, key);
else
  help_apropos(key);
end
endfunction
```

Since comments are dropped in the compile stage, they cannot be recovered, and thus the reconstructed source code will have empty lines where comments used to be.

[2] Actually, in Version 5.2, this particular example aborts with an error message. It does work in prior versions of Scilab.

Chapter 5

Scripts

A script is a sequence of Scilab commands stored in a file (while a script file may have any extension, the Scilab Group suggests the extension `.sce`). Scripts have neither input arguments nor output arguments. Since they do not create a new level of workspace all variables they create are available once the execution of the script is completed.
To invoke a script in Matlab its name without extension, say *script_file*, is typed on the command line. Furthermore, the file can be in any directory of the search path. Scilab, on the other hand, uses the concept of a working directory familiar from Unix. The command `pwd` (Print Working Directory) or the environmental variable `PWD` can be used to find out what it is. If a script, say `script_file.sce`, is in the working directory it can be executed by the command

```
-->exec('script_file.sce')
```
33a

A Scilab script can also be stored as a vector of strings; in this case it is executed by means of function `execstr`.
The function `exec` has two optional arguments: the string `'errcatch'` and the variable `mode`; the former allows a user to handle errors during execution of the script, the latter allows one to control the amount of output. It does not appear to really do what the documentation says. The following table is taken from the help file:

Value	Meaning
0	the default value
-1	print nothing
1	echo each command line
2	print prompt $--->$
3	echo + prompt
4	stop before each prompt
7	stop + prompt + echo : useful mode for demos

The following are several examples of the mode parameters. Let `test.sce` be the following script

```
// Script to illustrate the mode parameter
a = 1
b = a+3;
disp('mode = '+string(mode()))
```

Then, without setting the mode parameter, i.e.
mode not specified:

```
-->exec('D:\MyScilab\test.sce')

-->// Script to illustrate the mode parameter

-->a=1
 a  =

    1.

-->b=a+3;

-->disp('mode = '+string(mode()))

 mode = 3

-->disp('mode is '+string(mode()))
 mode is 2
```

In this case `exec` echoes every line of the script and displays the results of every statement that has no terminating semicolon. Here and in the following examples spaces between lines output by `test` have been preserved to more accurately reflect the output of the script. Obviously, the default is for `exec` to set `mode` to 3. But once `exec` has run `mode` reverts to 2.
mode = 0:

```
-->exec('D:\MyScilab\test.sce',0)
 a  =

    1.

 mode = 0
```

This value of mode produces the result one would expect from Matlab.
mode = 1:

```
-->exec('D:\MyScilab\test.sce',1)
-->// Script to illustrate the mode parameter
-->a = 1
 a  =

    1.
-->b = a+3;
-->disp('mode = '+string(mode()))

 mode = 1
```

This is the same information displayed with `mode = 0` but in a somewhat more compact form (fewer blank lines).
mode = -1:

```
-->exec('D:\MyScilab\test.sce',-1)

 mode = -1
```

In this case the result of expressions is not displayed even if they are not terminated by a semicolon.
mode = 2:
Displays the same information as `mode = 0` but with more empty lines.
mode = 3:
Default mode (mode parameter not given).
mode = 4:
Prints
`step-by-step mode: enter carriage return to proceed`
but then behaves like `mode = 0` after all.
mode = 7:

```
-->exec('D:\MyScilab\test.sce',7)
step-by-step mode: enter carriage return to proceed
>>
-->// Script to explain mode parameter
>>
-->a = 1
 a  =


    1.
>>
-->b = a+3;
>> -->>disp('mode = '+string(mode()))

 mode = 7
```

This mode works as advertised. It prompts the user to press the <ENTER> key after each statement.
Function `exec` satisfies the requirements of the command-style syntax (see Section 4.1). Thus 33a and 33b, 33c below are equivalent statements.

```
-->exec 'script_file.sce'
```
33b

and

```
-->exec script_file.sce
```
33c

Furthermore, for all three variants, a trailing semicolon will suppress echoing the commands `exec` executes.

As in Matlab, Scilab scripts can include function definitions. However, Scilab is more flexible in the way functions can be defined within a script (or within another function). This is explained below in Section 6.2.
If a file with a Scilab script is not in the working directory then either the working directory needs to be changed (with function chdir) or the full filename of the script must be given. Scilab does not provide for a search path the way Matlab does or the way Unix provides for executables.
A bare-bones simulation of a search path for the execution of a Scilab script is afforded by the following function.

```
function myexec(filename,mod)
// Function emulates use of a search path for the execution
// of a script.
//
// INPUT
// filename  filename of the script to be executed; the
//           extension .sce is added if the file name
//           of the script has no extension
// mod       mode parameter(determines amount of printout;
//           see help file for exec); default: mod = 1.
//

// Path to the directories to search for script ''filename''
path=['D:\MyScilab\Experimental\', ...
      'D:\MyScilab\tests\', ...
      'D:\MyScilab\General\', ...
      'D:\MyScilab\Sci_files\'];

//      Test the filename with directories in path
for ii=1:size(path,'*');
   testfile=path(ii)+filename;
   [fid,ierr]=file('open',testfile,'old');
   if ierr == 0
      file('close',fid)
      disp(' .... now in ""myexec"" executing '+testfile)

//      Write file to a temporary location and execute it from there
//      so that it does not prevent the original file from being edited
      %tempfile=pathconvert(TMPDIR)+filename;   34

      select getos()
      case 'Windows'
         unix_s('copy '+testfile+' '+%tempfile)
      else
         unix_s('cp '+testfile+' '+%tempfile)
```

```
      end

      oldvars=who('local');   // Variables prior to execution of script

      exec(%tempfile,mod);    // Execute the script

//        Delete the temporary file once the script has been executed;
//        the appropriate statement depends on the operating system used
      select getos()
      case 'Windows'
         unix_s('del '+%tempfile)
      else
         unix_s('rm '+%tempfile)
      end

//      Return variables created in script to the calling environment
//      level from which myexec was called
      newvars=who('local');   // Variables after execution of script
      newvars=newvars(1:size(newvars,'*')-size(oldvars,'*')-1);
      if newvars == []
         return
      else
         str1=strcat(newvars,',');
//      Return to workspace from which "myexec" was called
         execstr('['+str1+']=return('+str1+')')  35
      end
   end
   if ierr ~= 240
      break
   end
end

if ierr == 240      // File not found in any of the directories
   write(%io(2),'File '+filename+ ' not found. Directories searched:')
   write(%io(2), '    '+path)
end

endfunction
```

The four directories of the search path are defined in the string vector `path`. One directory after the other is concatenated with the file name `filename` of the script to be executed. If a file by this name does not exist in the directory then `file` aborts with error 240 (File filename does not exist or read access denied) and the next directory is tried. If `file` is successful the file is closed

again and exec is executed with the file in that directory. In the next step any variables that have been created by the script are returned to the calling program.
If the file is in none of the directories the function prints an error message, lists the directories in the search path, and terminates.
Line $\boxed{34}$ illustrate one way Scilab can handle differences between Windows and Unix. Function pathconvert converts a directory path to the appropriate form for the type of operating system under which it is running.

```
-->SCI
 SCI  =
 D:/Science Programs/Scilab-cvs-02-13-2003

-->pathconvert(SCI)
 ans  =
 D:\Science Programs\Scilab-cvs-02-13-2003\

-->pathconvert(SCI+'/')
 ans  =
 D:\Science Programs\Scilab-cvs-02-13-2003\
```

Furthermore, it checks if the path ends with a slash ("/")—or a backslash ("\") for Windows—and appends one if it does not.
Improving compatibility with Matlab, function filesep is also available; like Matlab's *filesep*, it outputs the character that separates directory/folder names in filenames, i.e. a backslash under Windows and a slash under Unix.

Chapter 6

User Functions

While functions in Scilab are variables and not files they have many features in common with those in Matlab. They consist of a function head and the function body. The function head has the form

```
function [out1,out2,···] = function_name(in1,in2,···)
```

familiar from Matlab. The ellipses `...` indicate that the number of input and output arguments is arbitrary. A function may have no input and/or no output arguments. For functions with one output argument, the brackets are optional. For functions with no input arguments the parentheses are optional when it is defined, but not when it is called.Only white spaces and comments are allowed after the closing parenthesis. The function head can extend over more than one line with the usual continuation indicator (`..`).

The function body consists of a number of Scilab statements. Functions can be either in separate files (one or more functions per file and the name of the file is not necessarily related to the names of the functions) or they can be created within scripts or other functions (in-line functions). Functions can be used recursively, i.e. a function can call itself.

An example of a simple function is

```
function [r,phi] = polcoord(x,y)
  // Function computes polar coordinates from cartesian coordinates
  r = sqrt(x^2+y^2);
  phi = atan(y,x)*180/%pi;
endfunction
```

This function looks much like a Matlab function except for:

- the two slashes preceding the comment;
- the use of the function `atan` rather than its Matlab equivalent *`atan2`*;
- the use of special constant `%pi` rather than its Matlab equivalent *`pi`*;
- the use of `endfunction`.

Scilab	Description
abort	Interrupts current evaluation and return to prompt
argn	Number of imput/output arguments of a function
endfunction	Indicate end of function
error	Print error message and abort
function	Identify header of function definition
getos	Output string(s) with name/version of operating system
halt	Stop execution and wait for a key press
macrovar	Provides names of variables used, and functions called, in user function
mode	Control amount of information displayed by function/script
pause	Interrupt execution of function or script
plotprofile	Create graphic display of execution profile of a Scilab function
profile	Extract execution profiles from a Scilab function
resume	Return from a function or resume execution after a pause
return	Return from a function or resume execution after a pause
showprofile	Display execution profiles of a Scilab function
varargin	Variable number of input arguments for a function
varargout	Variable number of output arguments for a function
warning	Print warning message
where	Output current instruction calling tree to variable
whereami	Display current instruction calling tree
whereis	Display name of library containing a specific function

Table 6.1: Functions/commands/keywords relevant for user functions

The following example computes the Chebyshev polynomial $T_n(x)$ by means of the recurrence relation

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

to illustrate the recursive use of functions (functions calling themselves).

```
function ch = cheby(x,n)
  // Compute Chebyshev polynomial of order n for argument x
  if n == 0
    ch = 1;
  elseif n == 1
    ch = x;
  else
    ch = 2*x*cheby(x,n-1)-cheby(x,n-2);
  end
endfunction
```

In Scilab, variables can be passed to functions in three different ways:

- as a variable in the input argument list

- as a global variable
- as a variable not local to the function, i.e. a variable that is not initially defined in the function

The first two ways of input and output are also used by Matlab. The third one is not. It essentially means that any variable defined in the calling workspace of a function is available to that function as long as it is not defined there. A variable defined in the calling workspace that is also defined in the called function is called "shadowed". The following function illustrates this point. Even if the statement `endfunction` were omitted the function

```
function y = func1(x)   36a
  a = (a+1)^2
  y = x+a;
endfunction
```

would not work in Matlab since the variable `a` is not defined prior to its first use in `func1`. In Scilab the following code fragment works:

```
-->a = 1;   37a

-->y = func1(3)
 y  =
    7.

-->disp(a)
    1.
```

Since the variable `a` (set to 1) is available in the calling workspace, it is also available in `func1`. The new value of `a` created in `func1` (`a` is changed to 4) is not passed on to the calling workspace. This approach works across an arbitrary number of levels. Assume `funcB(x)`, which uses a variable `a` without first defining it, is called by `funcA(x)` which does not use a variable `a`. Then

```
a = 5; funcA(10);
```

still works: `a` in `func2B` is taken to be 5 since `a` is part of the calling workspace not only of `funcA` but also of `funcB`. So one might wonder about the purpose of the `global` statement if variables are passed to functions even if they are not in the argument list or defined as global. The answer is simply that the global statement allows one to "export" variables from a function. Thus changing line $\boxed{37a}$ by defining `a` as global has no effect on the result

```
-->global a,  a = 1;   37b

-->y = func1(3)
 y  =
    7.

-->disp(a)
    1.
```

However, if function func1 36a is changed to func1g which also includes a global statement

```
function y = func1g(x)   36b
  global a
  a = (a+1)^2
  y = x+a;
endfunction
```

then

```
-->global a   37c

-->a = 1;

-->y = func1g(3)
 y  =
    7.

-->disp(a)
    4.
```

The variable a at the end of code fragment 37c is 4, the value computed in func1g. If the global a is dropped from 37c then

```
-->a = 1;   37d
-->y = func1g(3)
 y  =
    7.
-->disp(a)
    1.
```

Thus 37a, which uses func1, and 37d, which uses func1g, leave the variable a unchanged in the calling program where it is not defined as global.

Scilab — like Matlab — has variable-length input argument and output argument lists. They even have the same names, varargin and varargout, and work the same way[1]. If specified together with regular (positional) arguments, they must be last in the argument list. An example is

```
function sizes(varargin)
  // Arguments must be numeric or string matrices
  for ii = 1:length(varargin)
    [nrows,ncols] = size(varargin(ii));
    disp('Input argument no '+string(ii)+' has '+string(nrows)...
         +' row(s) and '+string(ncols)+' column(s)')
  end
endfunction
```

[1] In Scilab varargin and varargout are lists whereas they are cell vectors in Matlab.

which can be called with any number of input arguments and prints the number of rows and columns for each input argument (provided the input arguments are not lists and the like for which `size` has fewer than 2 or more than 2 output arguments). Thus

```
-->sizes(1:10,'test',['a','ab';'abc','abcd'])
 Input argument no 1 has 1 row(s) and 10 column(s)
 Input argument no 2 has 1 row(s) and 1 column(s)
 Input argument no 3 has 2 row(s) and 2 column(s)
```

The number (`in`) of actually defined input arguments and the number (`out`) of output arguments of a function is provided by function `argn` as follows

```
[out [,in] ]=argn()
out=argn(1)
in=argn(2)
```

This function does what *`nargin`* and *`nargout`* do in Matlab. However, there is a slight twist. It is not possible to determine if a function has been called without an explicit output argument since there is always the implied output argument `ans`. Thus `argn(1)` will never be 0.
Scilab functions can return variables to the calling program in three different ways as well:

- as a variable in the output argument list
- as a global variable
- as the argument of the `resume` or `return` command

The first two are familiar from Matlab; furthermore, the above discussion of global variables has also touched on the role of global variables as means to output data from a function. So it is only the third item that needs an explanation.
In order to explain how the third way of returning parameters works it is necessary refer to a difference between the Matlab *`keyboard`* command and the Scilab `pause` command. Both commands interrupt the execution of a function or script. In Matlab one ends up in the workspace of the interrupted function (or script). Any variable created in this workspace is available to the interrupted function once execution resumes. Scilab, on the other hand, creates a new workspace. As with functions, all variables defined in the lower workspaces are available. But, upon return to the workspace below (Scilab command `resume`, all newly created variables (or any modifications of variables of the higher workspaces) are not available to this lower workspace. This is discussed in more detail on pages 12 ff.
Before I go on to the next section it is appropriate to shed some light on this section's opening statement that functions in Scilab are variables. The following sequence of Scilab statements is meant to illustrate this somewhat abstract statement.

```
-->a = 1;

-->typeof(a)
 ans  =
 constant
```

```
-->convstr('AbCdE')
 ans  =
 abcde

-->typeof(convstr)
 ans  =
 function

-->a = convstr;

-->typeof(a)
 ans  =
 function

-->a('UvWxY')
 ans  =
 uvwxy
```

Initially, the variable a is assigned the value 1 and is of type constant. On the other hand, the function convstr, which converts upper case characters to lower case, is of type function. Obviously, like a, convstr is used as an argument of function typeof. Now I set a equal to convstr (note, that convstr is used without parentheses or argument). This turns a into a function and, as shown in the last statement, makes it an alias for convstr.
The fact that functions are variables has number of advantages not the least of which is that they can be input arguments or output arguments of other functions (in Matlab one needs to pass the function name as a string and use *feval* to evaluate it). A practical application is statement $\boxed{38}$ on page 103.

6.1 Functions in Files

Text files with Scilab function generally have the extension .sci though, in principle, any other extension (or no extension at all) is permissible (but see comments/restrictions below). Scilab functions exist in three forms: uncompiled, compiled, and compiled with provisions for profiling. More specifically, Scilab functions in text files are uncompiled. In order to be usable in Scilab they have to be compiled. To allow profiling, i.e. to determine how often each line is executed and how much time is spent executing it, one needs to add provisions for profiling. Profiling is explained in Section 6.4 starting on page 103.
Like in Matlab, there can be more than one function in a text file. But in Matlab the file name is actually the function name, and the second, third, etc. function in a file is only visible to the first function. In Scilab the file name is immaterial and all functions in a file are "visible" to any function loaded, command-line statement, or script.

In order to be able to use a function defined in a text file it has to be compiled and loaded first.[2] In Scilab, functions are loaded with the `exec`[3] command. However, some functions, like `genlib` and `getd` use the extension to recognize files with Scilab functions. Hence, it is a good idea to comply with this convention.
Scilab comes with an integrated text editor; it can be invoked via the statement `editor`, which opens the editor window. Statement `editor filename` opens the editor window and lodes the file `filename`, if it exists, or creates a new file by this name. One can also open the editor window via menu item "Editor" on the Scilab Console's drop-down menu "Applications". The editor comes with the standard syntax highlighting; but, apart from that, it is still fairly unsophisticated. However, using this editor — rather than a Scilab-independent editor — has a big advantage: files can be compiled and loaded directly into Scilab. In the statement `exec('filename')` command the argument `filename` is the name of the file containing the function; if this file is not in the working directory the full path must be given. Thus

```
-->getf('polcoord.sci')
```

is sufficient to load the file `polcoord.sci` if it is in the working directory. If this is not the case then something like

```
-->exec('D:\MyScilab\Filters\polcoord.sci')
```

must be used. It is important to note that the filename must include the extension (whereas Matlab implies the extension `.m`). Once `exec` is executed the functions in the file are immediately available. This differs from the `load` command discussed below. Also, see the "gotcha" regarding `exec` on page 112.
Functions can also be collected in libraries. This is discussed in Section 7.1. It is important to remember, however, that loading a library does not mean that the functions in it are loaded but rather that they are marked as available to be loaded when called—**provided** they are undefined at that time. If the name of a function happens to be that of an already defined function or a built-in function it will never be loaded. One can use `getf` to force loading of a function (provided it does not have the same name as a protected built-in function).
This last condition points to one of the challenges of writing a function: choosing its name. It is important that a name reflects the purpose of the function, is easy to remember, and is not already used. The set of names that satisfy these criteria is surprisingly small — significantly smaller than in Matlab. And there are four reasons:

1. Variable names are shorter (24 vs 63 characters in Matlab)

2. In Matlab a subfunction, i.e. a second, third, etc. function in a single file, is only visible to the first function in the file. So there is no conflict with any other function in Matlab's search path.

3. Matlab has the concept of "private functions". These are functions that reside in a subdirectory named `private` and that are only visible to functions in the parent directory: when

[2] This means a significant departure from the approach Matlab uses where the interpreter searches the directories of the search path and loads and compiles the function in the first file encountered with the name

[3] Function `getf` is obsolete and will be dropped in Version 5.3

a function in a directory that has a subdirectory `private` calls another function the subdirectory `private` is searched first to check if the function is there; only if it is not found the standard search path is checked.

4. Matlab has the concept of function handles which allows one—among other things—to pass a function reference to other functions. The function is resolved at the time the function handle is created. These other functions then use the value as a means to call the previously resolved function which need not be in the scope by the time it is called (incidentally, this is a way to make a Matlab subfunction available to functions outside the file in which it is created).

It is particularly the lack of the "private directory" concept that makes writing a program package that peacefully coexists with other packages more challenging than it needs be.

6.2 In-line Functions

Functions need not be set-up in files. They can also be created "on the fly". There are two ways to do so; one of them has already been used for examples in previous chapters (see e. g. function `ismember` on page 38). The function can be typed into the Scilab window as if it were typed in a file editor; the important thing to remember is that the statement `endfunction` is required to tell the interpreter that the function definition is complete. While the function statements are typed in, the usual double-spaced display format is replaced by single spacing.
The other way of inputting a function uses the function `deff`. A simple example of its use is

```
-->deff('y = funct(x)','y = x^2')

-->funct(3.5)
 ans  =
    12.25

-->typeof(funct)
 ans  =
 function

-->type(funct)
 ans  =
    13.
```

The first argument of `deff` is a character strings with the function header, the second is a string or a string vector which contains the body of the function. An optional third argument specifies if the function should be compiled (`'c'`) or not (`'n'`). The former is more efficient than the latter and is the default (there is actually a third possible value for the third input argument: `'p'` which prepares the function for profiling; see page 103). Thus, in the example above, `funct` is compiled. This is also proven by the fact that `funct` has type 13 (see Table 3.1). On the other hand, with

```
-->deff('y = funct(x)','y = x^2','n');
```

```
-->typeof(funct)
 ans  =
 function

-->type(funct)
 ans  =
    11.
```

The variable funct has type 11 (uncompiled function), while the output of typeof is unchanged. Another example for the use of deff is on page 103.
With more complicated functions or functions that contain string definitions the first version of in-line function definition is generally easier to read.
It is important to note that inline functions can be defined not just in the Scilab Console. They can also be included in Scilab scripts and functions.

6.3 Functions for operator overloading

Operator overloading refers to the ability to give operators that are used for one kind of data object a new meaning for another one. An example mentioned before is the use of the + to concatenate two strings or string matrices. But not only operators can be overloaded. The way a data object is displayed can be overloaded as well. For typed lists and matrix-oriented typed lists it is the type name, the first string in the first entry of a typed list, that defines the type of data object. The typed list seismic_data has type seismic; it simulates a seismic data set with 10 seismic traces, each consisting of 251 samples; hence, in the following, it is generally referred to as "seismic data set"

```
-->seismic_data = tlist(['seismic','first','last','step', ...
                  'units','traces'],0,1000,4,'ms');

-->nsamp = (seismic_data.last-seismic_data.first)/seismic_data.step+1;

-->seismic_data.traces=rand(nsamp,10);

-->seismic_data

       seismic_data(1)
!seismic  first  last  step  units  traces  !

       seismic_data(2)
    0.

       seismic_data(3)
    1000.
```

```
        seismic_data(4)
     4.

        seismic_data(5)
  ms

        seismic_data(6)
          column 1 to 5
     0.3914068    0.2173720    0.4883297    0.4061224    0.9985317
     0.8752304    0.4418458    0.9141346    0.9613220    0.1959695
     0.5266080    0.9798274    0.6645192    0.8956145    0.9872472
     0.9856596    0.5259225    0.5468820    0.0717050    0.4248699
 [More (y or n ) ?]
```

The default display of such a typed list is needlessly long; for this reason the function `show` had been introduced to provide a more compact display for typed lists (see page 51). It is highly desirable to use `show` as the default display of typed lists of type `seismic`. This can be done surprisingly easily by means of the function

```
function %seismic_p(seis)
// Function displays the typed list ``seis'' of type 'seismic' much like
// a Matlab structure
   show(seis)
endfunction
```

The result is

```
-->seismic_data =

LIST OF TYPE "seismic"
   first: 0
    last: 1000
    step: 4
   units: ms
  traces: 251 by 10 matrix
```

Overloading the way a variable is displayed is possible because the typed list of type `seismic` looks for a function with the name `%seismic_p`. As shown in this example the function name consists of the `%` sign followed by the type name, `seismic`, an underscore as a separator, and the letter `p` which indicates display (the fact that the underscore serves as a separator between the list type and the "p" does not mean that there cannot be an underscore in the type name).

In principle, any operator that is not predefined for given types of operands can be overloaded. The name of the overloading function is constructed according to certain rules. For the three unary operators `-`, `'`, and `~`, for example, it has the form %<operand_type>_<op_code>. An example is the use of the minus sign in front of the seismic-typed list `seismic_data` to change the sign of the entries of the matrix `seismic_data.traces`. As shown in Table 6.2 the operator code for the minus sign is `s`. Thus

Operator	Op-code	Operator	Op-code
'	t	\.	w
+	a	[a,b]	c
-	s	[a;b]	f
*	m	() extraction	e
/	r	() insertion	i
\	l	==	o
^	p	<>	n
.*	x	|	g
./	d	&	h
.\	q	.^	j
.*.	k	~	5
./.	y	.'	0
.\.	z	<	1
:	b	>	2
*.	u	<=	3
/.	v	>=	4

Table 6.2: Operator codes used to construct function names for operator overloading

```
function seismic = %seismic_s(seismic)
// Function defines the unary negation for a seismic data set
   seismic.traces=-seismic.traces;
endfunction
```

With the seismic data set seismic_data defined above

```
-->seismic_data.traces(1,1)
 ans  =
     0.2113249

-->seismic_data = -seismic_data;

-->seismic_data.traces(1,1)
 ans  =
  -  0.2113249
```

The function name for overloading binary operators has the form %<first_operand_type>_<op_code>_<second_operand_type>. In this definition <operand_type> is code for the type of variable the operator of type <op_code> is operating on. Operand types, i.e. codes for the various Scilab variables, are listed in the rightmost column of Table 3.1 on page 17. An example is the following function which defines the operation of adding a scalar to a seismic data set.

```
function seismic = %seismic_a_s(seismic,c)
// Function adds a constant to the matrix "seismic traces"
```

```
   seismic.traces = seismic.traces + c;
endfunction
```

Here `<first_operand_type>` is `seismic` and the `<second_operand_type>` is `s`. A quick look at Table 3.1 shows that `s` is the operand type of a constant. As shown in Table 6.2, `a` is the operator code for `+` (addition). Thus

```
-->seismic_data.traces(1,1)
 ans  =
     0.2113249

-->seismic_data = seismic_data + 1;

-->seismic_data.traces(1,1)
 ans  =
    1.2113249
```

It is important to note that overloading the operator `+` via `%seismic_a_s(seismic,c)` is only defined for this specific sequence of operands. The expression `1 + seismic_data` causes an error message as does, for example, `seismic_data - 1`; but `seismic_data + (-1)` works.
Another example of overloading the `+` operator is on page 111.
Some primitive functions can also be overloaded if they are not defined for the data type. In this case the function name has the form `%<type_of_argument>_<function_name>`. The function below takes the absolute value of the traces of a seismic data set.

```
function seismic = %seismic_abs(seismic)
// Function takes the absolute value of the entries of the
// matrix ‘‘seismic.traces’’
   seismic.traces = abs(seismic.traces);
endfunction
```

Thus, for the seismic data set `seismic_data` defined above,

```
-->seismic_data.traces(1,1)
 ans  =
     0.2113249

-->seismic_data = abs(-seismic_data);

-->seismic_data.traces(1,1)
 ans  =
     0.2113249
```

Extraction of object elements can be overloaded by means of a function the name of which has the form `%<operand_type>_e(i1,...,in,operand)`. A somewhat simplified example is

```
function seis = %seismic_e(i,j,seis)
// Function extracts rows i and columns j of the ...
```

```
// matrix "seis.traces"; i and j can be vectors
   seis.traces = seis.traces(i,j);
   seis.last = seis.first+(i($)-1)*seis.step;
   seis.first = seis.first+(i(1)-1)*seis.step;
endfunction
```

which outputs a seismic data set where seis.matrix consists only of the elements `i,j` of the input matrix. An example is

```
-->seismic_data.traces(5,10)
 ans  =
     0.1853351

-->temp = seismic_data(5,10)

 temp  =

LIST OF TYPE "seismic"
   first: 16
    last: 16
    step: 4
  traces:  0.1853351
   units: ms
```

It is important to keep in mind that typed lists have extraction (component extraction) defined for one index. Hence, extraction with one index cannot be overloaded, and

```
-->seismic_data(4)
 ans  =
    4.
```

produces the fourth element of typed list `seismic`, `step`, which is 4 (remember that the first element is the string vector `['seismic','first','last','step,'traces']`.
It is also possible to extract data object elements to more than one output object. Furthermore, the insertion syntax and row and column concatenation can also be overloaded.
Overloading insertion allows one to add a new field to a typed list simply by assigning a value to it—analogous to the way a field of a Matlab structure can be defined. For example, the name of the function that performs insertion of a field that can accept a numeric value into a typed list with type name `seismic` is `%c_i_seismic`. It has the form

```
function seis=%s_i_seismic(field,value,seis)
// Function adds a numeric field to a typed list with
// type name seismic and sets its value.
// INPUT
// seis      typed list to which to add the field
// value     numeric value to be assigned to the field
// field     string with name of field
```

```
// OUTPUT
// seis      typed list with new field "field" set to "value"

   temp=getfield(1,seis);
   temp($+1)=field;
   setfield(1,temp,seis);
endfunction
```

If this function has been loaded or is in a library one can create a new field— in this example `reel_no`—simply by assigning it a numeric value.

```
-->seismic_data.reel_no=2345
 seismic_data  =
Seismic data set
    first: 0
     last: 1000
     step: 4
    units: ms
   traces: 251 by 10 matrix
  reel_no: 2345
```

However, this function handles numeric scalars, vectors, or matrices only; no character strings or other data objects. Thus

```
-->seismic_data.line = 'EW-3241';
                    !--error     4
undefined variable : %c_i_seismic
```

Overloading insertion of a field for a string-type variable is not yet defined. However, the function body of `%c_i_seismic` is identical to that of `%s_i_seismic`. Hence, it is enough to copy `%s_i_seismic` to `%c_i_seismic` and bingo!

```
-->%c_i_seismic = %s_i_seismic    [38]
 %c_i_seismic  =
 [seis]=%c_i_seismic(field,value,seis)

-->seismic_data.line = 'EW-3241'
 seismic_data  =
LIST OF TYPE "seismic"
    first: 0
     last: 1000
     step: 4
    units: ms
   traces: 251 by 10 matrix
   reel_no: 2345
     line: EW-3241
```

Statement $\boxed{38}$ illustrates a practical use of the fact (discussed on page 94) that a function is just a special type of variable.

6.4 Profiling of functions

For those concerned about execution times and function efficiency, Scilab offers a profiler which works somewhat differently compared to Matlab's profiling facility. It requires that the function to be profiled is compiled with `exec` and then prepared for profiling via a call to function `add_profiling` or that it is defined via `deff` with the profiling option turned on. This is illustrated in the following example. Profiling is turned on if the optional third argument of `deff` is set to `'p'`.

```
-->deff('y=funct(n)',['y=rand(n,n)';'for ii=1:10';'y=sqrt(y)';'end'],'p')


-->y=funct(1000);      // Execute function ''funct''

-->profile(funct)      // Profile function ''funct''
 ans  =

   1.     0.          0.
   1.     0.000094    4.
   10.    0.          0.
   10.    0.000906    3.
   1.     0.          0.
```

The output of `profile` is a three-column matrix with one row for each line in function `funct`. The number in the first column is the number of times each line of `funct` has been executed. The numeric value in the second column represents the total execution time in seconds for each line. The number in the last column reflects the effort of the Scilab interpreter for each line. Of course, one has to know which particular line of the function corresponds to a row of the profile matrix. Function `showprofile` provides this association.

```
-->showprofile(funct)
function y=fun(n)|1 |0|0|
y = rand(n, n);  |1 |0|4|
for ii = 1:10,   |10|0|0|
  y = sqrt(y);   |10|0|3|
end              |1 |0|0|
```

It displays the profile matrix attached to the listing of function `funct`; small matrix entries are rounded to 0 (see function `clean`).
It is also possible to represent the result of profiling in graphic form (`plotprofile`).

6.5 Translation of Matlab m-files to Scilab Format

A function, `mfile2sci`, is available to translate Matlab m-files to Scilab. This function is still being worked on by someone in the Scilab team (it is likely to be a never-ending task) but the existing version greatly simplifies this kind of conversion—provided the files are not too long or too complicated. It relieves the user of a lot of drudgery and lets him concentrate on the thornier problems: instances where Matlab and Scilab functions may differ slightly, possibly depending on parameters in the argument list, where functions unknown to mfile2sci are used, etc. mfile2sci allows individual files or whole directories to be converted. In the process it creates a *.sci file, a *.cat file (help file generated from the comment lines at the beginning of the m-file, those lines that are also used by Matlab's help facility), and, if possible, a "compiled" *.bin file. The latter is. An example is

```
mfile2sci('D:\MyScilab\Geophysics\read_las_file.m', ...
          'D:\MyScilab\Geophysics')
```

If no input argument is provided mfile2sci opens a Graphic User Interface (GUI) window. It allows interactive selection of either the Matlab m-file or a directory with m-files; the user can also select the directory to which the Scilab files created should be saved.
Another related function is translatepaths which translates all Matlab m-files in a set of directories to Scilab. It uses mfile2sci to translate the individual files. If called without an input argument it opens the very same GUI mfile2sci does. In this case mfile2sci and mfile2sci are equivalent.

Chapter 7

Function Libraries and the Start-up File

7.1 Creating function libraries

This is a topic that has no analog in Matlab. Libraries are collections of compiled functions that can be loaded automatically upon start-up or that can be loaded on demand. There are several ways of creating libraries; the one described in the following appears to be the least painful.
Say, `C:\MyScilab` is a directory/folder with two Scilab functions (file extension `.sci`).

```
-->unix_w('ls C:\MyScilab')   // List files in folder C:\MyScilab
 lower.sci
 upper.sci
```

Then a possible procedure to create a library is as follows;

```
-->genlib('Mylib','C:\MyScilab')
```

Function `genlib` compiles every Scilab function (file with extension `.sci`) in directory `C:\MyScilab` and saves it in a file with the same root but extension `.bin`. It also creates the text file `names` with the names of all functions, and a library file `lib`. Hence, directory `C:\MyScilab` now contains the following files

```
-->unix_w('ls C:\MyScilab')
 lib
 lower.bin
 lower.sci
 names
 upper.bin
 upper.sci
```

In addition, the variable `Mylib` of type library (numeric type code 14) is created in the workspace, and all Scilab functions in `C:\MyScilab` are now available for use.
It is important to note that this does not create help files. This has to be done separately.
Furthermore, the statement

```
load('D:\MyScilab\lib');
```

in the start-up file `.scilab` or `scilab.ini` will load the library `Mylib` every time Scilab is started. Note that the library name `Mylib` is not mentioned in the load statement. Nevertheless, the variable `Mylib` of type library shows up in the workspace (the library `lib` "knows" that its name in the workspace is `Mylib`).
There are a few things to keep in mind with regard to loading libraries. This is illustrated in the following.

```
-->clear  // Remove all unprotected variables from the workspace

-->who    // Show all variables
 your variables are...
%helps    scicos_pal          MSDOS     home      PWD TMPDIR
 plotlib   percentlib          soundlib  xdesslib  utillib tdcslib
 siglib    s2flib    roblib    optlib    metalib   elemlib commlib
 polylib   autolib   armalib   alglib    intlib    mtlblib   WSCI
 SCI       %F        %T        %z        %s        %nan    %inf
 $         %t        %f        %eps      %io       %i      %e
 using       5517 elements  out of   10000000.           and 41
variables out of       1791

-->lc = lower('ABC')
                !--error     4
undefined variable : lower
```

Since all unprotected variables have been removed the function `lower` is not available. To get it back one can load the library `Mylib`, and the command `who` shows that the variable `Mylib` is now in the workspace.

```
-->load('D:\MyScilab\lib') // Load library containing function lower

-->who
 your variables are...
 Mylib     %helps    scicos_pal          MSDOS     home
 PWD       TMPDIR    plotlib   percentlib          soundlib  xdesslib
 utillib   tdcslib   siglib    s2flib    roblib    optlib  metalib
 elemlib   commlib   polylib   autolib   armalib   alglib  intlib
 mtlblib   WSCI      SCI       %F        %T        %z        %s
 %nan      %inf      $         %t        %f        %eps      %io
 %i        %e
 using       5553 elements  out of   10000000.
         and         42 variables out of       1791
```

However, functions `lower` and `upper` are not yet in the workspace. Loading a library does not mean that the functions in it are loaded into the workspace; they are only marked as available to be loaded when called. Thus, we can now execute the function `lower` and expect it to be loaded.

```
-->lc = lower('ABC')
 lc  =
 abc

-->who
 your variables are...
 lc         lower      Mylib      %helps     scicos_pal
 MSDOS      home       PWD        TMPDIR     plotlib    percentlib
 soundlib   xdesslib   utillib    tdcslib    siglib     s2flib  roblib
 optlib     metalib    elemlib    commlib    polylib    autolib   armalib
 alglib     intlib     mtlblib    WSCI       SCI        %F         %T
 %z         %s         %nan       %inf       $          %t         %f
 %eps       %io        %i         %e
 using        5650 elements   out of    10000000.
          and          44 variables out of       1791
```

This statement adds two more variables to the workspace: the string variable `lc` and the function `lower`.
Functions in libraries are actually loaded only if they are still undefined and their name is encountered during execution! Thus a potential problem exists if the library function name is the same as that of a built-in function or an already defined user function. In this case it would not be loaded. A related problem would come up if one found a bug in, say, `lower`, fixed it, and rebuilt and reloaded the library. If `lower` is executed again one would not get the corrected version in the rebuild library `Mylib` but rather the one in variable `lower`. Hence, in order to get the corrected version it is not only necessary to rebuild and load the new library but also to remove the variable `lower` from the workspace; in other words: it is necessary to execute the command

```
-->clear lower
```

An alternative is to bring the corrected version of `lower` into the workspace via

```
-->getf('C:\MyScilab\lower.sci')
```

One of the benefits of the built-in editor editor is that all this compiling and loading is being taken care of automatically if one uses the submenu items "`Load into Scilab`" of the "`Execute`" drop-down menu.

7.2 Start-up file

Matlab users who want to write their own functions tend to look for a "Scilab Path", something akin to the Matlab Path. As mentioned before, there is no equivalent to the Matlab path in Scilab; instead, users themselves must "load" their functions into Scilab. They can, for example, create libraries as described above and load them into Scilab and, fortunately, this can be automated via the "start-up file".
This start-up file has been mentioned earlier; its name can be either `.scilab` or `scilab.ini` and, for MS Windows, it needs to be in directory `SCIHOME`, i.e. in the directory defined by the global variable `SCIHOME`. The following start-up file is a simplified example of a start-up file.

```
// Start-up file

stacksize(10000000);   // Set the size of the stack
lines(1000)            // Increase the limit on the number of
                       // lines displayed

mydir='G:\Backed-up\MyScilab\';  // Specify the folder with user functions

// Initialize a global structure with default settings
global MYSCILAB
MYSCILAB=tlist(['struct','directory','sce_path','sci_path'])

// Set a field of the global structure
MYSCILAB.directory=mydir;

sep=filesep();         // Select the appropriate file separator
                       // for constructing dirctories

// Path for scripts (one directory)
MYSCILAB.sce_path=mydir+'Scripts'+sep;

// Path for functions (two different directories)
MYSCILAB.sci_path=[mydir+'Seislab'+sep; mydir+'General'+sep]

// Create two personal libraries
genlib('Generallib',mydir+'General')
genlib('Geophysicslib',mydir+'Seislab')

// Load the personal libraries
load(mydir+'Seislab'+sep+'lib')
load(mydir+'General'+sep+'lib')
```

It sets the stack size and specifies how many lines should be displayed before the user is prompted to decide whether to continue or abort. Then a global typed list, `MYSCILAB`, is defined so that a few parameters are readily available in the workspace. Finally, two libraries are created and loaded into the workspace. As mentioned above, the functions in them are not loaded right away but will be loaded when called.

7.3 User-supplied Libraries

For quite a long time the Scilab web site has maintained directories with user-supplied functions that expand Scilab's capabilities in specific fields. These modules can now be downloaded and installed directly from the Scilab Console (of course, an Internet connection is required). All it takes is

clicking on submenu item "Module Manager – ATOMS" on the Scilab Console's drop-down menu "Applications", selecting from the list of modules, and clicking on the install button.
Modules can also be installed via function calls from the Scilab Console. For example, function `atomsLoad` installs one or more external modules, and `atomsRemove` removes installed modules. Table 7.1 list a number of these functions.

Scilab	Description
`atomsAutoLoadList`	Get list of the modules scheduled for auto-loading
`atomsGetInstalled`	Get list of installed external modules
`atomsInstall`	Installs one or more external modules
`atomsIsInstalled`	Checks is a certain external module is installed
`atomsLoad`	Install one or more external modules
`atomsRemove`	Remove one or more external modules
`atomsUpdate`	Update one or more external modules

Table 7.1: Functions installing, managing and removing user-supplied Scilab modules

Chapter 8

Error Messages and Gotchas

8.1 Scilab error messages

More often than not, error messages of computer programs are considered cryptic. They reflect the thinking of him who wrote the program and do not seem to meant for those who use them; Scilab confirms this general experience. However, error messages have improved. 14

8.1.1 !–error 4: undefined variable:

This is a popular message where the colon is frequently followed by a strange variable name. An example is

```
-->three='3'
 three  =
 3

-->str=three+4                    39
            !--error     144
Undefined operation for the given operands.
check or define function  %c_a_s for overloading.
```

The error committed here is obviously the fact that statement 39 tries to add the number 4 the string '3'. This is an operation that is not defined in Scilab. The following is a brief explanation of the composition of the function name. More details are in the section on operator overloading that begins on page 98. Functions that overload a binary operator (in this case the `+` sign) have the form %<first_operand_type>_<op_code>_<second_operand_type>. Here the type of the first operand is `c`, that of the second operand is `s`. The rightmost column of Table 3.1 on page 17 shows that operand type `c` denotes a character string and operand type `s` a numeric variable. Furthermore, Table 6.2 on page 100 shows that operator code `a` denotes addition. Hence, function `%c_a_s`—if it existed—would "add" a constant to a string. To demonstrate this define

```
function str=%c_a_s(str,num)
// Convert a numeric variable into a string and append it to another string
// INPUT
```

```
// str     string
// num     numeric variable
// OUTPUT
// str     input string with numeric variable appended

   str=str+string(num);
endfunction
```

and then execute statement $\boxed{39}$ again.

```
-->str=three+4
 str  =
 34
```

The error message is gone. However, it is important to remember that `4 + three` would still cause an error message "undefined variable". To catch it as well one needs to define function `%s_a_c` where the operands are reversed.

8.1.2 !–error 66: Too many files opened!

As explained on page 71 a user can open no more than 16 files before Scilab runs out of logical unit numbers. The problem is that he may not be aware that he has so many open files. This can happen if he repeatedly runs a misbehaving program that opens a file and does not close it again (possibly because it terminates abnormally). Function `dispfiles`, which displays a list of all open files, also fails with an error message. To then close all open files use the command `mclose('all')`.
Be careful with `mclose('all')`, though. If it is used inside a Scilab script file, it also closes the script; consequently, Scilab will not execute commands following `mclose('all')`. And there will be no error message.

8.2 Gotchas

This section deals with unexpected problems I encountered during my travails.

Function `getf`

Function `getf`[1] reads and compiles a function from a file (see page 96). In case it encounters an error while compiling it aborts with an error message. When one corrects the error and wants to save the file to repeat the call to `getf` one finds out that this is not possible since `getf` has not closed the file. The sledge-hammer approach to this problem is to close all user files with `mclose all`. A more nimble approach is to find the offending file's identifier by executing `dispfiles()` and then close only that specific file. This is illustrated below

```
-->dispfiles()
|File name                                |Unit|Type|Options          |
|----------------------------------------------------------------------|
```

[1]Function `getf` has been deprecated and will be removed in Scilab version 5.3

```
|D:/PROGRAM FILES/SCILAB-2.6/scilab.hist |1   |F77 |unknown formatted |
|D:\MyScilab\Tests\read_segy_file.sci    |2   |C   |r b               |
|Input                                   |5   |F77 |old formatted     |
|Output                                  |6   |F77 |new formatted     |

-->mclose(2);
```

Line numbers in error messages

Line numbers displayed with error messages all too frequently do not agree with the line numbers of the offending statement in the function file. Apparently, there are various reasons for that. If there are comment lines prior the function header those lines are not counted. Other irregularities seem to be associated with expressions that continue over two or more lines.

Appendix A

Matlab functions and their Scilab Equivalents

The following table is an alphabetic list of Matlab functions and their Scilab functional equivalents. The third column contains one-line descriptions that pertain to the Scilab function and not to the Matlab function (in case there is a difference). In some instances the term "equivalent" is defined rather loosely; parameters may be different or the output may be somewhat different in certain circumstances (an example is the Scilab function `length` which may return, for numeric matrices or string matrices, a different result than the Matlab function *`length`*). In other cases functions provide the same functionality, but in a somewhat different way. For this reason it is not generally sufficient to replace a Matlab function by the equivalent listed here; it is necessary to check the Scilab help file before using one of these equivalents.

Matlab	Scilab	
[]	[]	Empty matrix
abs(a)	abs(a)	Absolute value of a, $\|a\|$
acos	acos	Arc cosine
acosh	acosh	Inverse hyperbolic cosine
all	and	Logical AND of the elements of boolean or real numeric matrix a
any	or	Logical OR of the elements of boolean or real numeric matrix a
asin	asin	Arc sine
asinh	asinh	Inverse hyperbolic sine
atan	atan	Arc tangent
atan2	atan	Arc tangent
atanh	atanh	Inverse hyperbolic tangent
balance	balanc	Balance matrix to improve condition number

Table A.1: Matlab functions and their Scilab equivalents

Matlab	Scilab	
besselh	besselh	Bessel function of the third kind (Hankel function), $H_\alpha(x)$
besseli	besseli	Modified Bessel function of the first kind, $I_\alpha(x)$
besselj	besselj	Bessel function of the first kind, $J_\alpha(x)$
besselk	besselk	Modified Bessel function of the second kind, $K_\alpha(x)$
bessely	bessely	Bessel function of the second kind, $Y_\alpha(x)$
break	break	Force exit from a for or while loop
case	case	Start clause within a select block
catch	catch	Begin catch block (after the try block)
ceil(a)	ceil(a)	Round the elements of a to the nearest integers ≥a
cell	cell	Initiate a cell array
char	ascii	Convert ASCII codes to equivalent string
chol(M)	chol(M)	Choleski factorization; R'*R = M
clear	clear	Clear unprotected variables and functions from memory
clear global	clearglobal	Clear global variables from memory
compan	companion	Companion matrix
cond	cond	Condition number of M
cond	cond	Condition number of a matrix
conj	conj	Complex conjugate
conv	convol	Convolution
cos	cos	Cosine
cosh	cosh	Hyperbolic cosine
cot	cotg	Cotangent
coth	coth	Hyperbolic cotangent
cumprod	cumprod	Cumulative product of all elements of a vector or array
cumsum	cumsum	Cumulative sum of all elements of a vector or array
date	date, getdate	Current date as string
dbstack	where	Output current instruction calling tree to variable
dbstack	whereami	Display current instruction calling tree
deblank	stripblanks	Remove leading and trailing blanks from a string
det	det	Determinant of a matrix
diag	diag	Create diagonal matrix or extract diagonal from matrix
diary	diary	Write screen output of a Scilab session to a file
disp	disp	Display input argument
double	double	Convert integer of any type/length to floating point
double	ascii	Convert string to equivalent ASCII codes
echo	mode	Control amount of information displayed by function/script
eig	spec	Eigenvalues of matrix
eig	bdiag	Block diagonalization of matrix
ellipj	%sn	Jacobian elliptic function, sn

Table A.1 (continued): Matlab functions and their Scilab equivalents

Matlab	Scilab	
else	else	Start an alternative in an if or case block
elseif	elseif	Start a conditional alternative in an if block
end [loop]	end	Terminate for, if, select, or while clause
end [matrix]	$	Index of last element of matrix or (row/column) vector
erf	erf	Error function, $\operatorname{erf}(x) = 2/\sqrt{\pi}\int_0^x \exp(-t^2)dt$
erfc	erfc	Complementary error function, $\operatorname{erfc}(x) = 2/\sqrt{\pi}\int_x^\infty \exp(-t^2)dt$
erfcx	erfcx	Scaled complementary error function, $\operatorname{erfcx}(x) = \exp(x^2)\operatorname{erfc}(x)$
error	error	Print error message and abort
eval	execstr	Evaluate string vector with Scilab expressions or statements
exist(a)	exists(a)	Test if variable a exists
exist(a,'dir')	isdir(a)	Test if directory a exists
exp	exp	Exponential function
expm	expm	Matrix xponential function
eye	eye	Identity matrix (or its generalization)
fclose	mclose	Close (all) open file(s)
fft	fft	Forward and inverse Fast Fourier Transform
fft2	fft	Forward and inverse Fast Fourier Transform
fftshift	fftshift	Shift zero-frequency component to center of spectrum
figure	xset	Set defaults for current graphics window
filesep	filesep	Returns the character that separates directory names (\ or /)
find	spget	Retrieve entries of a sparse matrix
find(a)	find(a)	Find the indices for which boolean matrix a is true
findstr	strindex	Find starting position(s) of a string in an other string
fix(a)	fix(a)	Rounds the elements of a to the nearest integers towards zero
flipdim(a)	flipdim(a)	Flip matrix a along a specified dimension
floor(a)	floor(a)	Rounds the elements of a to the nearest integers ≥ a
fopen	mopen	Open a file
for	for	Start a loop with a generally known number of repetitions
format	format	Set current display format of variables
fprintf	fprintf	Write formatted data to file (like C-language fprintf function)
full	full	Convert sparse to full matrix
function	function	Identify header of function definition
gamma	gamma	Gamma function, $\Gamma(x) = \int_0^\infty t^{x-1}\exp(-t)dt$
gammaln	gammaln	Logarithm of the Gamma function, $\ln(\Gamma(x))$
getenv	getenv	Get value of an environment variable
getfield	getfield	Get a data object from a list
global	global	Define variables as global
help	help	On-line help
hess(M)	hess(M)	Hessenberg form of M
if	if	Start a conditionally executed block of statements
ifft	fft	Forward and inverse Fast Fourier Transform

Table A.1 (continued): Matlab functions and their Scilab equivalents

Matlab	Scilab	
ifft2	fft	Forward and inverse Fast Fourier Transform
imag	imag	Imaginary part of a matrix
input	input	Prompt for user (keyboard) input
int16(a)	int16(a)	Convert a to 16-bit signed integer
int32(a)	int32(a)	Convert a to 32-bit signed integer
int8(a)	int8(a)	Convert a to 8-bit signed integer
intersect	intersect	Returns elements common to two vectors
inv(M)	inv(M)	Inverse of matrix M
isempty(a)	isempty(a)	Check if variable a is empty
isglobal(a)	isglobal(a)	Test if a is a global variable
isinf(a)	isinf(a)	Test if a is infinite
isnan(a)	isnan(a)	Output boolean vector with entries %t where a is %nan
isreal(a)	isreal(a)	Test if a is real (or if its imaginary part is "small")
keyboard	pause	Interrupt execution of function or script
length	length	Length of list; product of no. of rows and columns of matrix
linspace	linspace	Create vector with linearly spaced elements
load	loadmatfile	Load workspace variables from a disk file in Matlab format
log	log	Natural logarithm
log10	log10	Base-10 logarithm
log2	log2	Base-2 logarithm
logm	logm	Matrix natural logarithm
logspace	logspace	Create vector with logarithmically spaced elements
lookfor	apropos	Keyword search for a function
lower	convstr	Convert string to lower or upper case
max	max	Maximum of all elements of a vector or array
max	maxi	Maximum of all elements of a vector or array
mean	mean	Mean of all elements of a vector or array
median	median	Median of all elements of a vector or array
min	mini	Minimum of all elements of a vector or array
min	min	Minimum of all elements of a vector or array
mod(a,b)	pmodulo(a,b)	a-b.*floor(a./b) if b~=0; remainder of a divided by b
more	lines	Specify number of lines to display and columns/line
nargin	argn	Number of input/output arguments of a function
nargout	argn	Number of imput/output arguments of a function
nextpow2	nextpow2	For argument x compute smallest integer n such that $2^n \geq x$
nnz	nnz	Number of nonzero elements of a sparse matrix
norm(M)	norm(M)	Norm of M (matrix or vector)
null(M)	kernel(M)	Nullspace of M

Table A.1 (continued): Matlab functions and their Scilab equivalents

Matlab	Scilab	
num2str	string	Convert number(s) into string(s)
ones	ones	Matrix of ones
orth(M)	orth(M)	Orthogonal basis for the range of M
pause	halt	Stop execution and wait for a key press
pinv	pinv(M)	Pseudoinverse of M
planerot	givens	Given's rotation
prod	prod	Product of the elements of a matrix
profile	profile	Extract execution profiles from a Scilab function
qr(M)	qr(M)	QR decomposition of M
rand	rand	Create random numbers with uniform or normal distribution
randn	rand	Create random numbers with uniform or normal distribution
rank(M)	rank(M)	Rank of M
rcond(M)	rcond(M)	Reciprocal of the condition number of M; L-1 norm
real	real	Real part of a matrix
regexp	regexp	Match regular expression
rem(a,b)	modulo(a,b)	a-b.*fix(a./b) if b~=0; remainder of a divided by b
reshape	matrix	Reshape a vector or a matrix to a different-size matrix
return	resume	Return from a function or resume execution after a pause
return	return	Return from a function or resume execution after a pause
rmfield	null	Delete an element of a list
round(a)	round(a)	Round the elements of a to the nearest integers
save	savematfile	Save workspace variables to disk in Matlab format
schur(M)	schur(M)	Schur decomposition
setfield	setfield	Set a data object in a list
sign(a)	sign(a)	Signum function, $a/\|a\|$ for $a \neq 0$
sin	sin	Sine
sinc	sinc	Sinc function, $\sin(x)/x$
sinh	sinh	Hyperbolic sine
size	size	Size/dimensions of a Scilab object
sort(a)	gsort(a)	Sort the elements of a
sortrows(a)	gsort(a)	Sort elements/rows/columns of a
spalloc	spzeros	Sparse zero matrix
sparse	sparse	Create sparse matrix
speye	speye	Sparse identity matrix
spones	spones	Replace non-zero elements in sparse matrix by ones
sprand	sprand	Create sparse random matrix
sqrt(a)	sqrt(a)	$\sqrt{a}$
sqrtm	sqrtm	Matrix square root
sscanf	msscanf	Read variables from a string under format control

Table A.1 (continued): Matlab functions and their Scilab equivalents

Matlab	Scilab	
std	st_deviation	Standard deviation
strrep	strsubst	Substitute one string for another in a third string
strtok	tokens	Split string into substrings based on one or more “separators”
struct	struct	Create a structure
sum	sum	Sum of all elements of a matrix
svd	svd	Singular value decomposition
switch	select	Start a multi-branch block of statements
tan	tan	Tangent
tanh	tanh	Hyperbolic tangent
tic	tic	Starts clock for timing of a code segment
tic	timer	Starts clock for timing of a code segment
toc	toc	Outputs time elapsed since the preceding call to toc
toc	timer	Outputs CPU time used since the preceding call to timer
toeplitz	toeplitz	Toeplitz matrix
trace	trace	Trace (sum of diagonal elements) of a matrix
tril	tril	Extract lower-triangular part of a matrix
triu	triu	Extract upper-triangular part of a matrix
try	try	Trap error
uigetfile	uigetfile	Open dialog box for file selection
uint16(a)	uint16(a)	Convert a to 16-bit unsigned integer
uint32(a)	uint32(a)	Convert a to 32-bit unsigned integer
uint8(a)	uint8(a)	Convert a to 8-bit unsigned integer
union	union(a,b)	Extract the unique common elements of a and b
unique(a)	unique(a)	Return the unique elements of a in ascending order
unix	unix_w	Execute Unix/DOS command; output to Scilab window
upper	convstr	Convert string to lower or upper case
varargin	varargin	Variable number of input arguments for a function
varargout	varargout	Variable number of output arguments for a function
version	getversion	Display version of Scilab
warning	warning	Print warning message
which	whereis	Display name of library containing a specific function
while	while	Start repeated execution of a block while a condition is satisfied
who	who	Displays/outputs names of current variables
whos	whos	Displays/outputs names and specifics of current variables
zeros	zeros	Matrix of zeros

Table A.1 (continued): Matlab functions and their Scilab equivalents

Index

`.scilab`, *see* start-up file

, 82

arguments
  named, 62
  variable-length list of, 62, 94
ASCII codes, 25
ATOMS, 111

boolean operators, 35
boolean variables, 35–40

cell array, 3, 24, 41–43, 48–50
character string, *see* string
command-style syntax, 13, 61, 87
comments, 4
complex numbers, 19
constants, built-in, 7
continuation of a statement, 5
convolution, 70, 116

date, 15
directory
  private, *see* private directory
  temporary, 7
display of numbers, *see* format, numbers
DOS, *see* MS-DOS

editor, 4, 97, 109
end of file, 73
environmental variables, *see* variable, environmental
error trapping, 9, 11, 12, 32, 33, 78, 85
error messages
  undefined operation, 112
  undefined variable, 113
error trapping , 11–12

execution time, 105

Fast Fourier Transform, 70, 72, 117, 118
FFT, *see* Fast Fourier Transform
fields, 51
file, 72–81
  opening, closing, 72–73
flow control, 9
font, 1
format
  compact, vi
  loose, vi
  numbers, 1, 2
function
  handle, 98
  library, *see* library
  loading, 97
functions
  as arguments, 96
  as variables, 95, 104
  general discussion, 61–84
  inline, 98, 99
  overloading, 102
  private, *see* private functions
  profiling, 105
  user, 91–104

global variables
  `HOME`, 7
  `LANGUAGE`, 7
  `MSDOS`, 8
  `OS`, 7
  `PWD`, 7, 85
  `SCIHOME`, 1, 7
  `SCI`, 1, 3, 7, 8
  `TMPDIR`, 7, 88
  `home`, 7

help, 3

integers, 20, 22, 80
interrupt, *see* pause

Kronecker
  division, 5
  product, 5

language, 7
last element
  of vector or matrix, 15, 19
LaTeX, 82
length
  list, 50
  numeric matrix, 13
  string, 24, 25
libraries
  user-supplied, 110
library, 97, 107–109
lines displayed, 2
list, 42, 46, 47
  matrix-oriented typed, 46, 56–57
  ordinary, 48–51
  polynomials, 57–60
  typed, 50–56, 103, 110
logical variables, *see* boolean variables

Matlab
  emulation, 3
  m-file conversion, 105
  path, *see* path
Matlab format
  loading variables saved in, 80
  saving variables in, 80
Matlab functions
  `OS`, 7
  `[]`, 115
  `abs`, 115
  `acosh`, 115
  `acos`, 115
  `all`, 37, 115
  `any`, 37, 115
  `asinh`, 115
  `asin`, 115
  `atan2`, 66, 115
  `atanh`, 115
  `atan`, 66, 91, 115
  `balance`, 115
  `besselh`, 116
  `besseli`, 116
  `besselj`, 116
  `besselk`, 116
  `bessely`, 116
  `break`, 9, 116
  `case`, 9, 116
  `catch`, 9, 11, 116
  `ceil`, 116
  `cell`, 24, 116
  `char`, 25, 116
  `chol`, 116
  `clear global`, 116
  `clear`, 116
  `compan`, 116
  `cond`, 116
  `conj`, 116
  `conv`, 116
  `cosh`, 116
  `cos`, 116
  `coth`, 116
  `cot`, 116
  `cumprod`, 116
  `cumsum`, 116
  `date`, 116
  `dbstack`, 116
  `deal`, 6
  `deblank`, 116
  `det`, 116
  `diag`, 116
  `diary`, 82, 116
  `disp`, 81, 116
  `double`, 25, 116
  `echo`, 116
  `eig`, 116
  `ellipj`, 116
  `elseif`, 9, 117
  `else`, 9, 117
  `end [loop]`, 9, 117

end [matrix], 53, 117
end, 19
erfcx, 117
erfc, 117
erf, 117
error, 117
evalin, 33
eval, 31, 117
exist, 13, 117
expm, 117
exp, 117
eye, 117
false, 36
fclose, 117
feval, 96
fft2, 70, 117
fftshift, 117
fft, 70, 117
fieldnames, 46
figure, 117
filesep, 90, 117
findstr, 117
find, 117
finish, 3
fix, 117
flipdim, 117
floor, 117
fopen, 72, 81, 117
format compact, vi
format loose, vi
format, 117
for, 9, 117
fprintf, 117
full, 117
function, 117
funm, 67
gammaln, 117
gamma, 117
getenv, 117
getfield, 117
global, 117
help, 3, 117
hess, 117
ifft2, 70, 118
ifft, 70, 117
if, 9, 117
imag, 118
input, 81, 118
int16, 118
int32, 118
int8, 118
intersect, 118
inv, 118
isa, 17
iscell, 17
ischar, 17
isempty, 118
isglobal, 118
isinf, 118
ismember, 29, 38
isnan, 118
isnumeric, 17
isreal, 118
issparse, 17
isstruct, 17
keyboard, 13, 14, 95, 118
length, 118
linspace, 118
load, 80, 118
log10, 118
log2, 118
logical, 35, 36
logm, 118
logspace, 118
log, 118
lookfor, 3, 118
lower, 118
matlabroot, 7
max, 63, 118
mean, 118
median, 118
min, 118
mod, 118
more, 118
nargin, 95, 118
nargout, 95, 118

nextpow2, 118
nnz, 118
norm, 118
null, 118
num2str, 119
ones, 119
orth, 119
otherwise, 9
pause, 119
pinv, 119
planerot, 119
prefdir, 7
prod, 119
profile, 119
pwd, 7
qr, 119
randn, 119
rand, 119
rank, 119
rcond, 119
real, 119
regexp, 31, 119
rem, 119
reshape, 119
return, 119
rmfield, 119
roots, 58
round, 119
save, 79, 119
schur, 119
setfield, 47, 119
sign, 119
sinc, 119
sinh, 119
sin, 119
size, 24, 57, 119
sortrows, 65, 119
sort, 64, 119
spalloc, 119
sparse, 119
speye, 119
spones, 119
sprand, 119
sqrtm, 119
sqrt, 119
sscanf, 30, 119
std, 120
strrep, 120
strtok, 27, 120
struct, 120
sum, 120
svd, 120
switch, 9, 120
tanh, 120
tan, 120
tempdir, 7
tic, 120
toc, 120
toeplitz, 120
trace, 120
tril, 120
triu, 120
true, 36
try, 9, 11, 120
typeof, 58, 60
type, 60
uigetfile, 81, 120
uint16, 120
uint32, 120
uint8, 17, 120
union, 120
unique, 40, 120
unix, 120
upper, 120
varargin, 120
varargout, 120
version, 120
warning, 120
which, 120
while, 9, 120
whos, 120
who, 120
zeros, 120
Matlab-mode, 22
matrix, 18–20, 23, 120
companion, 23, 116

empty, 23, 115
Frank, 23
Hilbert, 23
identity, 23, 117
of ones, 23, 119
of zeros, 23, 120
polynomial, *see* polynomial, matrix, 59
random, 23, 119
rational, 59
reshape a, 119
sparse, 70
sparse identity, 70, 119
sparse random, 70, 119
Toeplitz, 23, 120
MS-DOS, 83

named arguments, *see* arguments, named

operator
binary, 101, 112
code, 101, 102
overloading, 17, 25, 52, 99–104, 112, 113
unary, 100
overloading
operator, *see* operator, overloading
variable display, *see* variable, display overloading

paging, *see* lines displayed
path, 32, 88–90, 97, 109
polynomial, 57–60
division, 59
matrix, 59
precision, 20
primitive, 84
print-out control, 85–87
private directory, 97, 98
private function, 97
profiling, 96
profiling of functions, 98, 105
program
abort, 15, 92
exit, 2
interrupt, 2, 12
quit, 2
resume, 2
return, 2

random numbers, 23
random numbers, 22, 23
regular expressions, 31

Scilab
script, *see* script
syntax, 4, 5
Scilab functions and variables
`[]`, 22, 23, 115
`$`, 15, 19, 20, 53, 117
`%asn`, 68
`%k`, 68
`%sn`, 68, 116
`abort`, 2, 15, 92
`abs`, 64, 115
`acoshm`, 68
`acosh`, 67, 115
`acosm`, 68
`acos`, 67, 115
`add_profiling`, 105
`amell`, 68
`and`, 37, 38, 115
`apropos`, 3, 15, 118
`argn`, 92, 95, 118
`ascii`, 25, 26, 28, 116
`asinhm`, 68
`asinh`, 67, 115
`asin`, 67, 115
`atanhm`, 68
`atanh`, 67, 115
`atan`, 66, 67, 91, 115
`atomRemove`, 111
`atomsAutoLoadList`, 111
`atomsGetInstalled`, 111
`atomsInstall`, 111
`atomsIsInstalled`, 111
`atomsLoad`, 111
`atomsRemove`, 111
`atomsUpdate`, 111
`auread`, 74

auwrite, 75
balanc, 69, 115
bandwr, 70
basename, 26, 73, 82, 83
bdiag, 69, 116
besseli, 68, 116
besselj, 68, 116
besselk, 68, 116
bessely, 68, 116
bezout, 59
blanks, 26
bool2s, 36, 38, 64
break, 9, 116
calerf, 68
case, 9, 83, 116
catch, 9, 11, 116
ceil, 64, 116
cell array, 57
cell2mat, 41
cell, 41, 116
chdir, 88
chfact, 70
chol, 69, 116
chsolve, 70
clean, 59, 64, 67, 105
clearglobal, 15, 116
clear, 15, 108, 109, 116
cmndred, 59
coeff, 59
coffg, 59
colcompr, 59
colcomp, 69
companion, 23, 116
complex, 19
cond, 69, 116
conj, 64, 116
convol, 70, 116
convstr, 26, 96, 118, 120
corr, 70
coshm, 68
cosh, 67, 116
cosm, 68
cos, 67, 116
cotg, 67, 116
coth, 67, 116
cumprod, 63, 64, 116
cumsum, 63, 64, 116
date, 15, 26, 116
deff, 98, 99, 105
degree, 59
delip, 68
denom, 59
derivat, 59
determ, 59
detr, 59
det, 59, 69, 116
diag, 23, 116
diary, 75, 82, 116
diophant, 59
dirname, 73, 82
dispfiles, 73, 113
disp, 15, 75, 81, 116
dlgamma, 68
double, 20, 116
editor, 97
elseif, 9, 117
else, 9, 117
emptystr, 24, 26, 28
endfunction, 92
end, 9, 117
erfcx, 68, 117
erfc, 68, 117
erf, 68, 117
errcatch, 9, 11, 12
errclear, 11, 12
error, 83, 92, 117
eval, 31, 32
evstr, 31, 33, 34
excel2sci, 74
execstr, 31, 32, 85, 89, 117
exec, 86, 87, 97, 105
exists, 13, 36, 38, 117
exit, 2
expm, 68, 117
exp, 67, 117
eye, 23, 117

factors, 59
fftshift, 70, 117
fft, 70, 117, 118
fileinfo, 73
filesep, 90, 117
file, 72, 73, 77, 78, 89
findfiles, 82, 83
find, 38, 117
fix, 64, 117
flipdim, 117
floor, 64, 117
format, 1, 2, 117
for, 9, 117
fprintfMat, 75
fprintf, 75, 117
fscanfMat, 74
fscanf, 74
fullrfk, 69
fullrf, 69
full, 70, 117
fun2string, 82, 84
function, 92, 117
gammaln, 68, 117
gamma, 68, 117
gcd, 59
genlib, 97, 107
getdate, 15, 116
getd, 97
getenv, 1, 3, 8, 82, 117
getfield, 46–48, 117
getf, 97, 109, 113
getio, 73
getlanguage, 7
getos, 8, 83, 88, 89, 92
getversion, 82, 120
givens, 69, 119
global, 15, 93–95, 117
grand, 23
grep, 26, 29, 31
gsort, 26, 64, 65, 119
halt, 12, 15, 92, 119
help, 3, 15, 117
hermit, 59
hess, 69, 117
horner, 59
host, 82, 83
householder, 69
hrmt, 59
iconvert, 20
if, 9, 10, 117
imag, 64, 118
input, 74, 81, 118
int16, 20, 118
int32, 20, 118
int8, 20, 118
intersect, 26, 64, 118
inttype, 15, 17, 18
invr, 59
inv, 59, 69, 118
isalphanum, 26
isascii, 26, 38
iscellstr, 38
iscell, 38, 42
isdef, 36, 38
isdigit, 26, 38
isdir, 38, 73, 117
isempty, 38, 118
iserror, 11, 38
isfield, 38
isglobal, 38, 118
isinf, 38, 118
isletter, 26, 38
ismember, 29, 39, 40
isnan, 38, 118
isnum, 26, 38
isreal, 118
isstruct, 38, 46
kernel, 69, 118
lcmdiag, 59
lcm, 59
ldiv, 59
length, 3, 24, 26, 46, 48, 50, 118
lex_sort, 64
lines, 82, 118
linsolve, 69
linspace, 64, 118

listfiles, 73, 82, 83
list, 18, 47, 48
loadmatfile, 74, 80, 118
loadwave, 74
load, 74, 79, 80, 97, 108
log10, 67, 118
log2, 67, 118
logm, 68, 118
logspace, 64, 118
log, 67, 118
lstcat, 48
lufact, 70
luget, 70
lusolve, 70
macrovar, 92
makecell, 41
matrix, 76, 119
maxi, 63, 64, 118
max, 63, 64, 118
mclearerr, 73
mclose, 73, 75, 113, 114, 117
mean, 64, 118
median, 64, 118
meof, 73
mfft, 70
mfile2sci, 106
mfprintf, 75
mfscanf, 74
mgeti, 74, 80
mgetl, 73–76
mgetstr, 74
mget, 74, 80, 81
mini, 63, 64, 118
min, 63, 64, 118
mlist, 18, 46–48, 56
mode, 85–87, 92, 116
modulo, 64, 119
mopen, 72, 73, 75, 77, 79–81, 117
mprintf, 75
mputl, 73, 75–77
mputstr, 75
mput, 75, 80
mscanf, 74
mseek, 73
msprintf, 26
msscanf, 26, 30, 119
mtell, 73
mtlb_mode, 22, 38
newest, 73
nextpow2, 70, 118
nnz, 70, 118
norm, 69, 118
null, 46–49, 119
ones, 23, 36, 119
orth, 69, 119
or, 37, 38, 115
part, 26
pathconvert, 26, 73, 82, 88, 90
pause, 2, 12–15, 92, 95, 118
pdiv, 59
pinv, 69, 119
plotprofile, 92
pmodulo, 64, 118
pol2str, 26, 59
polar, 69
poly, 57
prettyprint, 82
printf, 75
print, 75
prod, 63, 64, 119
profile, 92, 105, 119
pwd, 85
qr, 69, 119
quit, 2
rand, 23, 119
range, 69
rank, 69, 119
rcond, 69, 119
read4b, 74
readb, 74
readc_, 74
read, 74, 76–78
real, 64, 119
regexp, 26, 31, 119
reshape, 76
residu, 59

resume, 2, 14, 15, 92, 95, 119
return, 2, 13–15, 92, 95, 119
roots, 59
round, 64, 119
rowcompr, 59
savematfile, 75, 80, 119
savewave, 75
save, 75, 79, 80
scanf_conversion, 30
schur, 69, 119
sci2exp, 26, 30
select, 9, 83, 120
setenv, 8
setfield, 46, 48, 119
setlanguage, 7
sfact, 59
showprofile, 92, 105
sign, 64, 119
simp_mode, 38, 59, 60
simp, 59
sinc, 67, 119
sinhm, 68
sinh, 67, 119
sinm, 68
sin, 67, 119
size, 24, 26, 46, 48, 50, 119
spaninter, 69
spanplus, 69
sparse, 23, 70, 119
spchol, 70
spec, 69, 116
speye, 70, 119
spget, 70, 117
spones, 70, 119
sprand, 70, 119
spzeros, 70, 119
sqrtm, 68, 119
sqrt, 64, 119
st_deviation, 63, 64, 120
stacksize, 82
strcat, 25–27, 89
strcmpi, 26
strcmp, 26, 47
strcspn, 26
strindex, 26, 29, 31, 117
string, 26, 119
stripblanks, 26, 116
strncpy, 26
strrchr, 26
strrev, 26
strsubst, 26, 31, 120
strtod, 26
structure, 57
struct, 43, 46, 120
sum, 63, 64, 120
sva, 69
svd, 69, 120
sylm, 23, 59
tanhm, 68
tanh, 67, 120
tanm, 68
tan, 67, 120
testmatrix, 23
tic, 15, 16, 82, 120
timer, 15, 16, 82, 120
tlist, 18, 47, 48, 51, 53
toc, 15, 16, 82, 120
toeplitz, 23, 120
tokens, 26, 27, 120
trace, 69, 120
translatepaths, 106
tril, 120
triu, 120
try, 9, 11, 120
typename, 18
typeof, 15, 17, 18, 22, 30, 42, 57, 95, 96, 99
type, 15, 17, 18, 42, 57, 98, 99
uigetfile, 73, 81, 120
uint16, 20, 120
uint32, 20, 120
uint8, 17, 20, 21, 120
union, 26, 64, 120
unique, 26, 40, 64, 120
unix_g, 82
unix_s, 82, 88, 89
unix_w, 82, 107, 120

unix_x, 82
unix, 82
varargin, 62, 92, 94, 120
varargout, 92, 94, 120
warning, 92, 120
wavread, 74
wavwrite, 75
whereami, 92, 116
whereis, 15, 92, 120
where, 92, 116
while, 9, 120
who_user, 15, 18
whos, 15, 18, 120
who, 15, 18, 89, 108, 120
with_texmacs, 38
writb, 75
write4b, 75
write, 75–79
xset, 117
zeros, 23, 120
Scilab version, 82
scilab.ini, *see* start-up file
scilab.start, 1, 35
script, 85–90
size
list, 50
string, 24
sorting, 64–66
start-up file, 1, 2, 107–110
string
concatenation, 25, 27, 99
convert variable into, 30
convert number into, 26, 119
decomposition, 27
extraction of substrings, 28
length, 24, 25
matrix, 25
size, 24
strings , 24–34
structure, 43–47
structures, vi, 47, 52, 53, 55, 103
subfunction, 97, 98

termination script, 2, 3
TeXmacs, 38
text editor, 4, 97, 109
time, 15
timing of a code block, 15, 16
tuple assignment, 6, 49
type name, 51, 52, 99, 100
type of variable, 17
boolean matrix, 18
boolean sparse matrix, 18
compiled function, 18
constant, 18
function library, 18
implied-size vector, 18, 19
integer, 17
list, 18
Matlab sparse matrix, 18
matrix-oriented typed list, 18
pointer, 18
polynomial matrix, 18
rational, 38, 59, 60
sparse matrix, 18
string, 18
typed list, 18
un-compiled function, 18
typed list, *see* list, typed

UNIX, 83, 90

variable
convert into an expression, 30
display overloading, 100
environmental, 7, 85
global, 8, 15, 94, 95, 117
name, 4, 97
shadowed, 93
type, *see* type, variable

Windows, 83, 90
working directory, 85
workspace, vi, 2, 13, 14, 33, 79, 85, 93, 95, 107–109