An Introduction to

Scilab

from a Matlab User's Point of View

Version 2.6-1.0

Eike Rietsch

To Antje

Contents

List of Tables

Chapter 1

Introduction

For almost 10 years 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. 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. As a result, this manual is not really suited for someone who is not familiar with either Matlab or Scilab (unless he is desperate or brilliant). Documentation more suitable for a novice is available on-line or in bookstores.

Initially, I planned to organize the material ordered by Matlab functions since this was the way I approached the problem of converting Matlab functions to Scilab. However, there is not always an exact correspondence between Matlab and Scilab functions and syntax; furthermore, Scilab has features not available in Matlab, and so I reconsidered. Hence, this manual explains Scilab's functionality by drawing on the experience and expectations of a Matlab user.

To aid in the conversion of Matlab functions the table in Appendix A lists Matlab functions and their functional equivalents. Furthermore, there are three separate indexes: a general index, an index of Scilab functions and an index of Matlab functions. So one can look up quite a number of Matlab function to find out what means there are to achieve the same end in Scilab. 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 general index which points to those pages that describe this difference.

Incidentally, there is a subdirectory of a subdirectory in the Scilab directory with Scilab functions that "emulate" Matlab functions. As explained more fully in Section 2.4 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 tight`, 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-rocq.inria.fr/scilab/), 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 PC's. 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, this 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 all Matlab functions or Scilab functions. 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 Return key, are written with the name enclosed in angle brackets: <RETURN>. 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 for my own education and included what I learned.

Chapter 2

Preliminaries

2.1 Customizing Scilab for Windows

2.1.1 Startup File

Commands that should be executed at the beginning of a Scilab Session can be put in the startup file `.scilab` (the dot "." as the first character of the file name betrays the Unix heritage of Scilab). On a PC running Windows this start-up file must be in directory `bin` of the Scilab directory.

2.1.2 Fonts

Screen fonts can be set in two different ways. Either click on the Edit Button and then on `Choose Font` in the drop-down menu. Alternatively, click the right mouse button in the Scilab window and select `Choose Font`. To save the selected font, click the right mouse button in the Scilab window and select Update scilab.ini.

The Help Window comes with a proportional font preselected. However, in general a fixed-width font produces a more readable display, in particular with matrices. The fonts in the Help Window can be set by clicking the Format Button of the Help Window and selecting Font in the drop-down menu. In this case the selected font is saved automatically.

2.1.3 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 startup file `.scilab` to be run at start-up.

2.1.4 Copy and Paste

The standard Windows keyboard shortcuts for `Copy` and `Paste` do not work in the Scilab window (they do work in the `Help` window). However, the drop-down menu of the Scilab window's Edit Button has `Copy to Clipboard` and `Paste` commands. The same commands can also be found in the menu that opens up when one clicks the right mouse button in the Scilab window.

2.2 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 15). 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>). There are keyboard shortcuts for all commands in this menu.

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 2 (see the discussion of `pause` on page 15). If called from level 0 it terminates Scilab. In this case `quit` also executes a termination script, `scilab.quit`, located in the Scilab root directory (on a PC running Windows something like C:\Program Files\Scilab). This script can be modified by the user and is comparable to *`finish`* 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.

2.3 Help

The 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, in some instances, one has to type in misspelled words such as "tilda" (for "tilde") or "semicolumn" (for "semicolon").

The command `apropos`, somewhat equivalent to Matlab's *`lookfor`*, performs a string search and lists the result in a separate window. Selecting a command in this list and clicking on `OK` brings up the help window for that command. Both, `help` and `apropos`, can also be invoked from the Help Menu on the menu bar (menu items `Topic` and `Apropos`). The third item, `Help Dialog`, on the Help Button's Drop-Down Menu opens a window with two sections. One of them lists some 25 topics such as "Input/Output Functions", "Linear Algebra", "Character String Manipulation", etc. Clicking on a topic brings up, in the second section, a one-line-per-function list of relevant Scilab functions—a nice help to get started. It is particularly convenient that selecting a function and clicking the "Show" button opens a window with the help file for this function.

2.4 Emulated Matlab functions

As already mentioned in the Introduction, the Scilab distribution comes with a directory, `SCIDIR\macros\mtlb`, where `SCIDIR` denotes the Scilab root directory (in Windows something like C:\Program Files\Scilab). In this directory there are some 80 function that "emulate" Matlab functions; only four of them have help files. For several reasons I do not advocate their use. First of all, this kind of "translation" of a Matlab object to Scilab may prevent a user from fully exploiting powerful features Scilab offers. An example is `mtlb_cell` (most of the functions in the directory `mtlb` start with the prefix `mtlb_`), which emulates the Matlab function *`cell`* by means of a typed list. But there are many different ways a Matlab cell array can be expressed in Scilab. If all cell entries are strings then a string matrix is the appropriate "translation" in Scilab. Using `mtlb_cell` instead deprives one of the benefits string matrices offer (such as the overloaded `+` operator and the functionality of `length`). In other situations a ordinary list or a list of lists may be more appropriate.

A second reason for not using the functions in this directory is that careless use may lead to wrong results. An example is Matlab function *`diff`* and its Scilab "emulation" `mltb_diff`. With only one argument these two functions produce the same result; but with two arguments this is not necessarily the case since the second argument in *`diff`* serves a different purpose than in `mtlb_diff`. For example, in Matlab

```
>> diff([1:10].^2,2)
 ans =
     2     2     2     2     2     2     2     2
```

whereas in Scilab

```
-->mtlb_diff([1:10].^2,2)
 ans  =
!   8.    12.    16.    20.    24.    28.    32.    36. !
```

This just illustrates why it is better to stay away from these functions—at most use them as suggestions for implementing something in Scilab.

Chapter 3

Syntax

3.1 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 this example 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. 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 uses the first 31 characters).

```
-->a1234567890123456789012345678901234567890 = 34
a123456789012345678901234  =
    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 a <RETURN> 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 3.1: 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]        1a
 x  =
!    .0714286 !
!    .1428571 !
!    .2142857 !

-->x = 1./[1 2 3]       1b
 x  =
!    .0714286 !
!    .1428571 !
!    .2142857 !

-->x = 1 ./[1 2 3]        1c
 x  =
!   1.      .5      .3333333 !
```

Statements $\boxed{\text{1b}}$ and $\boxed{\text{1c}}$ 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 $\boxed{\text{1c}}$ where the dot is interpreted as part of the operator `./` whereas in $\boxed{\text{1b}}$ it is interpreted as the decimal point of the number 1. In Matlab, on the other hand, statements $\boxed{\text{1b}}$ and $\boxed{\text{1c}}$ produce the same result (the one produced in Scilab by $\boxed{\text{1c}}$), and $\boxed{\text{1a}}$ causes an error message. In Scilab, on the other hand, `a` is a solution of `a'*[1 2 3]' = 1`. More general, if

$$x^T A^T = b^T \tag{3.1}$$

where the superscripted T denotes transposition, then `x = b'/A'` computes the unknown x^T. Since Eq. 3.1 is equivalent to

$$Ax = b \tag{3.2}$$

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  =
!    .0714286      .1428571      .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.
```

3.2 Built-in Constants

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; ($\pi = 3.1415926\cdots$)
`%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`	*logical(1)*	Boolean variable: logical true
`%f`	*logical(0)*	Boolean variable: logical false
`%io`		Two-element vector with file identifiers for standard I/O

Table 3.2: Built-in constants

Table 3.2 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 by, for example, 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.

3.3 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 $\sim=$	not equal to

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 4.4.

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          2a
 ans  =
! T F F !

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

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

In Matlab $\boxed{\text{2a}}$ produces the same result, $\boxed{\text{2b}}$ aborts with an error message, and $\boxed{\text{2c}}$ creates the boolean vector [0 0].

3.4 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
`elseif`	*`elseif`*	Start a conditional alternative in an `if` block
`else`	*`else/otherwise`*	Start the alternative in an `if` or `select` block
`end`	*`end`*	Terminate `for`, `if`, `select`, and `while` blocks
`errcatch`	*`try`/`catch`*	Traps error and 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
`while`	*`while`*	Start repeated execution of a block while a condition is satisfied

But 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
```

we get

```
-->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 a <RETURN>. 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 a <RETURN>. The same is true for the `while` clause.

Matlab uses the *`try`*/*`catch`* syntax to trap errors. Its functionality can be emulated by means the combination of `errcatch` and `iserror`. This is illustrated in the following code fragment[1]. For the sake of clarity it is shown here the way it would look in a file.

```
errcatch(-1,'continue','nomessage'); // Start error trapping   3
a=1/0                                                           4a
if iserror()      // Check for error
  disp('A division by zero has occurred')
  errclear(-1)
end

a=1/0                                                           4b
b=1
errclear(-1)
errcatch(-1)      // Error trapping toggled off
a=1/0                                                           4c
```

Statement $\boxed{3}$ starts error trapping with the system error message suppressed. Statements $\boxed{4a}$, $\boxed{4b}$, and $\boxed{4b}$ represent errors. Execution of these statements produces the following result:

```
-->errcatch(-1,'continue','nomessage'); // Start error trapping   3

-->a=1/0                                                           4a

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

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

-->

-->a=1/0                                                           4b

-->b=1
 b  =
    1.
```

[1]`errcatch` is considered "fragile", and thus this construct should be used only for debugging; for more robust code the use of `execstr(···,'errcatch')` is preferable.

```
-->errclear(-1)

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

-->a=1/0                                                          4c
      !--error    27
division by zero...
```

Clearly, the three identical "division by zero" errors are treated differently. The first one, $\boxed{\text{4a}}$, is trapped and the user-supplied message is printed; the second, $\boxed{\text{4b}}$, is trapped and ignored; the third division by zero, $\boxed{\text{4c}}$, 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 beginning on page 15.

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.

3.5 General Functions

Table 3.3 lists Scilab's low-level functions and commands (commands are actually functions used with command-style syntax; see Section 5.1). 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 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 5a

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

Scilab	Description
$	Index of last element of matrix or (row/column) vector
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
timer	Ouputs time elapsed since the preceding call to timer()
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 function
who	Displays/outputs names of current variables
whos	Displays/outputs names and specifics of current variables

Table 3.3: General functions

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 not available anymore. This is not what we would have found with Matlab's *keyboard* command.

In order to get these new values to the original workspace they have to be passed on 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 5b

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

The above two code fragments are identical except for the return statements $\boxed{5a}$ and $\boxed{5b}$. Statement $\boxed{5b}$ returns variables a and c created in the level-1 workspace renaming a to aa. Without this change, the existing variable a would have overwritten by the value of a created in the level-1 workspace. A more complicated use of the return function is illustrated in statement $\boxed{29}$ on page 82.

The command `resume` is equivalent to the `return` command (we could have used `resume` instead of `return` in the two examples above).

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.

Chapter 4

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 other outputs a character string. The following table lists variable types and the output of `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 7.3).

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
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-like typed list (`mlist`)	17	Depends on type of list	ml
pointer	128	?	ptr
index vector with implicit size	129	'size implicit'	ip

Table 4.1: Variable types

In addition, there is a special function (see Table 4.2) for variables of type integer (see Table 4.3). For variables of type integer the function typeof outputs a character string identical to the name of the function that creates it (see Table 4.3). 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-like typed lists (mlist) is discussed in Section 4.5.

Function typeof affords a straightforward simulation of Matlab function *isa*:

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

and

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

are equivalent.

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
whos	Displays/outputs names and specifics of current variables

Table 4.2: Utility functions for managing/classifying of variables

4.1 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
```

define three 1×1 matrices, i.e. scalars. There is no function like `complex` in Scilab. Defining a complex number, such as `cx` above, is done via an arithmetic statement.

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. However, like Matlab, Scilab also knows integers. Conversion functions are shown in Table 4.3. 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 4.3: Integer conversion functions

Matlab allows no mathematical operations for such integers. Scilab is more lenient and lets the user worry about wrap-around and possibly other problems.

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

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

The result is wrapped (200-256). Unsigned integers give an analogous result.

```
-->x = uint8(100), y = uint8(2), z= uint8(3)
idxuint8
 x  =
 100
 y  =
 2
 z  =
 3

-->x*y
 ans  =
 200

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

Again, the last result is wrapped (300-256). Operations between integers of different type are not allowed, but those with 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`.

4.2 Special Matrices

Like Matlab, Scilab has a number of functions that create "standard" matrices. 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   6a
 a  =
    3.
```

whereas in Matlab

```
>>a = []+3   6b
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
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 4.4: Special matrices

While $\boxed{\text{6a}}$ is the default result, Scilab's behavior in this situation can be changed by invoking Matlab-mode.

```
-->mtlb_mode(%t)
-->a = []+3   6c
 a  =
     []
```

With Matlab-mode true, Scilab $\boxed{\text{6c}}$ produces the same result Matlab $\boxed{\text{6b}}$ 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  =
!    .2113249     .0002211     .6653811 !
!    .7560439     .3303271     .6283918 !
```

However, the syntax used in the following example

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

-->rand_mat = rand(a)     [7]
 rand_mat  =
!    .2113249     .0002211     .6653811 !
!    .7560439     .3303271     .6283918 !
```

has been deprecated in Matlab; it expects $\boxed{7}$ written as *`rand_mat = rand(size(a))`*.

4.3 Character String Variables

4.3.1 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 $\boxed{\text{8a}}$ and $\boxed{\text{8b}}$ below are equivalent.

```
-->test = "This is a test";    [8a]

-->test = 'This is a test';    [8b]
```

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. !
```

Not surprisingly, 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.

Scilab	Description
ascii	Convert ASCII codes to equivalent string and vice versa
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
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
pol2str	Convert polynomial to a string
sci2expr	Convert a variable into a string representing an expression
size	Size/dimensions of a Scilab object
strcat	Concatenate character strings
strindex(str1,str2)	Return starting index/indices where string str2 occurs in str1
string	Convert numbers to strings
stripblanks	Remove leading and trailing blanks from a string
strsubst(s1,s2,s3)	Substitute string s3 for s2 in s1
union(a,b)	Extract the unique common elements of a and b
unique(a)	Return the unique elements of a in ascending order

Table 4.5: Functions that manipulate strings

```
-->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 misses it in Matlab.

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. In fact, beginning with Scilab 2.6, it even supports the 8-bit ASCII standard (including the Euro which recently replaced the "international currency symbol"). Function `ascii` does in Scilab what the pair ***`char`*** and ***`double`*** does in Matlab.

Below is an example where `ascii` is used to emulate Matlab function ***`isletter`***.

```
function bool=isletter(str)
// Function creates a boolean vector bool the length of which is
// equal to the number of characters in string str.
// An element of bool is true if the corresponding character in
// str is a letter and false if it is not not.
// INPUT
// str    character string
// OUTPUT
// bool   boolean vector

  temp = ascii(str);
  bool = (temp >= 65 & temp <= 90) | (temp >= 97 & temp <= 122);
endfunction
```

With this function

```
-->isletter('abc123 END.')
 ans  =
! T T T F F F F T T T F !
```

Strings can be concatenated by means of the `+` sign

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

-->s2 = 'of ';

-->s3 = 'concatenation';
```

```
-->ss1 = s1+' '+s2+s3'        9a
 ss1  =
 Example of concatenation     10a

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

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

```
-->ss2 = [s1,' ',s2,s3]       9b
 ss2  =
!Example     of    concatenation  !      10b

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

In Scilab, statement $\boxed{\text{9a}}$ does what $\boxed{\text{9b}}$ would do in Matlab; in Scilab the variable `ss2` is a 4-element string vector, and Scilab's build-in function `strcat` can be used to convert string vector `ss2` to string `ss1`. Note the difference in the display of `ss1` $\boxed{\text{10a}}$ and `ss2` $\boxed{\text{10b}}$. The exclamation marks in $\boxed{\text{10b}}$ indicate that `ss2` is a string vector. In Matlab, strings in cell arrays are in quotes.

The operator `+` 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 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   !
```

With this kind of indexing 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

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

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

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 $\boxed{\text{11a}}$; str1 consists of the requested 10 characters, and $\boxed{\text{11b}}$ below shows that the last six characters of str1 are indeed blanks (ASCII code 32).

```
-->ascii(str1) 11b
 ans  =
!  116.   101.   115.   116.   32.   32.   32.   32.    32.    32. !
```

This property provides one way to emulate Matlab's function ***blanks*** and create a string of blanks by extending the empty string

```
-->blanks = part(emptystr(),1:10)  //  Create a string of 10 blanks
 blanks   =


 -->ascii(blanks)
 ans  =
!   32.    32.    32.    32.    32.    32.    32.    32.    32.    32. !
```

Scilab's stripblanks command is not quite like Matlab's ***deblank*** in that the latter only removes trailing blanks. A quick and dirty, but vectorized, implementation of deblank — at least for single strings — would be

```
function strout = deblank(str)
  // Function removes trailing blanks from input string
  index = find(ascii(str) ~= 32);
  strout = part(str,1:index($));
endfunction
```

which again uses functions part and ascii. This example again illustrates the use of $ to denote the last element of a vector — analogous to the use of *end* in Matlab as the last index. The simple modification required in the last statement to remove leading blanks (or leading and trailing blanks) is obvious.

It is worth reviewing a few more of the functions shown in Table 4.5.

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.   !
```

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 (see page 39).

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 come in handy. 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]
```

4.3.2 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

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 execerr 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).

An example of this use of execstr with the errcatch option is the simulation of a search path for the execution of a Scilab script on page 81.

In Scilab eval evaluates a vector of Scilab expressions. Thus

```
-->c = eval(['1+sin(1)';'1+cos(1)']) 
```
$\boxed{12a}$

```
 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       2 of function %eval                    called by :
line    16 of function eval                      called by :
eval('a=1+sin(1)')
 !--error     4
undefined variable : b
at line       2 of function %eval                    called by :
line    17 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, only works 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)'])
 ier =
    0.
 c  =
!   1.841471  !
!   1.5403023 !
```

$\boxed{\text{12b}}$

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*.

4.3.3 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.

4.4 Boolean Variables

Boolean variables are represented by `%t` (true) and `%f` (false). Since Scilab's main initialization file equates the corresponding upper-case and lower-case variables they can also be used with capital letters (`%T`, `%F`). This is different from Matlab where boolean variables, while intrinsically different from arithmetic numbers, are represented by 1 and 0. This analogy is illustrated in the following table which shows a line-by-line comparison of corresponding Scilab and Matlab statements.

Scilab	Matlab
`-->n = [1 1]`	`>> n = [1 1]`
` n  =`	`n =`
`!   1.     1. !`	`     1     1`
`-->b = [%t,%t]`	`>> b = logical([1,1])`
` b  =`	`b =`
`! T T !`	`     1     1`
`-->a = [1 2]`	`>> a = [1 2]`
` a  =`	`a =`
`!   1.     2. !`	`     1     2`
`-->a(n)`	`>> a(n)`
` ans   =`	`ans =`
`!   1.     1. !`	`     1     1`
`-->a(b)`	`>> a(b)`
` ans   =`	`ans =`
`!   1.     2.  !`	`     1     2`

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.

&	Logical AND
~	Logical NOT
|	Logical OR

In the proper context, both Scilab and Matlab treat numeric variables like logical 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. However, `isdef` outputs `T` if the variable exists and `F` otherwise, whereas `exists` outputs 1 and 0, respectively. In this sense the function `bool2s` 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 boolean then `b` is 1 where `a` is `%t` and 0 where `a` is `%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 `zeros` or `ones` a similar trick must 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 = ~~ones(1,10)
 true  =
! T T T T T T T T T T !
```

One could, in principle, define `true` as `true=~zeros(1,10)`, i. e. without the double negation, but in Scilab-2.6 function `zeros` is much slower than functions `ones`. This is expected to change in Scilab-2.7.

In Matlab all this would work as well but *`logical(ones(n,m))`* is faster.

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

```
-->5*%t
```

```
ans  =
    5.

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

Table 4.6 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.[1] 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')    [13]
 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 $\boxed{13}$), `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 !
```

[1]See also the discussion of `max`, `min`, etc. on page 56

Scilab	Description
MSDOS	Variable is %t if computer is PC and %f otherwise
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
isdef(a)	Test if variable a exists
iserror	Test if error has occurred
isglobal(a)	Test if a is a global variable
isinf(a)	Test if a is infinite
isnan(a)	Output boolean vector with entries %t where a is %nan
isreal(a)	Test if a is real ("or if its imaginary part is "small")
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

Table 4.6: Functions that operate on, or output, boolean variables

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
```

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 — but only for string vectors/matrices — 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(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
```

4.5 Lists

Lists are Scilab data objects and 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 structures.

4.5.1 Ordinary lists (list)

A list is a collection of data objects. Its Matlab equivalent is a vector of cells. 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'*.

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

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

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 4.7: Functions that create, or operate on, lists

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 7.3).

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());

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

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

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

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

Nevertheless,

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

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

4.5.2 Typed lists (tlist)

A typed list is a special kind of list. Typed lists allow the user to set up special kinds of data objects and to define operations on these objects (see Section 7.3). Examples are linear systems (type `'lls'`) or rational functions (type `'rational'`; see page 53).

The first element of a typed list must be a string (the type name or type) or a vector of strings. In the latter case the type name is the first element of the string vector; the other elements of this string vector are names (field names) for the other entries of the typed list. An example is

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

       a_tlist(1)
!example  first  second  !

       a_tlist(2)
    1.23
```

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

-->type(a_tlist)
ans =
16.

-->typeof(a_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.

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 as represented by the first element of the first string vector can, in principle, be ignored[2]. The other elements of the first string vector play the role of the field names of the structure. The elements of `a_tlist` can be accessed in the usual way via indices.

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

-->a_tlist(2)
 ans  =
    1.23

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

-->a_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

[2]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.

displays data objects in a more compact form and, for typed lists, is patterned after the format Matlab uses for structures. Thus

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

Section 7.3 shows how this kind of display can be made the default for displaying a particular type of a typed list.

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

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

-->a_tlist('second') 14a
 ans  =
!   1.    2. !
```

Thus the second and third element of `a_tlist(1)` can be used as “names” for the second and third element, respectively, of `a_tlist`. But there is another way of using these names. It is the representation of structures familiar to Matlab and C users.

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

-->a_tlist.second 14b
 ans  =
!   1.    2. !
```

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

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

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

One advantage of $\boxed{\text{14a}}$ over $\boxed{\text{14b}}$ 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 `a_tlist` could have been defined as

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

-->a_tlist.first = 1.23;

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

In contrast to Matlab, where the fields of a structure need not be defined before they are used, in Scilab one must define them. If `a_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);

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

-->a_tlist.new = 'value of new field';   15a ;

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

The statement $\boxed{15a}$ above could have been written as

```
a_tlist($+1) = 'value of new field';   15b
```

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.

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 reads:

```
-->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 the 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.
```

4.5.3 Matrix-oriented typed lists (mlist)

Help file and manuals provide only very sketchy information about matrix-oriented typed lists. An mlist is defined like a regular typed list discussed above. 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)   16a
!a  b  c  !

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

creates a matrix-like 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)   16b
             !--error     4
undefined variable : %l_e
```

This is in spite of the fact that $\boxed{\text{16b}}$ looks exactly like $\boxed{\text{16a}}$, the output created by function mlist. Also, the size function does not work with mlists. In practical terms, this implies more options for operator overloading. And, indeed, matrix-oriented typed lists appear to be better suited for operator overloading.

4.6 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.

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 coefficints 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 4.8: Functions related to polynomials and rational functions

```
-->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

-->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 4.8 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 5

Functions

5.1 General

For someone coming from Matlab Scilab functions are familiar entities. One difference 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-type 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

```
-->getf('fun1.sci')
-->getf 'fun1.sci'
-->getf 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-coordinate and y-coordinate 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`.

5.2 Functions that Operate on Scalars and Matrices

5.2.1 Basic Functions

Quite a number of functions in Table 5.1 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. !
```

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 ≥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 ≥ 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 5.1: Basic arithmetic functions

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

Scilab has two functions for sorting: `gsort` and `sort`. The latter is not only buggy (BOT, Chapter 6.1.3.3.6) but also less powerful. Hence, it is only `gsort` that is discussed here. 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 17
smatc =
!  4.  2.  - 1.  - 2.  !
!  3.  1.  0.  - 3.  !
```

In the help file $\boxed{17}$ 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'`).

```
-->smatri = gsort(mat,'r','i')  // Matlab-like result
 smatri  =
! - 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.

5.2.2 Elementary Mathematical Functions

Except for `cotg` the names of all the elementary transcendental functions listed in Table 5.2 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`*. Thus, for `x` $>$ `0`, `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            18a
 ans  =
!   1.     2. !
!   3.     4. !
```

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
sinh	Hyperbolic sine
tan	Tangent
tanh	Hyperbolic tangent

Table 5.2: Elementary transcendental functions

The functions listed in Table 5.3 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  =
!    .5536886 +  .4643942i     .8069607 -  .2124265i !
!   1.2104411 -  .3186397i    1.7641297 +  .1457544i !

-->b*b             18b
 ans  =
!   1. + 5.551E-17i    2. !
!   3. + 2.776E-17i    4. !
```

```
clean(b*b)        19
 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 $\boxed{19}$ illustrates how function clean can be used to remove such small matrix entries.

The important difference between these two examples is that in $\boxed{18a}$ corresponding entries of b are multiplied (the . in front of the *) whereas in $\boxed{18b}$ the matrices are multiplied.

The list of functions in Table 5.3 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 5.3: Matrix functions

5.2.3 Special Functions

Table 5.4 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, $Y_\alpha(x)$
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 5.4: Special functions

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 5.5: Functions for sparse matrices

5.2.4 Linear Algebra

Tables 5.5 and 5.6 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 5.6: Linear algebra

5.2.5 Signal-Processing Functions

Scilab proper and the Signal-Processing Toolbox offer quite a number of functions for signal processing. The three functions shown here in table 5.7 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
`fft`	Forward and inverse Fast Fourier Transform
`mfft`	Multidimensional Fast Fourier Transform

Table 5.7: Functions for signal processing

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

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

-->y = fft(x,-1);          20a

-->z = fft(y,1);           20b

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

where

$$y_m = \sum_{n=1}^{N} x_n \, e^{-2\pi i(n-1)(m-1)/N} \quad \text{for } m = 1, \cdots, N \tag{5.1}$$

with N denoting the number of elements x_n. The second argument, -1, in $\boxed{\text{20a}}$ corresponds to the minus sign in front of the exponent in (5.1). The operation performed in $\boxed{\text{20b}}$,

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

is the inverse of $\boxed{\text{20a}}$.

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 $\boxed{21}$

```
-->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);          21

-->norm(yy1-yy2)                  22
ans =
0.

-->zz = fft(yy2,1,n,1);           23

-->norm(xx-zz)                    24
ans =
3.368E-15
```

Expression $\boxed{22}$ shows that yy1 and yy2 are identical. Likewise, expression $\boxed{24}$ shows that the inverse Fourier transform $\boxed{23}$ works as expected with this syntax.

Furthermore, with yy2 computed in $\boxed{21}$, statement $\boxed{25}$ computes the two-dimensional FFT of xx:

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

-->uu2 = fft(yy2,-1,m,n);         25

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

Obviously, uu1 and uu2 are identically.

5.3 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 5.9 (reading), 5.10 (writing), and 5.8 (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.

5.3.1 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 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-2.6 for Windows. 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
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
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
xgetfile	Open dialog box for file selection

Table 5.8: Functions that open, querry, manipulate, and close files

5.3.2 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
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)
mtlb_load	Load variables from file with Matlab-4 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
save	Save current Scilab variables in binary file
wavread	Read a .wav sound file

Table 5.9: Functions that input data from files or 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).

In the examples above the file to use is identified by it 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 you must open the file yourself and use the file identifier created by mopen to specify the file to mgetl. Finally, once you are done reading, you need to close the file again.

```
fid = mopen('C:\temp\dummy.txt','r')     // Open file for reading
 fid  =
    3.
-->one = mgetl(fid,1)       // Read one line
 one  =
```

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
mtlb_save	Save variables to file in Matlab-4 format
print	Print variables to file in the format used for Scilab window
printf	Print to Scilab window (emulation of C-language printf function)
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 5.10: Functions that output data to files or to the Scilab window

```
 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. 26a

```
-->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)    27
 ans  =
! This is line 1a   Line 1b  !
!                            !
! This is line 2a   Line 2b  !
```

It is a nice feature of `matrix` that only one of the dimensions needs to be given; the other can be replaced by -1. Thus statement $\boxed{27}$ is equivalent to either of the two statements

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

The parameter not specified is computed from the dimension of the matrix to be reshaped.

5.3.3 Functions `read` and `write`

Functions `write` and `read` do what `mputl` and `mgetl` do — and more. The following statements are equivalent to those in $\boxed{26a}$ above.

```
-->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  !
-->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  =
! -  .7460990     .1023021  -  .3778182  -  .6453261   1.748736  !
! - 1.721103   - 1.2858605    2.5749104     .0116391    .1645912 !
! - 1.7157583     .6107784  -  .4575284  - 1.4344473    .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  =
! -  .7460990     .1023021  -  .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)') .51633 .64507 -.54852 -1.38505
```

```
-1.10499 1.04225 -.44840 1.13162 -1.62805 .76045 2.49761 -.72190
-1.36674 .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.

5.3.4 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.

5.3.5 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, IBM RS/6000) 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.

5.3.6 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 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 18) the arguments are displayed in reverse order.

5.3.7 Function `xgetfile`

Generally, functions whose name starts with an "x" have some connection to graphics (probably some reference to Xwindows). This is also true for `xgetfile` which opens a dialog box for interactive file selection. It works essentially like Matlab's ***`uigetfile`***. An example is

```
-->file_name = xgetfile(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.

5.4 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 do not fit well in any other topic discussed previously. Table 5.11 shows the functions I chose to put into this category.

Scilab	Description
`diary`	Write screen output of a Scilab session to a file
`getversion`	Display version of Scilab
`host`	Execute Unix/DOS command; outputs error code
`lines`	Specify number of lines to display and columns/line
`pol2tex`	LaTeX representation of a polynomial
`stacksize`	Determine/set the size of the stack
`texprint`	LaTeX representation of a Scilab object
`timer`	Ouputs time elapsed since the preceding call to `timer()`
`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 5.11: Utility functions

Of the two functions that create LaTeX code `pol2tex` appears to be superfluous since it is more specialized and `texprint` does what it does (in fact, `pol2tex` produces sets of unnecessary braces).

The LaTeXcode function `texprint` generates may require the `amsmath` package:

```
-->mat = [.2113249 .3303271 .8497452 .0683740 .7263507;
         .7560439 .6653811 .6857310 .5608486 .1985144;
         .0002211 .6283918 .8782165 .6623569 .5442573];

-->texprint(mat)
 ans  =
 {\pmatrix{ .2113249& .3303271& .8497452& .068374& .7263507\cr
           .7560439& .6653811& .685731& .5608486& .1985144\cr
           .0002211& .6283918& .8782165& .6623569& .5442573}}
```

The `diary` function causes a copy of all subsequent keyboard input and the resulting Scilab output to be copied to the file in the argument. Is more limited 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. In Scilab the diary file always creates a new file and aborts with an error message if the file already exists. Furthermore, `diary(0)` turns the recording off and closes the file. Recording cannot be toggled on and off.

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 has `unix` in it to execute DOS commands. The choice among the various functions beginning with `unix` depends on the desired output which is indicated in Table 5.11. Functions `host` and `unix` are interchangeable.

Since operating system commands are generally different for DOS and UNIX, a function that is expected to run on both must have different branches for the two. The built-in variable `MSDOS` (see Table 4.6) can be used to determine the type of operating system.

```
-->if MSDOS
-->  files=unix_g('dir D:\MyScilab\Experimental');
-->else
-->  files=unix_g('ls -l ~/MyScilab/Experimental');
-->end
-->write(%io(2),files)

 Volume in drive D has no label
 Volume Serial Number is 18F2-2730
 Directory of D:\MyScilab\Experimental

.              <DIR>        12-29-01  1:33p .
..             <DIR>        12-29-01  1:33p ..
READ_S 1 SCI        25,811  01-06-02  9:22p read_segy_file.sci
READ_S 1 M          23,103  02-21-01  8:17p read_segy_file.m
         2 file(s)         48,914 bytes
         2 dir(s)          858.80 MB free
```

Chapter 6

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) 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')
```
28a

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

```
// Test 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 explain mode parameter

-->a=1
 a  =


    1.

-->b=a+3;

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

 mode = 3

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

In this case `exec` echoes every line of the script and displays the results of every statement without a 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 explain 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 = 1` 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`.

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 for a <RETURN> after each statement.mode

Function exec satisfies the requirements of the command-style syntax (see Section 5.1). Thus $\boxed{\text{28a}}$ and $\boxed{\text{28b}}$, $\boxed{\text{28c}}$ below are equivalent statements.

```
-->exec 'script_file.sce'
```
$\boxed{\text{28b}}$

and

```
-->exec script_file.sce
```
$\boxed{\text{28c}}$

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 7.2.

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 amout of printout;
//           see help file for exec); default: mod = 1.
//
// EXAMPLES: myexec('test')
//           myexec test 1

global PATH

if PATH == []       // Set search path if it is not defined globally
   path=['D:\MyScilab\Experimental\', 'D:\MyScilab\tests\', ...
         'D:\MyScilab\General\', 'D:\MyScilab\Sci_files\'];
else
   path=PATH;
```

```
end

if argn(2) == 1     //Set default for mod, if not given
  mod=0;
end

//       Add extension to file name if necessary
tempname = filename;
lf=length(tempname);
if lf > 3
  if part(tempname,lf-3:lf) ~= '.sce'
    tempname=tempname+'.sce';
  end
else
  tempname=tempname+'.sce';
end

//      Test the filename with directories in path
for ii=1:size(path,'*');
  testfile = path(ii)+tempname;
  [fid,ierr]=file('open',testfile,'old');

  if ierr == 0
    file('close',fid)
    oldvars=who('local');     // Variables before execution of script

    exec(testfile,mod);       // Execute the script

//      Return variables created in script to the level from which
//      myexec was called
    newvars=who('local');     // Variables after execution of script
    newvars=newvars(1:size(newvars,'*')-size(oldvars,'*')-1);
    str1 = strcat((newvars+','));
    str1 = part(str1,1:length(str1)-1); // Comma-separated list of
                              // variables to return to lower level
    execstr('['+str1+']=return('+str1+')') // Return to lower level 29
  end

  if ierr ~= 240
    break
  end
end
```

```
if ierr == 240  // File not found in any of the directories
  write(%io(2),'File '+tempname+ ' 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 folder 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 it 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.

Chapter 7

User Functions

While functions in Scilab are variable 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 `...` 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. No characters other than white spaces — not even 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` which can be omitted (but would be required if `polcoord` were defined in-line).

Scilab	Description
`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
`halt`	Stop execution and wait for a key press
`mode`	Control amount of information displayed by function/script
`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`
`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
`whereami`	Display current instruction calling tree
`whereis`	Display name of library containing function

Table 7.1: 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. Even if the statement `endfunction` were omitted the function

```
function y = func1(x)   30a
  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;   31a

-->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 $\boxed{\text{31a}}$ by defining `a` as global has no effect on the result

```
-->global a,  a = 1;31b

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

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

However, if function `func1` $\boxed{30a}$ is changed to `func1g` which also includes a global statement

```
function y = func1g(x)   [30b]
  global a
  a = (a+1)^2
  y = x+a;
endfunction
```

then

```
-->global a   [31c]

-->a = 1;

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

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

The variable `a` at the end of code fragment $\boxed{31c}$ is 4, the value computed in `func1g`. If the `global a` is dropped from $\boxed{31c}$ then

```
-->a = 1;   [31d]
-->y = func1g(3)
 y  =
    7.
-->disp(a)
    1.
```

Thus $\boxed{31a}$, which uses `func1`, and $\boxed{31d}$, 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. 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
```

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 (left-hand side) since there is always the implied left-hand side `ans`. Thus `argn(2)` will never be 0.

In Scilab variables can be output 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 discuss 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 higher workspaces are available. But, upon return to the workspace above (Scilab command `resume`, all newly created variables (or any modifications of variables of the higher workspaces) are not available to this workspace.

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 input or output arguments of other functions (in Matlab on needs to pass the function name as a string and use *`feval`* to evaluate it).

7.1 Functions in Files

Scilab does not come with an integrated text editor. Thus files with Scilab functions are usually created with a text editor of the user's choosing. For Windows, PFE is a good choice, for Unix systems it is frequently `emacs` or `nedit`. The former is the default for the `edit` command (which is not implemented in the Windows version). Scilab function files generally have the extension `.sci` though, in principle, any other extension or no extension could be used as long as these files are loaded into Scilab with the `getf` command. However, some functions, like `genlib` and `getd` use the extension to identify files with Scilab functions. Hence, it is a good idea to comply with this convention.

Like in Matlab, there can be more than one function in a 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 file it has to be loaded first (this means a significant departure from the approach Matlab uses where the interpreter searches the directories of the search path for the function and loads and compiles the function in the first file encountered with the name).

A function can be loaded via the `getf('filename')` command. Here `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

```
-->getf('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 `getf` is executed the functions in the file are immediately available. This differs from the `load` command discussed below. Also, see the "gotcha" regarding `getf` on page 101.

Functions can also be collected in libraries. This is discussed in Chapter 8. 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 three reasons:

1. Variable names are shorter (24 vs 31 characters in Matlab)
2. In Matlab 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 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.

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.

7.2 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 39). 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. Thus, in the example above, `funct` is compiled. This is also proven by the fact that `funct` has type 13 (see Table 4.1). 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.

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 command window. They can also be included in Scilab scripts and functions.

7.3 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
!    .3914068     .2173720     .4883297     .4061224     .9985317 !
!    .8752304     .4418458     .9141346     .9613220     .1959695 !
!    .5266080     .9798274     .6645192     .8956145     .9872472 !
!    .9856596     .5259225     .5468820     .0717050     .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 44). It would be 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
```

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 7.2: Operator codes used to construct function names for operator overloading

Overloading the way a variable is displayed is possible because the typed list `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 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 7.2 the operator code for the minus sign is `s`. Thus

```
function seismic = %seismic_s(seismic)
// Function defines the unary negation for a seismic structure
  seismic.traces=-seismic.traces;
endfunction
```

With the seismic data set `seismic_data` defined above

```
-->seismic_data.traces(1,1)
 ans  =
     .2113249
```

```
-->seismic_data = -seismic_data;

-->seismic_data.traces(1,1)
 ans  =
  -  .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 4.1 on page 19. 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 4.1 shows that `s` is the operand type of a constant. As shown in Table 7.2, `a` is the operator code for `+` (addition). Thus

```
-->seismic_data.traces(1,1)
 ans  =
     .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.

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  =
     .2113249

-->seismic_data = abs(-seismic_data);

-->seismic_data.traces(1,1)
 ans  =
     .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  =
     .1853351

-->temp = seismic_data(5,10)

 temp  =

LIST OF TYPE "seismic"
   first: 16
    last: 16
    step: 4
  traces:  .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 objects. Furthermore, the insertion syntax and row and column concatenation can also be overloaded.

7.4 Translation of Matlab-4 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. 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 "compiled" *.bin file, and a *.cat file (help file). The latter is generated from the comment lines at the beginning of the m-file, those lines that are also used by Matlab's help facility. An example is

```
mfile2sci('D:\MyScilab\Geophysics\read_las_file.m', ...
          'D:\MyScilab\Geophysics')
```

Details about the conversion can be found in file `SCIDIR\macros\m2sci\README`; `SCIDIR` denotes the Scilab root directory (in Windows something like C:\Program Files\Scilab).

Another useful function is `translatepaths` which translates all Matlab m-files in a set of directories to Scilab. It uses `mfile2sci` to translate the individual files.

Chapter 8

Function Libraries

This is a topic that has no analog in Matlab. Libraries are collections of compiled functions that can be loaded automatically upon startup 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')
 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 (type 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 startup file `.scilab` 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
```

It is important to note, however, that 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. Hence, `lower` is not listed yet. Nevertheless, 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')
```

Chapter 9

Gotchas

This chapter deals with unexpected problems I encountered during my travails.

Function getf

Function getf reads and compiles a function from a file (see page 90). 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          |
|---------------------------------------------------------------------|
|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 for numeric matrices or string matrices produces 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
all(a)	and(a)	Output %t if all entries of the boolean matrix a are true
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

Matlab	Scilab	
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
ceil(a)	ceil(a)	Round the elements of a to the nearest integers $\geq$a
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	Current date as string
dbstack	whereami	Display current instruction calling tree
dbstack	where	Output 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	bdiag	Block diagonalization of matrix
eig	spec	Eigenvalues of matrix
ellipj	%sn	Jacobian elliptic function, sn
else	else	Start an alternative in an if or case block

Matlab	Scilab	
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, $\mathrm{erf}(x) = 2/\sqrt{\pi}\int_0^x \exp(-t^2)dt$
erfc	erfc	Complementary error function, $\mathrm{erfc}(x) = 2/\sqrt{\pi}\int_x^\infty \exp(-t^2)dt$
erfcx	erfcx	Scaled complementary error function, $\mathrm{erfcx}(x) = \exp(x^2)\mathrm{erfc}(x)$
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
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
figure	xset	Set defaults for current graphics window
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
floor(a)	floor(a)	Rounds the elements of a to the nearest integers $\geq$ a
fopen	mopen	Open a file
for	for	Start a loop with a generally known number of repetitions
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, $\ln(\Gamma(x))$
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
ifft2	fft	Forward and inverse Fast Fourier Transform
imag	imag	Imaginary part of a matrix
input	input	Prompt for user (keyboard) input

Matlab	Scilab	
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)	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
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	maxi	Maximum of all elements of a vector or array
max	max	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	min	Minimum of all elements of a vector or array
min	mini	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
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
num2str	string	Convert numbers to strings
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

Matlab	Scilab	
pinv	pinv(M)	Pseudoinverse of M
planerot	givens	Given's rotation
prod	prod	Product of the elements of a matrix
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
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
schur(M)	schur(M)	Schur decomposition
select	select	Start a multi-branch block of statements
setfield	setfield	Set a data object in a list
sign(a)	sign(a)	Signum function, $a/\|a\|$ for $a \neq 0$
sin	sin	Sine
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
std	st_deviation	Standard deviation
strrep	strsubst	Substitute one string for another in a third string
sum	sum	Sum of all elements of a matrix
svd	svd	Singular value decomposition
tan	tan	Tangent

Matlab	Scilab	
tanh	tanh	Hyperbolic tangent
tic	timer()	Ouputs time elapsed since the call to timer()
toc	timer	Ouputs time elapsed 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	errcatch	Trap error
uigetfile	xgetfile	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 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

Index

Index of Scilab Functions and Variables

Index of Matlab Functions and Variables