An Introduction to

Scilab

from a Matlab User’s Point of View

Version 2.6-1.0

Eike Rietsch
Copyright ©2001, 2002 by Eike Rietsch

Permission is granted to anyone to make or distribute verbatim copies of this document as received, in any medium, provided that the copyright notice and permission notice are preserved, and that the distributor grants the recipient permission for further redistribution as permitted by this notice.

IBM® and RS/6000® are registered trademarks of IBM Corp.
Macsyma™ is a trademark of Macsyma Inc.
Maple™ is a trademark of Waterloo Maple Inc.
Matlab™ is a trademark of The Mathworks, Inc.
Mathematica™ is a trademark of Wolfram Research, Inc.
Microsoft™ and Microsoft Windows™ are trademarks of Microsoft Corp.
Sun™ and Solaris™ are trademarks of Sun Microsystems, Inc.
UNIX® is a registered trademark of The Open Group.

Scilab© is copyrighted by INRIA, France
To Antje
Contents

1 Introduction

2 Preliminaries
   2.1 Customizing Scilab for Windows
      2.1.1 Startup File
      2.1.2 Fonts
      2.1.3 Paging
      2.1.4 Copy and Paste
   2.2 Interruption/Termination of Scripts and Scilab Session
   2.3 Help
   2.4 Emulated Matlab functions

3 Syntax
   3.1 Arithmetic Statements
   3.2 Built-in Constants
   3.3 Comparison Operators
   3.4 Flow Control
   3.5 General Functions

4 Variable Types
   4.1 Numeric Variables — Scalars and Matrices
   4.2 Special Matrices
   4.3 Character String Variables
      4.3.1 Creation and Manipulation of Character Strings
      4.3.2 Strings with Scilab Expressions
      4.3.3 Symbolic String Manipulation
   4.4 Boolean Variables
   4.5 Lists
      4.5.1 Ordinary lists (list)
      4.5.2 Typed lists (tlist)
      4.5.3 Matrix-oriented typed lists (mlist)
   4.6 Polynomials
CONTENTS

5 Functions ............................ 54
  5.1 General ........................................ 54
  5.2 Functions that Operate on Scalars and Matrices ...................... 56
    5.2.1 Basic Functions .......................... 56
    5.2.2 Elementary Mathematical Functions .................... 59
    5.2.3 Special Functions ............................ 62
    5.2.4 Linear Algebra ............................. 63
    5.2.5 Signal-Processing Functions ....................... 64
  5.3 File Input and Output ................................ 66
    5.3.1 Opening and Closing of Files .................... 66
    5.3.2 Functions mgetl and mputl ..................... 67
    5.3.3 Functions read and write ....................... 70
    5.3.4 Functions load and save ....................... 73
    5.3.5 Functions mput and mget/mgeti .................... 74
    5.3.6 Functions input and disp ....................... 75
    5.3.7 Function xgetfile ............................ 75
  5.4 Utility Functions ................................ 76

6 Scripts .................................... 78

7 User Functions ................................ 84
  7.1 Functions in Files .................................. 90
  7.2 In-line Functions .................................. 91
  7.3 Functions for operator overloading ....................... 92
  7.4 Translation of Matlab-4 m-files to Scilab Format ................ 97

8 Function Libraries ............................ 98

9 Gotchas ..................................... 101

A Matlab functions and their Scilab Equivalents .......................... 102
# List of Tables

3.1 List of arithmetic operators ................................................................. 8  
3.2 Built-in constants ............................................................................. 10  
3.3 General functions .............................................................................. 16  

4.1 Variable types ...................................................................................... 19  
4.2 Utility functions for managing/classifying of variables ....................... 20  
4.3 Integer conversion functions .............................................................. 22  
4.4 Special matrices .................................................................................. 24  
4.5 Functions that manipulate strings ...................................................... 26  
4.6 Functions that operate on, or output, boolean variables ..................... 38  
4.7 Functions that create, or operate on, lists .......................................... 40  
4.8 Functions related to polynomials and rational functions ..................... 51  

5.1 Basic arithmetic functions ................................................................... 57  
5.2 Elementary transcendental functions .................................................. 60  
5.3 Matrix functions .................................................................................. 61  
5.4 Special functions ................................................................................ 62  
5.5 Functions for sparse matrices .............................................................. 62  
5.6 Linear algebra ..................................................................................... 63  
5.7 Functions for signal processing ............................................................ 64  
5.8 Functions that open, query, manipulate, and close files ..................... 67  
5.9 Functions that input data from files or keyboard ................................ 68  
5.10 Functions that output data to files or to the Scilab window ............... 69  
5.11 Utility functions ................................................................................ 76  

7.1 Functions/commands/keywords relevant for user functions ................ 85  
7.2 Operator codes used to construct function names for operator overloading 94
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.
CHAPTER 1. INTRODUCTION

It was tempting to use unadulterated screen dumps as examples. However Scilab wastes screen real
estate the same way \texttt{format loose} does in Matlab — except, in Scilab, there is no equivalent to
\texttt{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.
\textit{Introduction To Scilab - Users Guide} by the Scilab Group
\textit{Une Introduction à Scilab} by Bruno Pinçon
\textit{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:
\textbf{Red typewriter font is used for Scilab commands, functions, variables, ...}
\textbf{Blue slanted typewriter font is used for Matlab commands, functions,
variables, ...}
\textbf{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:
\texttt{\textlangle\textgreater\texttt{RETURN}}. In the section on operator overloading angle brackets are also used to enclose
operand types and operator codes.

\textbf{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 \texttt{.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 \texttt{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 \texttt{Choose Font} in the drop-down menu. Alternatively, click the right mouse button in the Scilab window and select \texttt{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 \texttt{[More (y or n )?] } is displayed. The number of lines displayed can be controlled via the \texttt{lines} command. Paging (and that message) can be suppressed by means of the command \texttt{lines(0)}. If this appears desirable, this command can be put in the startup file \texttt{.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 “tilde” (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” mtlb_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
a12345678901234567890123 =
  34.
```
Variable names are case-sensitive (i.e., Scilab distinguishes between upper-case and lower-case letters). A semicolon (;) terminating a statement indicates that the result should not be displayed whereas a comma or 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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>−</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>.*</td>
<td>Array multiplication</td>
</tr>
<tr>
<td>.*</td>
<td>Kronecker multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>\</td>
<td>Left matrix division</td>
</tr>
<tr>
<td>./</td>
<td>Array division</td>
</tr>
<tr>
<td>.\</td>
<td>Left array division</td>
</tr>
<tr>
<td>./</td>
<td>Kronecker division</td>
</tr>
<tr>
<td>./</td>
<td>Kronecker left division</td>
</tr>
<tr>
<td>^ or **</td>
<td>Matrix exponentiation</td>
</tr>
<tr>
<td>.^</td>
<td>Array exponentiation</td>
</tr>
<tr>
<td>′</td>
<td>Matrix complex transposition</td>
</tr>
<tr>
<td>.′</td>
<td>Array transposition</td>
</tr>
</tbody>
</table>

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 !
```
3.1. ARITHMETIC STATEMENTS

\[
! \quad 0.2142857 \\
\]

\[
\text{--->} x = 1 ./ [1 2 3] \quad \text{(1c)}
\]

\[
x = \text{! 1. .5 .3333333 !}
\]

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

\[
x^T A^T = b^T \quad (3.1)
\]

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

\[
A x = b \quad (3.2)
\]

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

\[
\text{--->} x = [1 2 3]' \backslash 1
\]

\[
x = \text{! 0.0714286 .1428571 .2142857 !}
\]

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

\[
\text{--->} [u,v,w] = (1,2,3)
\]

\[
w = 3. \\
v = 2. \\
u = 1.
\]

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 \textit{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
\[
\rightarrow [u,v] = (v,u) \\
v = 1. \\
u = 2.
\]

### 3.2 Built-in Constants

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Matlab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%i</td>
<td>i, j</td>
<td>Imaginary unit ((\sqrt{-1}))</td>
</tr>
<tr>
<td>%e</td>
<td>e</td>
<td>Euler’s constant ((e \approx 2.7182818\ldots))</td>
</tr>
<tr>
<td>%pi</td>
<td>pi</td>
<td>Ratio of circumference to diameter of a circle; ((\pi \approx 3.1415926\ldots))</td>
</tr>
<tr>
<td>%eps</td>
<td>eps</td>
<td>Machine (\epsilon \approx 2.2 \cdot 10^{-16}); smallest number such that 1 + (\epsilon) &gt; 1</td>
</tr>
<tr>
<td>%inf</td>
<td>inf</td>
<td>Infinity ((\infty))</td>
</tr>
<tr>
<td>%nan</td>
<td>NaN</td>
<td>Not a number</td>
</tr>
<tr>
<td>%s</td>
<td></td>
<td>Polynomial (s=\text{poly}(0,'s'))</td>
</tr>
<tr>
<td>%z</td>
<td></td>
<td>Polynomial (z=\text{poly}(0,'z'))</td>
</tr>
<tr>
<td>%t</td>
<td>logical(1)</td>
<td>Boolean variable: logical true</td>
</tr>
<tr>
<td>%f</td>
<td>logical(0)</td>
<td>Boolean variable: logical false</td>
</tr>
<tr>
<td>%io</td>
<td></td>
<td>Two-element vector with file identifiers for standard I/O</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;&gt; or ~</td>
<td>not equal to</td>
</tr>
</tbody>
</table>

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

```plaintext
--->[1 2 3] == 1
ans =
! T F F !
```

```plaintext
--->[1 2 3] == [1 2]
ans =
F
```

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

In Matlab \[2a\] produces the same result, \[2b\] aborts with an error message, and \[2c\] creates the boolean vector \[0 0\].
3.4 Flow Control

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

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Matlab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>break</td>
<td>Force exit from a loop</td>
</tr>
<tr>
<td>case</td>
<td>case</td>
<td>Start clause within a select block</td>
</tr>
<tr>
<td>elseif</td>
<td>elseif</td>
<td>Start a conditional alternative in an if block</td>
</tr>
<tr>
<td>else</td>
<td>else/otherwise</td>
<td>Start the alternative in an if or select block</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
<td>Terminate for, if, select, and while blocks</td>
</tr>
<tr>
<td>errcatch</td>
<td>try/catch</td>
<td>Traps error and with several possible actions</td>
</tr>
<tr>
<td>for</td>
<td>for</td>
<td>Start a loop with a generally known number of repetitions</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td>Start a conditionally executed block of statements</td>
</tr>
<tr>
<td>select</td>
<td>switch</td>
<td>Start a multi-branch block of statements</td>
</tr>
<tr>
<td>while</td>
<td>while</td>
<td>Start repeated execution of a block while a condition is satisfied</td>
</tr>
</tbody>
</table>

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:

```plaintext
function foobar(a)
  // Scilab
  select a
  case ['foo','pipo']
    disp('ok')
  case 'foo'
    disp('not ok')
  elseif
    disp('invalid case')
  end
endfunction
```

we get

```plaintext
--> 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

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

on the other hand, leads to

```matlab
>> 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

```matlab
--> 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

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

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

```matlab
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.

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

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

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

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

-->a=1/0
-->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.
3.5. GENERAL FUNCTIONS

```plaintext
-->errclear(-1)

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

-->a=1/0

  !--error 27
  division by zero...
```

Clearly, the three identical “division by zero” errors are treated differently. The first one, \( 4a \), is trapped and the user-supplied message is printed; the second, \( 4b \), is trapped and ignored; the third division by zero, \( 4c \), 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:

```plaintext
-->a = 1, b = 2    // Variables in original workspace
    a =
    1.  
    b =
    2. 
```
CHAPTER 3. SYNTAX

-->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

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>Index of last element of matrix or (row/column) vector</td>
</tr>
<tr>
<td>apropos</td>
<td>Keyword search for a function</td>
</tr>
<tr>
<td>clear</td>
<td>Clear unprotected variables and functions from memory</td>
</tr>
<tr>
<td>clerglobal</td>
<td>Clear global variables from memory</td>
</tr>
<tr>
<td>date</td>
<td>Current date as string</td>
</tr>
<tr>
<td>disp</td>
<td>Display input argument</td>
</tr>
<tr>
<td>getdate</td>
<td>Get date and time in an 8-element numeric vector</td>
</tr>
<tr>
<td>global</td>
<td>Define variables as global</td>
</tr>
<tr>
<td>halt</td>
<td>Stop execution and wait for a key press</td>
</tr>
<tr>
<td>help</td>
<td>On-line help</td>
</tr>
<tr>
<td>inttype(a)</td>
<td>Output numeric code representing type of integer a</td>
</tr>
<tr>
<td>pause</td>
<td>Interrupt execution of function or script</td>
</tr>
<tr>
<td>timer</td>
<td>Outputs time elapsed since the preceding call to timer()</td>
</tr>
<tr>
<td>type(a)</td>
<td>Output numeric code representing type of variable a</td>
</tr>
<tr>
<td>typeof(a)</td>
<td>Output string with type of variable a</td>
</tr>
<tr>
<td>whereis</td>
<td>Display name of library containing function</td>
</tr>
<tr>
<td>who</td>
<td>Displays/outputs names of current variables</td>
</tr>
<tr>
<td>whos</td>
<td>Displays/outputs names and specifics of current variables</td>
</tr>
</tbody>
</table>

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!

```plaintext
--> 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 [5a] and [5b]. Statement [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 [29] on page 82.
The command \texttt{resume} is equivalent to the \texttt{return} command (we could have used \texttt{resume} instead of \texttt{return} in the two examples above).

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

\begin{verbatim}
-->disp(123,'The result is:')
The result is:
  123.
\end{verbatim}

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, isssparse, 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).

<table>
<thead>
<tr>
<th>Type of variable</th>
<th>type</th>
<th>typeof</th>
<th>Op-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>real or complex constant matrix</td>
<td>1</td>
<td>’constant’</td>
<td>s</td>
</tr>
<tr>
<td>polynomial matrix</td>
<td>2</td>
<td>’polynomial’</td>
<td>p</td>
</tr>
<tr>
<td>boolean matrix</td>
<td>4</td>
<td>’boolean’</td>
<td>b</td>
</tr>
<tr>
<td>sparse matrix</td>
<td>5</td>
<td>’sparse’</td>
<td>sp</td>
</tr>
<tr>
<td>sparse boolean matrix</td>
<td>6</td>
<td>’boolean sparse’</td>
<td>spb</td>
</tr>
<tr>
<td>Matlab sparse matrix</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrix of integers stored in 1, 2, or 4 bytes</td>
<td>8</td>
<td>Depends on type of integer</td>
<td>i</td>
</tr>
<tr>
<td>matrix of character strings</td>
<td>10</td>
<td>’string’</td>
<td>c</td>
</tr>
<tr>
<td>function (un-compiled)</td>
<td>11</td>
<td>’function’</td>
<td>m</td>
</tr>
<tr>
<td>function (compiled)</td>
<td>13</td>
<td>’function’</td>
<td>mc</td>
</tr>
<tr>
<td>function library</td>
<td>14</td>
<td>’library’</td>
<td>f</td>
</tr>
<tr>
<td>list</td>
<td>15</td>
<td>’list’</td>
<td>l</td>
</tr>
<tr>
<td>typed list (tlist)</td>
<td>16</td>
<td>Depends on type of list</td>
<td>tl</td>
</tr>
<tr>
<td>matrix-like typed list (mlist)</td>
<td>17</td>
<td>Depends on type of list</td>
<td>ml</td>
</tr>
<tr>
<td>pointer</td>
<td>128</td>
<td></td>
<td>ptr</td>
</tr>
<tr>
<td>index vector with implicit size</td>
<td>129</td>
<td>’size implicit’</td>
<td>ip</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intype(a)</td>
<td>Output numeric code representing type of integer a</td>
</tr>
<tr>
<td>type(a)</td>
<td>Output numeric code representing type of variable a</td>
</tr>
<tr>
<td>typename</td>
<td>Associate a variable type name with a numeric avariable type</td>
</tr>
<tr>
<td>typeof(a)</td>
<td>Output string with type of variable a</td>
</tr>
<tr>
<td>who</td>
<td>Displays/outputs names of current variables</td>
</tr>
<tr>
<td>whos</td>
<td>Displays/outputs names and specifics of current variables</td>
</tr>
</tbody>
</table>

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

```scilab
-->a = 1.2;
-->b = 1.0e3;
-->cx = a+%i*b
    cx =
        1.2 + 1000.i
```

define three $1 \times 1$ 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.

```scilab
-->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 \texttt{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 \texttt{‘size implicit’}.

\begin{verbatim}
-->index=2:1:
index = 
2:1:

-->type(index)
ans =
129.

-->typeof(index)
ans =
size implicit

-->mat2(2,index)
ans =
! 5. 6. !
\end{verbatim}

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 \texttt{iconvert}, which takes two input arguments, does essentially what the seven other conversion functions listed in this table do.

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{double}</td>
<td>Convert integer array of any type/length to floating point array</td>
</tr>
<tr>
<td>\texttt{iconvert}</td>
<td>Convert numeric array to any integer or floating point format</td>
</tr>
<tr>
<td>\texttt{int8(a)}</td>
<td>Convert ( a ) to an 8-bit signed integer</td>
</tr>
<tr>
<td>\texttt{int16(a)}</td>
<td>Convert ( a ) to a 16-bit signed integer</td>
</tr>
<tr>
<td>\texttt{int32(a)}</td>
<td>Convert ( a ) to a 32-bit signed integer</td>
</tr>
<tr>
<td>\texttt{uint8(a)}</td>
<td>Convert ( a ) to an 8-bit unsigned integer</td>
</tr>
<tr>
<td>\texttt{uint16(a)}</td>
<td>Convert ( a ) to a 16-bit unsigned integer</td>
</tr>
<tr>
<td>\texttt{uint32(a)}</td>
<td>Convert ( a ) to a 32-bit unsigned integer</td>
</tr>
</tbody>
</table>

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.
4.1. NUMERIC VARIABLES — SCALARS AND MATRICES

-->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 \( \text{typeof} \) returns the value \( \text{constant} \).
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,

\[
\begin{align*}
\text{-->} & \text{a = [\ ]+3} & 6a \\
& \text{a =} & 3. \\
\end{align*}
\]

whereas in Matlab

\[
\begin{align*}
\text{>>} & \text{a = [\ ]+3} & 6b \\
& \text{a =} & [\ ].
\end{align*}
\]

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

Table 4.4: Special matrices

While \(6a\) is the default result, Scilab’s behavior in this situation can be changed by invoking Matlab-mode.

\[
\begin{align*}
\text{-->} & \text{mtlb\_mode(\%t)} \\
\text{-->} & \text{a = [\ ]+3} & 6c \\
& \text{a =} & [\]
\end{align*}
\]
4.3. CHARACTER STRING VARIABLES

With Matlab-mode true, Scilab produces the same result Matlab 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)  
rand_mat =
! .2113249 .0002211 .6653811 !
! .7560439 .3303271 .6283918 !
```

has been deprecated in Matlab; it expects 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 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 \texttt{size} would give in Matlab if \texttt{test} were a Matlab cell. Thus it is not surprising that strings can be elements of matrices.

\begin{verbatim}
-->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. !
\end{verbatim}

Not surprisingly, function \texttt{size} again gives the same result \texttt{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 \texttt{length}; nevertheless, it is a straight-forward generalization of its behavior for an individual string.

\begin{table}[h]
\centering
\begin{tabular}{|l|p{10cm}|}
\hline
\textbf{Scilab} & \textbf{Description} \\
\hline
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 \texttt{a} \\
intersect(str1,str2) & Returns elements common to two vectors \texttt{str1} and \texttt{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 \texttt{str2} occurs in \texttt{str1} \\
string & Convert numbers to strings \\
stripblanks & Remove leading and trailing blanks from a string \\
strsubst(s1,s2,s3) & Substitute string \texttt{s3} for \texttt{s2} in \texttt{s1} \\
union(a,b) & Extract the unique common elements of \texttt{a} and \texttt{b} \\
unique(a) & Return the unique elements of \texttt{a} in ascending order \\
\hline
\end{tabular}
\caption{Functions that manipulate strings}
\end{table}
--> 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`.

```matlab
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

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

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

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

--> s2 = 'of ';

--> s3 = 'concatenation';
```
-->ss1 = s1 + ' ' + s2 + s3'  
ss1 =  
Example of concatenation  

-->length(ss1)
ans = 24.

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

-->ss2 = [s1, ' ', s2, s3]
ss2 =  
!Example of concatenation !

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

In Scilab, statement 9a does what 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 10a and ss2 10b. The exclamation marks in 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
4.3. **CHARACTER STRING VARIABLES**

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 \[11a\]; `str1` consists of the requested 10 characters, and \[11b\] 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

```plaintext
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.

```matlab
-->length(emptystr(2,5))
ans =
  1 0 0 0 0
  1 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.

```matlab
-->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.

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

-->typeof(a)
anstype = string

-->typeof(b)
anstype = 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

```plaintext
-->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

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Matlab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval</td>
<td>eval</td>
<td>Evaluate string vector with Scilab expressions</td>
</tr>
<tr>
<td>evstr</td>
<td>eval</td>
<td>Evaluate string vector with Scilab expressions</td>
</tr>
<tr>
<td>execstr</td>
<td>eval</td>
<td>Evaluate string vector with Scilab expressions or statements</td>
</tr>
</tbody>
</table>

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

```scilab
-->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

```scilab
-->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

```scilab
-->c = eval([ '1+sin(1)'; '1+cos(1)'] )
```

4.3. CHARACTER STRING VARIABLES

\[
c = \\
! 1.841471 ! \\
! 1.5403023 !
\]

Note that \texttt{eval(’a=1+sin(1)’)} produces the error message

\begin{verbatim}
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)’)
\end{verbatim}

Scilab expects an expression and interprets the \texttt{=} as a typo, assumes that the user really means \texttt{==}, and then finds that \texttt{b} is undefined.

The Scilab command \texttt{evstr} is very similar to \texttt{eval}; it, too, only works with expressions. However, while \texttt{eval} has no provisions to allow user-defined error handling, \texttt{evstr} will trap errors if used with two output arguments.

\begin{verbatim}
-->[c,ier] = evstr([’1+sin(1)’;’1+cos(1)’]) 12b
  ier =
    0.
  c =
! 1.841471 !
! 1.5403023 !
\end{verbatim}

If an error occurs, \texttt{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.

\begin{verbatim}
-->[c,ier] = evstr([’1+sin(1)’;’1+-cos(1)’])
  ier =
    2.
  c =
[]
\end{verbatim}

The function does not abort, but \texttt{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 \texttt{evalin}.
4.3.3 Symbolic String Manipulation

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

```plaintext
-->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.

```plaintext
-->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

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.

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Matlab</th>
</tr>
</thead>
<tbody>
<tr>
<td>--&gt;n = [1 1]</td>
<td>&gt;&gt; n = [1 1]</td>
</tr>
<tr>
<td>n = ! 1. 1. !</td>
<td>n = 1 1</td>
</tr>
<tr>
<td>--&gt;b = [%t,%t]</td>
<td>&gt;&gt; b = logical([1,1])</td>
</tr>
<tr>
<td>b = ! T T !</td>
<td>b = 1 1</td>
</tr>
<tr>
<td>--&gt;a = [1 2]</td>
<td>&gt;&gt; a = [1 2]</td>
</tr>
<tr>
<td>a = ! 1. 2. !</td>
<td>a = 1 2</td>
</tr>
<tr>
<td>--&gt;a(n)</td>
<td>&gt;&gt; a(n)</td>
</tr>
<tr>
<td>ans = ! 1. 1. !</td>
<td>ans = 1 1</td>
</tr>
<tr>
<td>--&gt;a(b)</td>
<td>&gt;&gt; a(b)</td>
</tr>
<tr>
<td>ans = ! 1. 2. !</td>
<td>ans = 1 2</td>
</tr>
</tbody>
</table>

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

```plaintext
-->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 \texttt{exists} and \texttt{isdef} do the same thing: they check if a variable exists. However, \texttt{isdef} outputs \texttt{T} if the variable exists and \texttt{F} otherwise, whereas \texttt{exists} outputs 1 and 0, respectively. In this sense the function \texttt{bool2s} can be considered as having boolean output. If \texttt{a} is a numeric matrix, then \texttt{b = bool2s(a)} creates a matrix \texttt{b} where all non-zero entries of \texttt{a} are 1. If \texttt{a} is boolean then \texttt{b} is 1 where \texttt{a} is \texttt{%t} and 0 where \texttt{a} is \texttt{%f}. The same result — even without an execution-time penalty — can be achieved by adding 0 to the boolean matrix \texttt{a}.

Since there is no Scilab function analogous to \texttt{zeros} or \texttt{ones} a similar trick must be used to create a boolean matrix or vector.

```plaintext
-->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 \texttt{true} as \texttt{true=~zeros(1,10)}, i. e. without the double negation, but in Scilab-2.6 function \texttt{zeros} is much slower than functions \texttt{ones}. This is expected to change in Scilab-2.7.

In Matlab all this would work as well but \texttt{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.

```plaintext
-->5*%t
```
4.4. BOOLEAN VARIABLES

\[
\text{ans} = 5.
\]

\[
\text{-->3*%f}
\]
\[
\text{ans} = 0.
\]

Table 4.6 lists a number of functions that output or use boolean variables.

Functions \texttt{and} and \texttt{or} are functional equivalents of Matlab’s functions \texttt{all} and \texttt{any}, respectively.\footnote{See also the discussion of \texttt{max}, \texttt{min}, etc. on page 56}

Function \texttt{and(a)} returns the boolean variable \%t if all entries of \texttt{a} are \%t (for a boolean matrix) or non-zero (for a numeric matrix).

\[
\text{a} =
\begin{array}{ccc}
! & 0. & 1. ! \\
! & 2. & 3. ! \\
\end{array}
\]

\[
\text{-->and(a)}
\]
\[
\text{ans} = \text{F}
\]

\[
\text{-->and(a,'r')} \quad 13
\]
\[
\text{ans} =
\begin{array}{c}
! \text{F} \text{ T} ! \\
\end{array}
\]

In the example above \texttt{a} has a zero entry in the upper left corner; hence, the answer is \texttt{false}. With the optional second argument ‘\texttt{r}’ (line 13), \texttt{and} analyzes the columns of \texttt{a} and outputs a row vector. The first column contains a zero; hence the first element of the output vector is \texttt{f}. Matlab’s \texttt{all} would output an analogous logical vector.

Function \texttt{or(a)} returns the boolean variable \%t if at least one entry of \texttt{a} is \%t (for a boolean matrix) or non-zero (for a numeric matrix). Hence, for the same matrix \texttt{a}

\[
\text{-->or(a)}
\]
\[
\text{ans} = \text{T}
\]

\[
\text{-->or(a,'c')} \quad 13
\]
\[
\text{ans} =
\begin{array}{c}
! \text{T} ! \\
! \text{T} ! \\
\end{array}
\]
### CHAPTER 4. VARIABLE TYPES

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSDOS</td>
<td>Variable is ( %t ) if computer is PC and ( %f ) otherwise</td>
</tr>
<tr>
<td>and(a)</td>
<td>Output ( %t ) if all entries of the boolean matrix ( a ) are true</td>
</tr>
<tr>
<td>bool2s</td>
<td>Replace ( %t ) (or non-zero entry) in matrix by 1 and ( %f ) by zero</td>
</tr>
<tr>
<td>exists(a)</td>
<td>Test if variable ( a ) exists</td>
</tr>
<tr>
<td>find(a)</td>
<td>Find the indices for which boolean matrix ( a ) is true</td>
</tr>
<tr>
<td>isdef(a)</td>
<td>Test if variable ( a ) exists</td>
</tr>
<tr>
<td>iserror</td>
<td>Test if error has occurred</td>
</tr>
<tr>
<td>isglobal(a)</td>
<td>Test if ( a ) is a global variable</td>
</tr>
<tr>
<td>isnan(a)</td>
<td>Output boolean vector with entries ( %t ) where ( a ) is ( %\text{nan} )</td>
</tr>
<tr>
<td>isreal(a)</td>
<td>Test if ( a ) is real (“or if its imaginary part is “small”)</td>
</tr>
<tr>
<td>mtlb_mode</td>
<td>Test for (or set) Matlab mode for empty matrices</td>
</tr>
<tr>
<td>or(a)</td>
<td>Output ( %t ) if at least one entry of the boolean matrix ( a ) is true</td>
</tr>
<tr>
<td>simp_mode</td>
<td>Test for (or set) simplification mode for rational expressions</td>
</tr>
</tbody>
</table>

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

With the second argument ‘c’ function \texttt{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 \texttt{any} is \texttt{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

\[
\text{bool} = \text{exists}('a') \& a > 0
\]

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

\[
\text{bool} = \text{exists}('a')
\]
\[
\text{if } \text{bool}
\]
\[
\text{bool} = a > 0;
\]
\[
\text{end}
\]

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 \texttt{ismember}. For example, the Matlab statements

\[
\text{>>vstr} = \{\text{'abcd', 'abc', 'xyz', 'uvwx'}\};
\]
\[
\text{>>str} = \text{'abc'};
\]
\[
\text{>>index} = \text{ismember(vstr,str)}
\]
\[
\text{index} =
\]
\[
\begin{array}{cccc}
0 & 1 & 0 & 0 \\
\end{array}
\]
produce the same result as the analogous Scilab statements

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

if the function `ismember` is defined as

```scilab
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):
Individual elements can be accessed with the usual index notation. Thus

```plaintext
--> 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   !
```

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

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

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

Table 4.7: Functions that create, or operate on, lists
This pushes all elements of \texttt{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

```
\texttt{a\_list=\{\texttt{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.

```
\texttt{a\_list1=\{\texttt{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 \texttt{’final element’} to element 8 of the list \texttt{a\_list}. Elements 5 to 7 are undefined. Thus

```
-->a\_list(5)
!--error 117 List element 5 is Undefined
```

The \texttt{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 =
a(1)
 1.
a(2)
 2.
a(3)
 4.
a(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 \texttt{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 \texttt{aa} defined above
\[\text{--->[u,v]=} \text{aa}(2:3)\]
\[\begin{align*}
u &= 4. \\
u &= 2. \\
\end{align*}\]

This kind of tuple assignment can also be used with typed lists.

The functions \texttt{size} and \texttt{length} have been appropriately overloaded for lists.

\[\text{--->blist = list(’abcd’, ’efg’, 1.3, [1 2; 3 4], list(’1’, 1))}\]
\[\text{blist } = \]
\[\text{blist(1)} \]
\[\text{abcd}\]
\[\text{blist(2)} \]
\[\text{efg}\]
\[\text{blist(3)} \]
\[1.3\]
\[\text{blist(4)} \]
\[\begin{bmatrix}
1. & 2. \\
3. & 4. \\
\end{bmatrix}\]
\[\text{blist(5)} \]
\[\text{blist(5)(1)} \]
\[1\]
\[\text{blist(5)(2)} \]
\[1.\]

\[\text{--->length(blist)}\]
\[\text{ans } = \]
\[5.\]

\[\text{--->size(blist)}\]
\[\text{ans } = \]
\[5.\]

Note that \texttt{length} and \texttt{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 alphanumerical entries while others are purely numeric). One can write it as a list of lists. The following is an example.

```markdown
-->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,

```markdown
-->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

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

   a_tlist(1)
!example first second !

   a_tlist(2)
1.23
```
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.

\[
\begin{align*}
\text{--a_tlist(1)} \\
\text{ans =} \\
\text{!example first second !}
\end{align*}
\]

\[
\begin{align*}
\text{--a_tlist(2)} \\
\text{ans =} \\
1.23
\end{align*}
\]

\[
\begin{align*}
\text{--a_tlist(3)} \\
\text{ans =} \\
\text{! 1. 2. !}
\end{align*}
\]

\[
\begin{align*}
\text{--a_tlist(1)(2)} \\
\text{ans =} \\
\text{first}
\end{align*}
\]

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')
an=
  1.23

--> a_tlist('second') \[14a\]
an=

  ! 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
an=
  1.23

--> a_tlist.second \[14b\]
an=

  ! 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
an=
  A new value
```

One advantage of \[14a\] over \[14b\] 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

```plaintext
-->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);

```plaintext
-->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 `\[15a]\` above could have been written as

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

Generally speaking, the $k$th element of the first-element character string vector of a typed list is the field name of the $k$th element of the typed list.

Lists can have other lists as elements. For example

```plaintext
-->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.
CHAPTER 4. VARIABLE TYPES

-->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
It is also straightforward to access fields of individual data sets. For example,

```matlab
-->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

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

an_mlist(1)
!VVV name value !

an_mlist(2)
!a b c !

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

creates a matrix-like typed list, and the statements

```matlab
-->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 16b looks exactly like 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)
anse = polynomial

-->typeof(p)
anse = 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.
### 4.6. POLYNOMIALS

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

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)
an =
   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)
an =
  2 - s
------
4.6. POLYNOMIALS

The result of \texttt{type} indicates that \( r \) is a typed list and \texttt{typeof} tells us that it is a list of type \texttt{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,

```plaintext
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

```plaintext
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

```plaintext
-->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

```plaintext
-->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,

```plaintext
-->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

```plaintext
-->plot2d('ll',x,y,leg='Curve legend')
```
In principle, any input argument could be supplied as a named parameter.

\[
\text{-->plot2d(x=x,y=y,leg='Curve legend')}
\]

but \text{plot2d} has internal checks that do not allow that. Also, named parameters are not compatible with variable-length input argument lists \text{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.

\[
\text{-->mat = matrix([1:20],4,5) // Create a matrix by rearranging a vector}
\]
\[
\text{mat =}
\begin{array}{cccc}
1 & 5 & 9 & 13 \\
2 & 6 & 10 & 14 \\
3 & 7 & 11 & 15 \\
4 & 8 & 12 & 16 & 20
\end{array}
\]

\[
\text{-->[maxa,index] = max(mat) // Find largest element and its location}
\]
\[
\text{index =}
\begin{array}{c}
4 \\
4 \\
4 \\
4
\end{array}
\]
\[
\text{maxa = 20.}
\]

\[
\text{-->[maxr,idx] = max(mat,'r')}
\]
\[
\text{idx =}
\begin{array}{cccc}
4 & 4 & 4 & 4 & 4
\end{array}
\]
\[
\text{maxr =}
\begin{array}{cccc}
4 & 8 & 12 & 16 & 20
\end{array}
\]

\[
\text{-->maxc = max(mat,’c’)}
\]
\[
\text{maxc =}
\begin{array}{cccc}
17 & 18 & 19 & 20
\end{array}
\]
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(a)</td>
<td>Absolute value of ( a,</td>
</tr>
<tr>
<td>bool2s</td>
<td>Replace %t (or non-zero entry) in matrix by 1 and %f by zero</td>
</tr>
<tr>
<td>ceil(a)</td>
<td>Round the elements of ( a ) to the nearest integers ( \geq a )</td>
</tr>
<tr>
<td>clean</td>
<td>“Clean” matrices; i.e. small entries are set to zero</td>
</tr>
<tr>
<td>conj</td>
<td>Complex conjugate</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product of all elements of a vector or array</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum of all elements of a vector or array</td>
</tr>
<tr>
<td>fix(a)</td>
<td>Rounds the elements of ( a ) to the nearest integers towards zero</td>
</tr>
<tr>
<td>floor(a)</td>
<td>Rounds the elements of ( a ) to the nearest integers ( \geq a )</td>
</tr>
<tr>
<td>gsort(a)</td>
<td>Sort elements/rows/columns of ( a )</td>
</tr>
<tr>
<td>imag</td>
<td>Imaginary part of a matrix</td>
</tr>
<tr>
<td>intersect(str1,str2)</td>
<td>Returns elements common to two vectors ( str1 ) and ( str2 )</td>
</tr>
<tr>
<td>lex_sort</td>
<td>Sort rows of matrices in lexicographic order</td>
</tr>
<tr>
<td>linspace</td>
<td>Create vector with linearly spaced elements</td>
</tr>
<tr>
<td>logspace</td>
<td>Create vector with logarithmically spaced elements</td>
</tr>
<tr>
<td>max</td>
<td>Maximum of all elements of a vector or array</td>
</tr>
<tr>
<td>maxi</td>
<td>Maximum of all elements of a vector or array</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of all elements of a vector or array</td>
</tr>
<tr>
<td>median</td>
<td>Median of all elements of a vector or array</td>
</tr>
<tr>
<td>min</td>
<td>Minimum of all elements of a vector or array</td>
</tr>
<tr>
<td>mini</td>
<td>Minimum of all elements of a vector or array</td>
</tr>
<tr>
<td>modulo(a,b)</td>
<td>( a-b.*\text{fix}(a./b) ) if ( b \neq 0 ); remainder of ( a ) divided by ( b )</td>
</tr>
<tr>
<td>pmodulo(a,b)</td>
<td>( a-b.*\text{floor}(a./b) ) if ( b \neq 0 ); remainder of ( a ) divided by ( b )</td>
</tr>
<tr>
<td>prod</td>
<td>Product of the elements of a matrix</td>
</tr>
<tr>
<td>real</td>
<td>Real part of a matrix</td>
</tr>
<tr>
<td>round(a)</td>
<td>Round the elements of ( a ) to the nearest integers</td>
</tr>
<tr>
<td>sign(a)</td>
<td>Signum function, ( a/</td>
</tr>
<tr>
<td>sqrt(a)</td>
<td>( \sqrt{a} )</td>
</tr>
<tr>
<td>st_deviation</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of all elements of a matrix</td>
</tr>
<tr>
<td>union(a,b)</td>
<td>Extract the unique common elements of ( a ) and ( b )</td>
</tr>
<tr>
<td>unique(a)</td>
<td>Return the unique elements of ( a ) in ascending order</td>
</tr>
</tbody>
</table>

Table 5.1: Basic arithmetic functions

There is no equivalent in Matlab for the behavior of \( \text{max} (\text{mat}) \). 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 \( \text{max} \) behaves just like \( \text{max} (\text{mat}, 'r') \) does — it computes a row vector representing the maxima of every column. Likewise, \( \text{max} (\text{mat}, 'c') \) computes a column vector with the maximum element in each row (equivalent to \( \text{max} (\text{mat}, [], 2) \) in Matlab).

The analogous behavior is found for Scilab functions \( \text{cumprod}, \text{cumsum}, \text{maxi}, \text{mean}, \text{median}, \text{min}, \text{mini}, \text{prod}, \text{sum}, \) and \( \text{st_deviation} \). The functions \( \text{max} \) and \( \text{maxi} \) are equivalent as are \( \text{min} \) and \( \text{mini} \).
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.

```matlab
-->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
smatc =
  4.  2.  0.  2. !
  3.  1.  1.  3. !

In the help file 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) ≥ 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’).

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

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.

```matlab
-->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. !
```
5.2. FUNCTIONS THAT OPERATE ON SCALARS AND MATRICES

! 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 \texttt{smat1 = mat1(index,:)}).

Changing input argument \texttt{'lr'} to \texttt{'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 \texttt{cotg} the names of all the elementary transcendental functions listed in Table 5.2 agree with those of their Matlab counterparts. Furthermore, \texttt{atan} can be called with one or with two arguments. With one argument it equivalent to Matlab’s \texttt{atan}; with two arguments it corresponds to Matlab’s \texttt{atan2}. Thus, for \(x > 0\), \texttt{atan(y,x)} == \texttt{atan(y/x)}.

If the argument of any of these functions is a matrix, the function is applied to each entry separately. Thus

\[
\begin{align*}
\text{--->} &\ a = [1 2; 3 4] \\
&\ a = \\
! &\ 1. 2. ! \\
! &\ 3. 4. ! \\
\text{--->} &\ b = \text{sqrt}(a) \\
&\ b = \\
! &\ 1. 1.4142136 ! \\
! &\ 1.7320508 2. !
\end{align*}
\]
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 $\text{sqrtm}$, looks like this

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

```
-->b*b
ans =
   ! 1. + 5.551E-17i  2. !
   ! 3. + 2.776E-17i  4. !
```
5.2. FUNCTIONS THAT OPERATE ON SCALARS AND MATRICES

\[
\text{clean}(b*b) \\
\text{ans} = \\
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

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 19 illustrates how function \texttt{clean} can be used to remove such small matrix entries.

The important difference between these two examples is that in 18a corresponding entries of \( b \) are multiplied (the . in front of the *) whereas in 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 \texttt{funm} function which works for any user-specified functions; for good accuracy, matrices should be symmetric or Hermitian.

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{acoshm}</td>
<td>Matrix inverse hyperbolic cosine</td>
</tr>
<tr>
<td>\texttt{acosm}</td>
<td>Matrix arc cosine</td>
</tr>
<tr>
<td>\texttt{asinhm}</td>
<td>Matrix inverse hyperbolic sine</td>
</tr>
<tr>
<td>\texttt{atanhm}</td>
<td>Matrix inverse hyperbolic tangent</td>
</tr>
<tr>
<td>\texttt{atanhm}</td>
<td>Matrix inverse hyperbolic tangent</td>
</tr>
<tr>
<td>\texttt{coshm}</td>
<td>Matrix hyperbolic cosine</td>
</tr>
<tr>
<td>\texttt{cosm}</td>
<td>Matrix cosine</td>
</tr>
<tr>
<td>\texttt{expm}</td>
<td>Matrix exponential function</td>
</tr>
<tr>
<td>\texttt{logm}</td>
<td>Matrix natural logarithm</td>
</tr>
<tr>
<td>\texttt{sinhm}</td>
<td>Matrix hyperbolic sine</td>
</tr>
<tr>
<td>\texttt{sinm}</td>
<td>Matrix sine</td>
</tr>
<tr>
<td>\texttt{sqrtm}</td>
<td>Matrix square root</td>
</tr>
<tr>
<td>\texttt{tanhm}</td>
<td>Matrix hyperbolic tangent</td>
</tr>
<tr>
<td>\texttt{tanm}</td>
<td>Matrix tangent</td>
</tr>
</tbody>
</table>

Table 5.3: Matrix functions
5.2.3 Special Functions

Table 5.4 lists so-called special functions of mathematical physics available in Scilab.

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

Table 5.4: Special functions

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

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.

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>balanc</td>
<td>Balance matrix to improve condition number</td>
</tr>
<tr>
<td>bdiag</td>
<td>Block diagonalization of matrix</td>
</tr>
<tr>
<td>bdiag(M)</td>
<td>Block diagonalization/generalized eigenvectors of M</td>
</tr>
<tr>
<td>chol(M)</td>
<td>Choleski factorization; $R^T \cdot R = M$</td>
</tr>
<tr>
<td>colcomp(M)</td>
<td>Column compression of M</td>
</tr>
<tr>
<td>cond</td>
<td>Condition number of M</td>
</tr>
<tr>
<td>det</td>
<td>Determinant of a matrix</td>
</tr>
<tr>
<td>fullrf(M)</td>
<td>Full-rank factorization of M</td>
</tr>
<tr>
<td>fullrfk(M)</td>
<td>Full-rank factorization of $M^K$</td>
</tr>
<tr>
<td>givens</td>
<td>Given’s rotation</td>
</tr>
<tr>
<td>hess(M)</td>
<td>Hessenberg form of M</td>
</tr>
<tr>
<td>householder</td>
<td>Householder orthogonal reflection matrix</td>
</tr>
<tr>
<td>inv(M)</td>
<td>Inverse of matrix M</td>
</tr>
<tr>
<td>kernel(M)</td>
<td>Nullspace of M</td>
</tr>
<tr>
<td>linsolve</td>
<td>Linear-equation solver</td>
</tr>
<tr>
<td>norm(M)</td>
<td>Norm of M (matrix or vector)</td>
</tr>
<tr>
<td>orth(M)</td>
<td>Orthogonal basis for the range of M</td>
</tr>
<tr>
<td>pinv(M)</td>
<td>Pseudoinverse of M</td>
</tr>
<tr>
<td>polar(M)</td>
<td>Polar form of M, $M = R \cdot \expm(%i \cdot \Theta)$</td>
</tr>
<tr>
<td>qr(M)</td>
<td>QR decomposition of M</td>
</tr>
<tr>
<td>range(M)</td>
<td>Range of M</td>
</tr>
<tr>
<td>rank(M)</td>
<td>Rank of M</td>
</tr>
<tr>
<td>rcond(M)</td>
<td>Reciprocal of the condition number of M; L-1 norm</td>
</tr>
<tr>
<td>schur(M)</td>
<td>Schur decomposition</td>
</tr>
<tr>
<td>spaninter(M,N)</td>
<td>Intersection of the span of M and N</td>
</tr>
<tr>
<td>spanplus(M,N)</td>
<td>Span of M and N</td>
</tr>
<tr>
<td>spec</td>
<td>Eigenvalues of matrix</td>
</tr>
<tr>
<td>sva(M)</td>
<td>Singular-value approximation of M for specified rank</td>
</tr>
<tr>
<td>svd(M)</td>
<td>Singular-value decomposition of M</td>
</tr>
<tr>
<td>trace(M)</td>
<td>Trace (sum of diagonal elements) of M</td>
</tr>
</tbody>
</table>

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) \texttt{fft} may need some explanation. After all, it does what \texttt{fft}, \texttt{ifft}, \texttt{fft2}, and \texttt{ifft2} do in Matlab.

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>convol</td>
<td>Convolution</td>
</tr>
<tr>
<td>\texttt{fft}</td>
<td>Forward and inverse Fast Fourier Transform</td>
</tr>
<tr>
<td>\texttt{mfft}</td>
<td>Multidimensional Fast Fourier Transform</td>
</tr>
</tbody>
</table>

Table 5.7: Functions for signal processing

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

\begin{verbatim}
--->x = rand(100,1);

--->y = fft(x,-1); \hspace{1cm} 20a

--->z = fft(y,1); \hspace{1cm} 20b

--->norm(x-z)
ans =
1.532E-15
\end{verbatim}

where

\[ y_m = \sum_{n=1}^{N} x_n e^{-2\pi i (n-1)(m-1)/N} \text{ for } m = 1, \cdots, N \]  \hspace{1cm} (5.1)

with \( N \) denoting the number of elements \( x_n \). The second argument, -1, in \texttt{20a} corresponds to the minus sign in front of the exponent in (5.1). The operation performed in \texttt{20b}, \( z_m = \frac{1}{N} \sum_{n=1}^{N} y_n e^{2\pi i (n-1)(m-1)/N} \text{ for } m = 1, \cdots, N, \)

is the inverse of \texttt{20a}.

If \( xx \) is a matrix then \( \texttt{f(xx,-1)} \) performs the two-dimensional Fourier transform. It is thus equivalent to Matlab’s \texttt{fft2}. Matlab’s \texttt{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 \texttt{fft} in the form \texttt{21}.
5.2. FUNCTIONS THAT OPERATE ON SCALARS AND MATRICES

```matlab
--> 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);

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

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

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

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

Furthermore, with `yy2` computed in 21, statement 25 computes the two-dimensional FFT of `xx`:

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

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

--> 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.
5.3. FILE INPUT AND OUTPUT

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dispfiles</td>
<td>Display properties of opened files</td>
</tr>
<tr>
<td>file</td>
<td>Open/close a file, define file attributes</td>
</tr>
<tr>
<td>fileinfo</td>
<td>Get information about a file</td>
</tr>
<tr>
<td>getio</td>
<td>Get Scilab’s standard logical input/output units</td>
</tr>
<tr>
<td>mclearerr</td>
<td>Reset binary-file access errors</td>
</tr>
<tr>
<td>mclose</td>
<td>Close (all) open file(s)</td>
</tr>
<tr>
<td>meof</td>
<td>Check if end-of-file has been reached</td>
</tr>
<tr>
<td>mopen</td>
<td>Open a file</td>
</tr>
<tr>
<td>mseek</td>
<td>Set position in a binary file</td>
</tr>
<tr>
<td>mtell</td>
<td>Output the current position in a binary file</td>
</tr>
<tr>
<td>newest</td>
<td>Find newest of a set of files</td>
</tr>
<tr>
<td>xgetfile</td>
<td>Open dialog box for file selection</td>
</tr>
</tbody>
</table>

Table 5.8: Functions that open, query, 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:

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

Table 5.9: Functions that input data from files or keyboard

```scilab
-->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 its name. In a case like this `mgetl` does three things. It opens the file for reading, reads the lines requested, and closes the file again. This convenience comes at a price. It is not possible to read a file a few lines at a time. If this is necessary 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.

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

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

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 !
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)
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 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 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)
```
5.3. FILE INPUT AND OUTPUT

-->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 need 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.
CHAPTER 5. FUNCTIONS

-->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
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, \texttt{write} can also be used to write a string vector (but not a matrix) to the Scilab window without the “almost blank” lines.

\begin{verbatim}
-1.10499  1.04225  -0.44840  1.13162  -1.62805  0.76045  2.49761  -0.72190
-1.36674  0.77577  -0.65881


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
\end{verbatim}

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

5.3.4 Functions \texttt{load} and \texttt{save}

Functions \texttt{save} and \texttt{load} perform the same function they perform in Matlab: \texttt{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 \texttt{mopen} with parameter \texttt{wb} (write binary). But variables can be saved incrementally to the same file. An example is below.

\begin{verbatim}
-->a = 3;

-->fid = mopen(’C:\Temp\saved.bin’,’wb’);
\end{verbatim}
CHAPTER 5. FUNCTIONS

```matlab
-->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,

```matlab
-->clear a, clear b

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

```matlab
--->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:

<table>
<thead>
<tr>
<th>Type</th>
<th>in file</th>
<th>in Scilab</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>8-bit integer</td>
<td>int8</td>
</tr>
<tr>
<td>s</td>
<td>16-bit integer</td>
<td>int16</td>
</tr>
<tr>
<td>i</td>
<td>32-bit integer</td>
<td>int32</td>
</tr>
<tr>
<td>l</td>
<td>64-bit integer</td>
<td>double</td>
</tr>
<tr>
<td>uc</td>
<td>Unsigned 8-bit integer</td>
<td>uint8</td>
</tr>
<tr>
<td>us</td>
<td>Unsigned 16-bit integer</td>
<td>uint16</td>
</tr>
<tr>
<td>ui</td>
<td>Unsigned 32-bit integer</td>
<td>uint32</td>
</tr>
<tr>
<td>ul</td>
<td>Unsigned 64-bit integer</td>
<td>double</td>
</tr>
<tr>
<td>f</td>
<td>32-bit floating-point number</td>
<td>double</td>
</tr>
<tr>
<td>d</td>
<td>64-bit floating-point number</td>
<td>double</td>
</tr>
</tbody>
</table>
The functions can incrementally read/write files that have been opened with \texttt{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 \texttt{fopen}. Function \texttt{mopen} in Scilab has no such option; instead, byte ordering is specified together with the variable type by appending a \texttt{b} or \texttt{l} to the type parameter. Thus the statement for reading 6 big-endian, 32-bit integers from a file with file identifier \texttt{fid} is

\begin{verbatim}
-->from_file = mget(6,'ib',fid)
\end{verbatim}

Had the \texttt{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.

\subsection*{5.3.6 Functions input and disp}

Function \texttt{input} is completely equivalent to Matlab’s \texttt{input}:

\begin{verbatim}
-->response = input('Prompt user for input')
Prompt user for input-->3
response =
  3.
\end{verbatim}

The user response (3 in this example) can also be an expression involving variables in the workspace. Furthermore, by adding a second argument, \texttt{'string'} or simply \texttt{'s'}, the user response can be interpreted as a string. There is no need to put it in quotes.

Function \texttt{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.

\subsection*{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 \texttt{xgetfile} which opens a dialog box for interactive file selection. It works essentially like Matlab’s \texttt{uigetfile}. An example is

\begin{verbatim}
-->file_name = xgetfile(filemask='*.sgy',dir='D:\Data\Seismic', ...
         title='Read SEG-Y file');
\end{verbatim}

which opens a file selection window with the title “Read SEG-Y file”. The initial directory shown in the window is \texttt{D:\Data\Seismic}, and only files with file name extension \texttt{.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.

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>diary</td>
<td>Write screen output of a Scilab session to a file</td>
</tr>
<tr>
<td>getversion</td>
<td>Display version of Scilab</td>
</tr>
<tr>
<td>host</td>
<td>Execute Unix/DOS command; outputs error code</td>
</tr>
<tr>
<td>lines</td>
<td>Specify number of lines to display and columns/line</td>
</tr>
<tr>
<td>pol2tex</td>
<td>\LaTeX{} representation of a polynomial</td>
</tr>
<tr>
<td>stacksize</td>
<td>Determine/set the size of the stack</td>
</tr>
<tr>
<td>texprint</td>
<td>\LaTeX{} representation of a Scilab object</td>
</tr>
<tr>
<td>timer</td>
<td>Outputs time elapsed since the preceding call to \texttt{timer()}</td>
</tr>
<tr>
<td>unix</td>
<td>Execute Unix/DOS command; outputs error code</td>
</tr>
<tr>
<td>unix_g</td>
<td>Execute Unix/DOS command; output to variable</td>
</tr>
<tr>
<td>unix_s</td>
<td>Execute Unix/DOS command; no output (silent)</td>
</tr>
<tr>
<td>unix_w</td>
<td>Execute Unix/DOS command; output to Scilab window</td>
</tr>
<tr>
<td>unix_x</td>
<td>Execute Unix/DOS command; output to a new window</td>
</tr>
</tbody>
</table>

Table 5.11: Utility functions

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

The \LaTeX{}code function \texttt{texprint} generates may require the \texttt{amsmath} package:

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

-->texprint(mat)
ans =
\begin{pmatrix}
.2113249 & .3303271 & .8497452 & .068374 & .7263507 \\
.7560439 & .6653811 & .6857310 & .5608486 & .1985144 \\
.0002211 & .6283918 & .8782165 & .6623569 & .5442573
\end{pmatrix}
```

The \texttt{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 \texttt{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.

```scilab
--> 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

\[ \texttt{exec('script\_file.sce')} \]

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the default value</td>
</tr>
<tr>
<td>-1</td>
<td>print nothing</td>
</tr>
<tr>
<td>1</td>
<td>echo each command line</td>
</tr>
<tr>
<td>2</td>
<td>print prompt (\texttt{-- &gt;})</td>
</tr>
<tr>
<td>3</td>
<td>echo + prompt</td>
</tr>
<tr>
<td>4</td>
<td>stop before each prompt</td>
</tr>
<tr>
<td>7</td>
<td>stop + prompt + echo : useful mode for demos</td>
</tr>
</tbody>
</table>

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. Function `exec` satisfies the requirements of the command-style syntax (see Section 5.1). Thus and below are equivalent statements.

```plaintext
-->exec 'script_file.sce'
```

and

```plaintext
-->exec script_file.sce
```

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.

```plaintext
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;
```
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
        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
    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

\[
\text{function} \ [\text{out1},\text{out2},\ldots] = \text{function}_\text{name}(\text{in1},\text{in2},\ldots)
\]

familiar from Matlab. The \ldots 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 (\ldots).

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

\[
\begin{align*}
\text{function} & \ [r,\phi] = \text{polcoord}(x,y) \\
& \quad // \text{Function computes polar coordinates from cartesian coordinates} \\
& \quad r = \sqrt{x^2 + y^2}; \\
& \quad \phi = \text{atan}(y,x) \times 180/\%\text{pi}; \\
& \quad \text{endfunction}
\end{align*}
\]

This function looks much like a Matlab function except for:

- the two slashes preceding the comment;
- the use of the function \texttt{atan} rather than its Matlab equivalent \texttt{atan2};
- the use of special constant \%\texttt{pi} rather than its Matlab equivalent \texttt{pi}
• the use of `endfunction` which can be omitted (but would be required if `polcoord` were defined in-line).

<table>
<thead>
<tr>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>argn</code></td>
<td>Number of input/output arguments of a function</td>
</tr>
<tr>
<td><code>endfunction</code></td>
<td>Indicate end of function</td>
</tr>
<tr>
<td><code>error</code></td>
<td>Print error message and abort</td>
</tr>
<tr>
<td><code>function</code></td>
<td>Identify header of function definition</td>
</tr>
<tr>
<td><code>halt</code></td>
<td>Stop execution and wait for a key press</td>
</tr>
<tr>
<td><code>mode</code></td>
<td>Control amount of information displayed by function/script</td>
</tr>
<tr>
<td><code>pause</code></td>
<td>Interrupt execution of function or script</td>
</tr>
<tr>
<td><code>resume</code></td>
<td>Return from a function or resume execution after a <code>pause</code></td>
</tr>
<tr>
<td><code>return</code></td>
<td>Return from a function or resume execution after a <code>pause</code></td>
</tr>
<tr>
<td><code>varargin</code></td>
<td>Variable number of input arguments for a function</td>
</tr>
<tr>
<td><code>varargout</code></td>
<td>Variable number of output arguments for a function</td>
</tr>
<tr>
<td><code>warning</code></td>
<td>Print warning message</td>
</tr>
<tr>
<td><code>where</code></td>
<td>Output current instruction calling tree</td>
</tr>
<tr>
<td><code>whereami</code></td>
<td>Display current instruction calling tree</td>
</tr>
<tr>
<td><code>whereis</code></td>
<td>Display name of library containing function</td>
</tr>
</tbody>
</table>

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).

```scilab
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

```plaintext
function y = func1(x)
    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:

```plaintext
-->a = 1;
-->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

```plaintext
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 [31a] by defining `a` as global has no effect on the result

```plaintext
-->global a, a = 1;
```
However, if function `func1` is changed to `func1g` which also includes a global statement

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

```plaintext
--> global a
--> a = 1;
--> y = func1g(3)
    y =
    7.
--> disp(a)
    4.
```

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

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

Thus 31a, which uses `func1`, and 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 differ-
ence 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 inter-
rupted 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 \texttt{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.

\begin{verbatim}
-->a = 1;

-->typeof(a)
ans =
constant

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

-->typeof(convstr)
ans =
function

-->a = convstr;

-->typeof(a)
ans =
function

-->a('UvWxY')
ans =
uvwxyz
\end{verbatim}

Initially, the variable \texttt{a} is assigned the value 1 and is of type \texttt{constant}. On the other hand,
the function \texttt{convstr}, which converts upper case characters to lower case, is of type \texttt{function}.
Obviously, like \texttt{a}, \texttt{convstr} is used as an argument of function \texttt{typeof}. Now I set \texttt{a} equal to
\texttt{convstr} (note, that \texttt{convstr} is used without parentheses or argument). This turns \texttt{a} into a
function and, as shown in the last statement, makes it an alias for \texttt{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 \texttt{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

```plaintext
-->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

```plaintext
-->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

```plaintext
-->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

\[
-->\text{deff('y = funct(x)','y = x^2','n');}
\]

\[
-->\text{typeof(funct)}
\]
\[
\text{ans = function}
\]

\[
-->\text{type(funct)}
\]
\[
\text{ans = 11.}
\]

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”

\[
-->\text{seismic_data = tlist(['seismic','first','last','step',...}
\]
\[
\text{'units','traces']},0,1000,4,'ms');}
\]

\[
-->\text{nsamp = (seismic_data.last-seismic_data.first)/seismic_data.step+1;}
\]

\[
-->\text{seismic_data.traces=\text{rand(nsamp,10);}
\]

\[
-->\text{seismic_data}
\]

\[
\text{seismic_data(1)}
\]
7.3. FUNCTIONS FOR OPERATOR OVERLOADING

!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
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` 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

```matlab
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

```matlab
-->seismic_data.traces(1,1)
an =
     0.2113249
```
7.3. FUNCTIONS FOR OPERATOR OVERLOADING

-->seismic_data = -seismic_data;

-->seismic_data.traces(1,1)
ans =
-0.2113249

The function name for overloading binary operators has the form
\%<first_operand_type><op_code><second_operand_type>. In this definition <operand_type> is code for the type of variable the operator of type <op_code> is operating on. Operand types, i.e. codes for the various Scilab variables, are listed in the rightmost column of Table 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 =
0.2113249

-->seismic_data = seismic_data + 1;

-->seismic_data.traces(1,1)
ans =
1.2113249

It is important to note that overloading the operator + via %seismic_a_s(seismic,c) is only defined for this specific sequence of operands. The expression 1 + seismic_data causes an error message as does, for example, seismic_data - 1; but seismic_data + (-1) works.

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,

```plaintext
--->seismic_data.traces(1,1)
ans =
    0.2113249

--->seismic_data = abs(-seismic_data);

--->seismic_data.traces(1,1)
ans =
    0.2113249
```

Extraction of object elements can be overloaded by means of a function the name of which has the form `%<operand_type>_e(i1,...,in,operand)`. A somewhat simplified example is

```plaintext
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

```plaintext
--->seismic_data.traces(5,10)
ans =
    0.1853351

--->temp = seismic_data(5,10)

temp =

LIST OF TYPE "seismic"
    first: 16
    last: 16
    step: 4
    traces: 0.1853351
    units: ms
```

It is important to keep in mind that typed lists have extraction (component extraction) defined for one index. Hence, extraction with one index cannot be overloaded, and
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

```plaintext
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).

```plaintext
-->unix_w('ls C:\MyScilab')
lower.sci
upper.sci
```

Then a possible procedure to create a library is as follows;

```plaintext
-->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

```plaintext
-->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 \texttt{.scilab} will load the library \texttt{Mylib} every time Scilab is started. Note that the library name \texttt{Mylib} is not mentioned in the load statement. Nevertheless, the variable \texttt{Mylib} of type library shows up in the workspace (the library \texttt{lib} “knows” that its name in the workspace is \texttt{Mylib}).

There are a few things to keep in mind with regard to loading libraries. This is illustrated in the following.

\begin{verbatim}
-->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')    // Load library containing function lower
    !--error 4
    undefined variable : lower
\end{verbatim}

Since all unprotected variables have been removed the function \texttt{lower} is not available. To get it back one can load the library \texttt{Mylib}, and the command \texttt{who} shows that the variable \texttt{Mylib} is now in the workspace.

\begin{verbatim}
-->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    utilib    tdcslib    siglib     s2flib    roblib    optlib    metalib
elemlib    commlib    polylib    autolib    armalib    alglib    intlib
mtlbib    WSCI    SCI        %F        %T        %z        %s
%nan        %inf        $        %t        %f        %eps        %io
%i        %e
\end{verbatim}
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.

```plaintext
--> lc = lower('ABC')
lc =
abc
```

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

```plaintext
--> clear lower
```

An alternative is to bring the corrected version of `lower` into the workspace via

```plaintext
--> 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

```plaintext
-->dispfiles()
<table>
<thead>
<tr>
<th>File name</th>
<th>Unit</th>
<th>Type</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----------------</td>
</tr>
<tr>
<td>D:/PROGRAM FILES/SCILAB-2.6/scilab.hist</td>
<td>1</td>
<td>F77</td>
<td>unknown formatted</td>
</tr>
<tr>
<td>D:\MyScilab\Tests\read_segy_file.sci</td>
<td>2</td>
<td>C</td>
<td>r b</td>
</tr>
<tr>
<td>Input</td>
<td>5</td>
<td>F77</td>
<td>old formatted</td>
</tr>
<tr>
<td>Output</td>
<td>6</td>
<td>F77</td>
<td>new formatted</td>
</tr>
</tbody>
</table>

-->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.

<table>
<thead>
<tr>
<th>Matlab</th>
<th>Scilab</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>[]</td>
<td>Empty matrix</td>
</tr>
<tr>
<td>abs(a)</td>
<td>abs(a)</td>
<td>Absolute value of (a), (</td>
</tr>
<tr>
<td>acos</td>
<td>acos</td>
<td>Arc cosine</td>
</tr>
<tr>
<td>acosh</td>
<td>acosh</td>
<td>Inverse hyperbolic cosine</td>
</tr>
<tr>
<td>all(a)</td>
<td>and(a)</td>
<td>Logical AND of the elements of boolean or real numeric matrix (a)</td>
</tr>
<tr>
<td>any(a)</td>
<td>or(a)</td>
<td>Output (%t) if all entries of the boolean matrix (a) are true</td>
</tr>
<tr>
<td>asin</td>
<td>asin</td>
<td>Arc sine</td>
</tr>
<tr>
<td>asinh</td>
<td>asinh</td>
<td>Inverse hyperbolic sine</td>
</tr>
<tr>
<td>atan</td>
<td>atan</td>
<td>Arc tangent</td>
</tr>
<tr>
<td>atan2</td>
<td>atan</td>
<td>Arc tangent</td>
</tr>
<tr>
<td>atanh</td>
<td>atanh</td>
<td>Inverse hyperbolic tangent</td>
</tr>
<tr>
<td>balance</td>
<td>balanc</td>
<td>Balance matrix to improve condition number</td>
</tr>
<tr>
<td>Matlab</td>
<td>Scilab</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>besseli</td>
<td>besseli</td>
<td>Modified Bessel function of the first kind, $I_\alpha(x)$</td>
</tr>
<tr>
<td>besselj</td>
<td>besselj</td>
<td>Bessel function of the first kind, $J_\alpha(x)$</td>
</tr>
<tr>
<td>bessely</td>
<td>bessely</td>
<td>Bessel function of the second kind, $Y_\alpha(x)$</td>
</tr>
<tr>
<td>break</td>
<td>break</td>
<td>Force exit from a for or while loop</td>
</tr>
<tr>
<td>case</td>
<td>case</td>
<td>Start clause within a select block</td>
</tr>
<tr>
<td>ceil(a)</td>
<td>ceil(a)</td>
<td>Round the elements of a to the nearest integers $\geq a$</td>
</tr>
<tr>
<td>char</td>
<td>ascii</td>
<td>Convert ASCII codes to equivalent string</td>
</tr>
<tr>
<td>chol(M)</td>
<td>chol(M)</td>
<td>Choleski factorization; $R^T R = M$</td>
</tr>
<tr>
<td>clear</td>
<td>clear</td>
<td>Clear unprotected variables and functions from memory</td>
</tr>
<tr>
<td>clear global</td>
<td>clearglobal</td>
<td>Clear global variables from memory</td>
</tr>
<tr>
<td>compan</td>
<td>companion</td>
<td>Companion matrix</td>
</tr>
<tr>
<td>cond</td>
<td>cond</td>
<td>Condition number of $M$</td>
</tr>
<tr>
<td>cond</td>
<td>cond</td>
<td>Condition number of a matrix</td>
</tr>
<tr>
<td>conj</td>
<td>conj</td>
<td>Complex conjugate</td>
</tr>
<tr>
<td>conv</td>
<td>convol</td>
<td>Convolution</td>
</tr>
<tr>
<td>cos</td>
<td>cos</td>
<td>Cosine</td>
</tr>
<tr>
<td>cosh</td>
<td>cosh</td>
<td>Hyperbolic cosine</td>
</tr>
<tr>
<td>cot</td>
<td>cotg</td>
<td>Cotangent</td>
</tr>
<tr>
<td>coth</td>
<td>coth</td>
<td>Hyperbolic cotangent</td>
</tr>
<tr>
<td>cumprod</td>
<td>cumprod</td>
<td>Cumulative product of all elements of a vector or array</td>
</tr>
<tr>
<td>cumsum</td>
<td>cumsum</td>
<td>Cumulative sum of all elements of a vector or array</td>
</tr>
<tr>
<td>date</td>
<td>date</td>
<td>Current date as string</td>
</tr>
<tr>
<td>dbstack</td>
<td>whereami</td>
<td>Display current instruction calling tree</td>
</tr>
<tr>
<td>dbstack</td>
<td>where</td>
<td>Output current instruction calling tree</td>
</tr>
<tr>
<td>deblank</td>
<td>stripblanks</td>
<td>Remove leading and trailing blanks from a string</td>
</tr>
<tr>
<td>det</td>
<td>det</td>
<td>Determinant of a matrix</td>
</tr>
<tr>
<td>diag</td>
<td>diag</td>
<td>Create diagonal matrix or extract diagonal from matrix</td>
</tr>
<tr>
<td>diary</td>
<td>diary</td>
<td>Write screen output of a Scilab session to a file</td>
</tr>
<tr>
<td>disp</td>
<td>disp</td>
<td>Display input argument</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>Convert integer of any type/length to floating point</td>
</tr>
<tr>
<td>double</td>
<td>ascii</td>
<td>Convert string to equivalent ASCII codes</td>
</tr>
<tr>
<td>echo</td>
<td>mode</td>
<td>Control amount of information displayed by function/script</td>
</tr>
<tr>
<td>eig</td>
<td>bdiag</td>
<td>Block diagonalization of matrix</td>
</tr>
<tr>
<td>eig</td>
<td>spec</td>
<td>Eigenvalues of matrix</td>
</tr>
<tr>
<td>ellipj</td>
<td>%sn</td>
<td>Jacobian elliptic function, sn</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td>Start an alternative in an if or case block</td>
</tr>
<tr>
<td>Matlab</td>
<td>Scilab</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>elseif</td>
<td>elseif</td>
<td>Start a conditional alternative in an if block</td>
</tr>
<tr>
<td>end [loop]</td>
<td>end</td>
<td>Terminate for, if, select, or while clause</td>
</tr>
<tr>
<td>end [matrix]</td>
<td>$</td>
<td>Index of last element of matrix or (row/column) vector</td>
</tr>
<tr>
<td>erf</td>
<td>erf</td>
<td>Error function, $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)dt$</td>
</tr>
<tr>
<td>erfc</td>
<td>erfc</td>
<td>Complementary error function, $erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty \exp(-t^2)dt$</td>
</tr>
<tr>
<td>erfCx</td>
<td>erfCx</td>
<td>Scaled complementary error function, $erfcx(x) = \exp(x^2)erfc(x)$</td>
</tr>
<tr>
<td>error</td>
<td>error</td>
<td>Print error message and abort</td>
</tr>
<tr>
<td>eval</td>
<td>execstr</td>
<td>Evaluate string vector with Scilab expressions or statements</td>
</tr>
<tr>
<td>exist(a)</td>
<td>exists(a)</td>
<td>Test if variable a exists</td>
</tr>
<tr>
<td>exp</td>
<td>exp</td>
<td>Exponential function</td>
</tr>
<tr>
<td>expm</td>
<td>expm</td>
<td>Matrix xponential function</td>
</tr>
<tr>
<td>eye</td>
<td>eye</td>
<td>Identity matrix (or its generalization)</td>
</tr>
<tr>
<td>fclose</td>
<td>mclose</td>
<td>Close (all) open file(s)</td>
</tr>
<tr>
<td>fft</td>
<td>fft</td>
<td>Forward and inverse Fast Fourier Transform</td>
</tr>
<tr>
<td>fft2</td>
<td>fft</td>
<td>Forward and inverse Fast Fourier Transform</td>
</tr>
<tr>
<td>figure</td>
<td>xset</td>
<td>Set defaults for current graphics window</td>
</tr>
<tr>
<td>find</td>
<td>spget</td>
<td>Retrieve entries of a sparse matrix</td>
</tr>
<tr>
<td>find(a)</td>
<td>find(a)</td>
<td>Find the indices for which boolean matrix a is true</td>
</tr>
<tr>
<td>findstr</td>
<td>strindex</td>
<td>Find starting position(s) of a string in an other string</td>
</tr>
<tr>
<td>fix(a)</td>
<td>fix(a)</td>
<td>Rounds the elements of a to the nearest integers towards zero</td>
</tr>
<tr>
<td>floor(a)</td>
<td>floor(a)</td>
<td>Rounds the elements of a to the nearest integers $\geq a$</td>
</tr>
<tr>
<td>fopen</td>
<td>mopen</td>
<td>Open a file</td>
</tr>
<tr>
<td>for</td>
<td>for</td>
<td>Start a loop with a generally known number of repetitions</td>
</tr>
<tr>
<td>fprintf</td>
<td>fprintf</td>
<td>Write formatted data to file (like C-language fprintf function)</td>
</tr>
<tr>
<td>full</td>
<td>full</td>
<td>Convert sparse to full matrix</td>
</tr>
<tr>
<td>function</td>
<td>function</td>
<td>Identify header of function definition</td>
</tr>
<tr>
<td>gamma</td>
<td>gamma</td>
<td>Gamma function, $\Gamma(x) = \int_0^\infty t^{x-1} \exp(-t)dt$</td>
</tr>
<tr>
<td>gammaln</td>
<td>gammaln</td>
<td>Logarithm of the Gamma function, $\ln(\Gamma(x))$</td>
</tr>
<tr>
<td>getfield</td>
<td>getfield</td>
<td>Get a data object from a list</td>
</tr>
<tr>
<td>global</td>
<td>global</td>
<td>Define variables as global</td>
</tr>
<tr>
<td>help</td>
<td>help</td>
<td>On-line help</td>
</tr>
<tr>
<td>hess(M)</td>
<td>hess(M)</td>
<td>Hessenberg form of M</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
<td>Start a conditionally executed block of statements</td>
</tr>
<tr>
<td>ifft</td>
<td>fft</td>
<td>Forward and inverse Fast Fourier Transform</td>
</tr>
<tr>
<td>ifft2</td>
<td>fft</td>
<td>Forward and inverse Fast Fourier Transform</td>
</tr>
<tr>
<td>imag</td>
<td>imag</td>
<td>Imaginary part of a matrix</td>
</tr>
<tr>
<td>input</td>
<td>input</td>
<td>Prompt for user (keyboard) input</td>
</tr>
<tr>
<td>Matlab</td>
<td>Scilab</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>int16(a)</td>
<td>int16(a)</td>
<td>Convert a to 16-bit signed integer</td>
</tr>
<tr>
<td>int32(a)</td>
<td>int32(a)</td>
<td>Convert a to 32-bit signed integer</td>
</tr>
<tr>
<td>int8(a)</td>
<td>int8(a)</td>
<td>Convert a to 8-bit signed integer</td>
</tr>
<tr>
<td>intersect</td>
<td>intersect</td>
<td>Returns elements common to two vectors</td>
</tr>
<tr>
<td>inv(M)</td>
<td>inv(M)</td>
<td>Inverse of matrix M</td>
</tr>
<tr>
<td>isempty(a)</td>
<td>a==[]</td>
<td>Check if variable a is empty</td>
</tr>
<tr>
<td>isglobal(a)</td>
<td>isglobal(a)</td>
<td>Test if a is a global variable</td>
</tr>
<tr>
<td>isnan(a)</td>
<td>isnan(a)</td>
<td>Test if a is infinite</td>
</tr>
<tr>
<td>isreal(a)</td>
<td>isreal(a)</td>
<td>Output boolean vector with entries %t where a is %nan</td>
</tr>
<tr>
<td>keyboard</td>
<td>pause</td>
<td>Interrupt execution of function or script</td>
</tr>
<tr>
<td>length</td>
<td>length</td>
<td>Length of list; product of no. of rows and columns of matrix</td>
</tr>
<tr>
<td>linspace</td>
<td>linspace</td>
<td>Create vector with linearly spaced elements</td>
</tr>
<tr>
<td>log</td>
<td>log</td>
<td>Natural logarithm</td>
</tr>
<tr>
<td>log10</td>
<td>log10</td>
<td>Base-10 logarithm</td>
</tr>
<tr>
<td>log2</td>
<td>log2</td>
<td>Base-2 logarithm</td>
</tr>
<tr>
<td>logm</td>
<td>logm</td>
<td>Matrix natural logarithm</td>
</tr>
<tr>
<td>logspace</td>
<td>logspace</td>
<td>Create vector with logarithmically spaced elements</td>
</tr>
<tr>
<td>lookfor</td>
<td>apropos</td>
<td>Keyword search for a function</td>
</tr>
<tr>
<td>lower</td>
<td>convstr</td>
<td>Convert string to lower or upper case</td>
</tr>
<tr>
<td>max</td>
<td>maxi</td>
<td>Maximum of all elements of a vector or array</td>
</tr>
<tr>
<td>max</td>
<td>max</td>
<td>Maximum of all elements of a vector or array</td>
</tr>
<tr>
<td>mean</td>
<td>mean</td>
<td>Mean of all elements of a vector or array</td>
</tr>
<tr>
<td>median</td>
<td>median</td>
<td>Median of all elements of a vector or array</td>
</tr>
<tr>
<td>min</td>
<td>min</td>
<td>Minimum of all elements of a vector or array</td>
</tr>
<tr>
<td>min</td>
<td>mini</td>
<td>Minimum of all elements of a vector or array</td>
</tr>
<tr>
<td>mod(a,b)</td>
<td>pmodulo(a,b)</td>
<td>a-b.*floor(a./b) if b~0; remainder of a divided by b</td>
</tr>
<tr>
<td>more</td>
<td>lines</td>
<td>Specify number of lines to display and columns/line</td>
</tr>
<tr>
<td>nargin</td>
<td>argn</td>
<td>Number of input/output arguments of a function</td>
</tr>
<tr>
<td>nargout</td>
<td>argn</td>
<td>Number of input/output arguments of a function</td>
</tr>
<tr>
<td>nnz</td>
<td>nnz</td>
<td>Number of nonzero elements of a sparse matrix</td>
</tr>
<tr>
<td>norm(M)</td>
<td>norm(M)</td>
<td>Norm of M (matrix or vector)</td>
</tr>
<tr>
<td>null(M)</td>
<td>kernel(M)</td>
<td>Nullspace of M</td>
</tr>
<tr>
<td>num2str</td>
<td>string</td>
<td>Convert numbers to strings</td>
</tr>
<tr>
<td>ones</td>
<td>ones</td>
<td>Matrix of ones</td>
</tr>
<tr>
<td>orth(M)</td>
<td>orth(M)</td>
<td>Orthogonal basis for the range of M</td>
</tr>
<tr>
<td>pause</td>
<td>halt</td>
<td>Stop execution and wait for a key press</td>
</tr>
<tr>
<td>Matlab</td>
<td>Scilab</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>pinv</td>
<td>pinv(M)</td>
<td>Pseudoinverse of M</td>
</tr>
<tr>
<td>planerot</td>
<td>givens</td>
<td>Given's rotation</td>
</tr>
<tr>
<td>prod</td>
<td>prod</td>
<td>Product of the elements of a matrix</td>
</tr>
<tr>
<td>qr(M)</td>
<td>qr(M)</td>
<td>QR decomposition of M</td>
</tr>
<tr>
<td>rand</td>
<td>rand</td>
<td>Create random numbers with uniform or normal distribution</td>
</tr>
<tr>
<td>randn</td>
<td>rand</td>
<td>Create random numbers with uniform or normal distribution</td>
</tr>
<tr>
<td>rank(M)</td>
<td>rank(M)</td>
<td>Rank of M</td>
</tr>
<tr>
<td>rcond(M)</td>
<td>rcond(M)</td>
<td>Reciprocal of the condition number of M; L-1 norm</td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>Real part of a matrix</td>
</tr>
<tr>
<td>rem(a,b)</td>
<td>modulo(a,b)</td>
<td>a-b.*fix(a./b) if b~0; remainder of a divided by b</td>
</tr>
<tr>
<td>reshape</td>
<td>matrix</td>
<td>Reshape a vector or a matrix to a different-size matrix</td>
</tr>
<tr>
<td>return</td>
<td>resume</td>
<td>Return from a function or resume execution after a pause</td>
</tr>
<tr>
<td>return</td>
<td>return</td>
<td>Return from a function or resume execution after a pause</td>
</tr>
<tr>
<td>rmfield</td>
<td>null</td>
<td>Delete an element of a list</td>
</tr>
<tr>
<td>round(a)</td>
<td>round(a)</td>
<td>Round the elements of a to the nearest integers</td>
</tr>
<tr>
<td>schur(M)</td>
<td>schur(M)</td>
<td>Schur decomposition</td>
</tr>
<tr>
<td>select</td>
<td>select</td>
<td>Start a multi-branch block of statements</td>
</tr>
<tr>
<td>setfield</td>
<td>setfield</td>
<td>Set a data object in a list</td>
</tr>
<tr>
<td>sign(a)</td>
<td>sign(a)</td>
<td>Signum function, a/</td>
</tr>
<tr>
<td>sin</td>
<td>sin</td>
<td>Sine</td>
</tr>
<tr>
<td>sinh</td>
<td>sinh</td>
<td>Hyperbolic sine</td>
</tr>
<tr>
<td>size</td>
<td>size</td>
<td>Size/dimensions of a Scilab object</td>
</tr>
<tr>
<td>sort(a)</td>
<td>gsort(a)</td>
<td>Sort the elements of a</td>
</tr>
<tr>
<td>sortrows(a)</td>
<td>gsort(a)</td>
<td>Sort elements/rows/columns of a</td>
</tr>
<tr>
<td>spalloc</td>
<td>spzeros</td>
<td>Sparse zero matrix</td>
</tr>
<tr>
<td>sparse</td>
<td>sparse</td>
<td>Create sparse matrix</td>
</tr>
<tr>
<td>speye</td>
<td>speye</td>
<td>Sparse identity matrix</td>
</tr>
<tr>
<td>spones</td>
<td>spones</td>
<td>Replace non-zero elements in sparse matrix by ones</td>
</tr>
<tr>
<td>sprand</td>
<td>sprand</td>
<td>Create sparse random matrix</td>
</tr>
<tr>
<td>sqrt(a)</td>
<td>sqrtm</td>
<td>Matrix square root</td>
</tr>
<tr>
<td>sscanf</td>
<td>msscanf</td>
<td>Read variables from a string under format control</td>
</tr>
<tr>
<td>std</td>
<td>st_deviation</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>strrep</td>
<td>strsubstit</td>
<td>Substitute one string for another in a third string</td>
</tr>
<tr>
<td>sum</td>
<td>sum</td>
<td>Sum of all elements of a matrix</td>
</tr>
<tr>
<td>svd</td>
<td>svd</td>
<td>Singular value decomposition</td>
</tr>
<tr>
<td>tan</td>
<td>tan</td>
<td>Tangent</td>
</tr>
<tr>
<td>Matlab</td>
<td>Scilab</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>tanh</code></td>
<td><code>tanh</code></td>
<td>Hyperbolic tangent</td>
</tr>
<tr>
<td><code>tic</code></td>
<td><code>timer()</code></td>
<td>Outputs time elapsed since the call to <code>timer()</code></td>
</tr>
<tr>
<td><code>toc</code></td>
<td><code>timer()</code></td>
<td>Outputs time elapsed since the preceding call to <code>timer()</code></td>
</tr>
<tr>
<td><code>toeplitz</code></td>
<td><code>toeplitz</code></td>
<td>Toeplitz matrix</td>
</tr>
<tr>
<td><code>trace</code></td>
<td><code>trace</code></td>
<td>Trace (sum of diagonal elements) of a matrix</td>
</tr>
<tr>
<td><code>tril</code></td>
<td><code>tril</code></td>
<td>Extract lower-triangular part of a matrix</td>
</tr>
<tr>
<td><code>triu</code></td>
<td><code>triu</code></td>
<td>Extract upper-triangular part of a matrix</td>
</tr>
<tr>
<td><code>try</code></td>
<td><code>errcatch</code></td>
<td>Trap error</td>
</tr>
<tr>
<td><code>uigetfile</code></td>
<td><code>xgetfile</code></td>
<td>Open dialog box for file selection</td>
</tr>
<tr>
<td><code>uint16(a)</code></td>
<td><code>uint16(a)</code></td>
<td>Convert <code>a</code> to 16-bit unsigned integer</td>
</tr>
<tr>
<td><code>uint32(a)</code></td>
<td><code>uint32(a)</code></td>
<td>Convert <code>a</code> to 32-bit unsigned integer</td>
</tr>
<tr>
<td><code>uint8(a)</code></td>
<td><code>uint8(a)</code></td>
<td>Convert <code>a</code> to 8-bit unsigned integer</td>
</tr>
<tr>
<td><code>union</code></td>
<td><code>union(a,b)</code></td>
<td>Extract the unique common elements of <code>a</code> and <code>b</code></td>
</tr>
<tr>
<td><code>unique(a)</code></td>
<td><code>unique(a)</code></td>
<td>Return the unique elements of <code>a</code> in ascending order</td>
</tr>
<tr>
<td><code>unix</code></td>
<td><code>unix_w</code></td>
<td>Execute Unix/DOS command; output to Scilab window</td>
</tr>
<tr>
<td><code>upper</code></td>
<td><code>convstr</code></td>
<td>Convert string to lower or upper case</td>
</tr>
<tr>
<td><code>varargin</code></td>
<td><code>varargin</code></td>
<td>Variable number of input arguments for a function</td>
</tr>
<tr>
<td><code>varargout</code></td>
<td><code>varargout</code></td>
<td>Variable number of output arguments for a function</td>
</tr>
<tr>
<td><code>version</code></td>
<td><code>getversion</code></td>
<td>Display version of Scilab</td>
</tr>
<tr>
<td><code>warning</code></td>
<td><code>warning</code></td>
<td>Print warning message</td>
</tr>
<tr>
<td><code>which</code></td>
<td><code>whereis</code></td>
<td>Display name of library containing function</td>
</tr>
<tr>
<td><code>while</code></td>
<td><code>while</code></td>
<td>Start repeated execution of a block while a condition is satisfied</td>
</tr>
<tr>
<td><code>who</code></td>
<td><code>who</code></td>
<td>Displays/outputs names of current variables</td>
</tr>
<tr>
<td><code>whos</code></td>
<td><code>whos</code></td>
<td>Displays/outputs names and specifics of current variables</td>
</tr>
<tr>
<td><code>zeros</code></td>
<td><code>zeros</code></td>
<td>Matrix of zeros</td>
</tr>
</tbody>
</table>
Index

arguments
  named, 55
  variable-length list of, 56, 87
ASCII codes, 27

boolean operators, 35
boolean variables, 35

cell array, 6, 25, 26, 39–42
character string, see string
command-style syntax, 15, 54, 81
comments, 7
complex number, 21
constants, built-in, 10
continuation of a statement, 8
convolution, 64, 103
copy, 5
date, 16
directory
  private, see private directory
DOS, see MS-DOS

dead of file, 67
error trapping, 12, 14, 15, 32, 33, 72, 78, 82

Fast Fourier Transform, 64–65, 104
FFT, see Fast Fourier Transform
file, 66–75
  opening, closing, 66
flow control, 12
font, 4
format
  loose, 2
  tight, 2
function
  as argument, 89
  as variable, 89

inline, 91, 92
library, 90
loading, 90
overloading, 95
private, see private function
Scilab, 54–77
user, 84–97

help, 5, 6

integer, 74
interrupt, see pause

Kronecker
  product, 8
  division, 8

last element
  of vector or matrix, 22, 30
\LaTeX, 76
length
  list, 42
  numeric matrix, 15
  string, 25, 26, 30
library, 98–100
lines displayed, 4
list, 39
  matrix-like typed, 49–50
  ordinary, 39–43
  polynomials, 50–53
  typed, 6, 42–49, 96

Matlab
  m-file conversion, 97
  emulation, 6
Matlab-mode, 24
matrix, 21, 22, 24, 107
  companion, 24, 103
empty, 24, 102
Frank, 24
Hilbert, 24
identity, 24, 104
of ones, 24, 105
of zeros, 24, 107
polynomial, see polynomial matrix, 51
random, 24, 106
rational, 51
reshape a, 106
sparse, 62
sparse identity, 62, 106
sparse random, 62, 106
Toeplitz, 24, 107
MS-DOS, 77

operator
  binary, 95
  code, 94, 95
  overloading, 19, 44, 50, 92–97
  unary, 94
overloading
  operator, see operator, overloading
  variable display, see variable, display overloading

paging, see lines displayed
paste, 5
path, 81
polynomial, 50–53, 76
  division, 51
  matrix, 51
private directory, 91
private function, 91
program
  exit, 5
  interrupt, 5, 15
  quit, 5
  resume, 5

Scilab
  syntax, 7, 8
script, 78–83
search path, 32, 81

size
  list, 42
  string, 25–27
sorting, 58–59
startup file, 4, 99
string
  concatenation, 27, 92
  extraction, 28
structure, 1, 39, 44–47
termination script, 5
time, 16
tuple assignment, 9, 41
type name, 43, 44, 92, 94
type of variable, 19, 20
  boolean matrix, 19
  boolean sparse matrix, 19
  compiled function, 19
  constant, 19
  function library, 19
  implied-size vector, 19, 22
  integer, 20
  list, 19
  Matlab sparse matrix, 19
  matrix-like typed list, 19
  pointer, 19
  polynomial matrix, 19
  rational, 38, 51, 53
  sparse matrix, 19
  string, 19
  typed list, 19
  un-compiled function, 19
typed list, see list, typed

UNIX, 77

variable
  conversion into an expression, 31
  display overloading, 93, 94
  global, 16, 87, 88, 104
  name, 7, 8, 91
  type, see type, variable
working directory, 78
<table>
<thead>
<tr>
<th>Scilab Function/Variable</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>workspace</td>
<td>1, 5, 15–17, 33, 73, 78, 86, 88, 89, 98–100</td>
</tr>
<tr>
<td>break</td>
<td>12, 103</td>
</tr>
<tr>
<td>case</td>
<td>12, 103</td>
</tr>
<tr>
<td>ceil</td>
<td>57, 103</td>
</tr>
<tr>
<td>chdir</td>
<td>81</td>
</tr>
<tr>
<td>chfact</td>
<td>62</td>
</tr>
<tr>
<td>chol</td>
<td>63, 103</td>
</tr>
<tr>
<td>chsolve</td>
<td>62</td>
</tr>
<tr>
<td>clean</td>
<td>51, 57, 61</td>
</tr>
<tr>
<td>clear</td>
<td>16, 99, 100, 103</td>
</tr>
<tr>
<td>clearglobal</td>
<td>16, 103</td>
</tr>
<tr>
<td>cmndred</td>
<td>51</td>
</tr>
<tr>
<td>coeff</td>
<td>51</td>
</tr>
<tr>
<td>coffg</td>
<td>51</td>
</tr>
<tr>
<td>colcomp</td>
<td>63</td>
</tr>
<tr>
<td>colcompr</td>
<td>51</td>
</tr>
<tr>
<td>companion</td>
<td>24, 103</td>
</tr>
<tr>
<td>cond</td>
<td>63, 103</td>
</tr>
<tr>
<td>conj</td>
<td>57, 103</td>
</tr>
<tr>
<td>convol</td>
<td>64, 103</td>
</tr>
<tr>
<td>convstr</td>
<td>26, 89, 105, 107</td>
</tr>
<tr>
<td>cos</td>
<td>60, 103</td>
</tr>
<tr>
<td>cosh</td>
<td>60, 103</td>
</tr>
<tr>
<td>coshm</td>
<td>61</td>
</tr>
<tr>
<td>cosm</td>
<td>61</td>
</tr>
<tr>
<td>cotg</td>
<td>60, 103</td>
</tr>
<tr>
<td>coth</td>
<td>60, 103</td>
</tr>
<tr>
<td>cumprod</td>
<td>57, 103</td>
</tr>
<tr>
<td>cumsum</td>
<td>57, 103</td>
</tr>
<tr>
<td>date</td>
<td>16, 26, 103</td>
</tr>
<tr>
<td>deff</td>
<td>91, 92</td>
</tr>
<tr>
<td>degree</td>
<td>51</td>
</tr>
<tr>
<td>derivat</td>
<td>51</td>
</tr>
<tr>
<td>denom</td>
<td>51</td>
</tr>
<tr>
<td>det</td>
<td>51, 63, 103</td>
</tr>
<tr>
<td>determ</td>
<td>51</td>
</tr>
<tr>
<td>detr</td>
<td>51</td>
</tr>
<tr>
<td>diag</td>
<td>24, 103</td>
</tr>
<tr>
<td>diary</td>
<td>69, 76, 103</td>
</tr>
<tr>
<td>diophant</td>
<td>51</td>
</tr>
<tr>
<td>disp</td>
<td>16, 18, 69, 75, 103</td>
</tr>
<tr>
<td>dispfiles</td>
<td>67, 101</td>
</tr>
<tr>
<td>amell</td>
<td>62</td>
</tr>
<tr>
<td>and</td>
<td>37, 38, 102</td>
</tr>
<tr>
<td>apropos</td>
<td>5, 6, 16, 105</td>
</tr>
<tr>
<td>argn</td>
<td>85, 88, 105</td>
</tr>
<tr>
<td>ascii</td>
<td>26, 27, 29, 30, 103</td>
</tr>
<tr>
<td>asin</td>
<td>60, 102</td>
</tr>
<tr>
<td>asinh</td>
<td>60, 102</td>
</tr>
<tr>
<td>asinhm</td>
<td>61</td>
</tr>
<tr>
<td>atanh</td>
<td>60, 102</td>
</tr>
<tr>
<td>atan</td>
<td>59, 60, 102</td>
</tr>
<tr>
<td>atanhm</td>
<td>61</td>
</tr>
<tr>
<td>blanks</td>
<td>29</td>
</tr>
<tr>
<td>bool2s</td>
<td>38, 57</td>
</tr>
<tr>
<td>char</td>
<td>24, 102</td>
</tr>
<tr>
<td>$</td>
<td>16, 22, 29, 30, 46, 104</td>
</tr>
<tr>
<td>%asn</td>
<td>62</td>
</tr>
<tr>
<td>%io</td>
<td>77</td>
</tr>
<tr>
<td>%x</td>
<td>62</td>
</tr>
<tr>
<td>%sn</td>
<td>62, 103</td>
</tr>
<tr>
<td>a==[]</td>
<td>105</td>
</tr>
<tr>
<td>abort</td>
<td>5</td>
</tr>
<tr>
<td>abs</td>
<td>57, 102</td>
</tr>
<tr>
<td>acos</td>
<td>60, 102</td>
</tr>
<tr>
<td>acosh</td>
<td>60, 102</td>
</tr>
<tr>
<td>acoshm</td>
<td>61</td>
</tr>
<tr>
<td>acosm</td>
<td>61</td>
</tr>
<tr>
<td>amell</td>
<td>62</td>
</tr>
<tr>
<td>and</td>
<td>37, 38, 102</td>
</tr>
<tr>
<td>apropos</td>
<td>5, 6, 16, 105</td>
</tr>
<tr>
<td>argn</td>
<td>85, 88, 105</td>
</tr>
<tr>
<td>ascii</td>
<td>26, 27, 29, 30, 103</td>
</tr>
<tr>
<td>asin</td>
<td>60, 102</td>
</tr>
<tr>
<td>asinh</td>
<td>60, 102</td>
</tr>
<tr>
<td>asinhm</td>
<td>61</td>
</tr>
<tr>
<td>atanhm</td>
<td>61</td>
</tr>
<tr>
<td>auread</td>
<td>68</td>
</tr>
<tr>
<td>auwrite</td>
<td>69</td>
</tr>
<tr>
<td>balanc</td>
<td>63, 102</td>
</tr>
<tr>
<td>bandwr</td>
<td>62</td>
</tr>
<tr>
<td>bdiag</td>
<td>63, 103</td>
</tr>
<tr>
<td>besseli</td>
<td>62, 103</td>
</tr>
<tr>
<td>besselj</td>
<td>62, 103</td>
</tr>
<tr>
<td>besselk</td>
<td>62, 103</td>
</tr>
<tr>
<td>bessely</td>
<td>62, 103</td>
</tr>
<tr>
<td>bezout</td>
<td>51</td>
</tr>
<tr>
<td>dispfiles</td>
<td>67, 101</td>
</tr>
</tbody>
</table>
INDEX


dlgamma, 62
double, 22, 103

edit, 90
else, 12, 103
elseif, 12, 104
emptystr, 26, 29, 30
end, 12, 104
endfunction, 85
erf, 62, 104
erfc, 62, 104
erfpx, 62, 104
errcatch, 12, 14, 15, 107
errclear, 14, 15
error, 85, 104
eval, 32
evstr, 32–34
excel2sci, 68
exec, 79, 81
execstr, 32, 78, 82, 104
exists, 15, 36, 38, 104
exit, 5
exp, 60, 104
expm, 61, 104
eye, 24, 104

factors, 51
fft, 64, 104
file, 66, 67, 71, 72, 82, 83
fileinfo, 67
find, 38, 104
fix, 57, 104
floor, 57, 104
for, 12, 104
fprintf, 69, 104
fprintfMat, 69
fscanf, 68
fscanfMat, 68
full, 62, 104
fullrf, 63
fullrffk, 63
function, 85, 104
gamma, 62, 104
gammaln, 62, 104
gcd, 51
genlib, 90, 98
getd, 90
getdate, 16
getf, 90, 100, 101
getfield, 40, 104
getio, 67
getversion, 76, 107
givens, 63, 106
global, 16, 81, 86–88, 104
grand, 24
grep, 26, 30
gsort, 26, 57–59, 106
halt, 15, 16, 85, 105
help, 5, 6, 16, 104
hermit, 51
hess, 63, 104
horner, 51
host, 76, 77
householder, 63
hrmt, 51

iconvert, 22
if, 12, 13, 104
imag, 57, 104
input, 68, 75, 104
int16, 22, 105
int32, 22, 105
int8, 22, 23, 105
intersect, 26, 57, 105
inttype, 26, 57, 105
inv, 51, 63, 105
inver, 51
isdef, 36, 38
iserror, 14, 38
isglobal, 38, 105
isinf, 38, 105
isletter, 27
ismember, 39
isnan, 38, 105
isreal, 38, 105
INDEX

kernel, 63, 105
lcm, 51
lcmdiag, 51
ldiv, 51
length, 6, 26, 40, 42, 105
lex_sort, 57
lines, 76, 105
linsolve, 63
linspace, 57, 105
list, 19, 39, 40
load, 68, 73, 74, 90, 99
loadwave, 68
log, 60, 105
log10, 60, 105
log2, 60, 105
logm, 61, 105
logspace, 57, 105
lstcat, 40
lufact, 62
luget, 62
lusolve, 62
matrix, 70, 106
max, 56, 57, 105
maxi, 57, 105
mclearerr, 67
mclose, 66, 67, 69, 101, 104
mean, 57, 105
median, 57, 105
meof, 67
mfft, 64
mfprintf, 69
mscanf, 68
mget, 68, 74, 75
mgeti, 68, 74
mgetl, 67–70
mgetstr, 68
min, 57, 105
mini, 57, 105
mlist, 19, 39, 40, 49
mode, 78–81, 85, 103
modulo, 57, 106
mopen, 66–68, 71, 73, 75, 104
mprintf, 69
mput, 69, 74
mputl, 67, 69–71
mputstr, 69
mscanf, 68
MSDOS, 38, 77
mseek, 67
msprintf, 26
msscanf, 26, 31, 106
mtell, 67
mtlb_cell, 6
mtlb_diff, 6
mtlb_load, 68
mtlb_mode, 24, 38
mtlb_save, 69
newest, 67
nnz, 62, 105
norm, 63, 105
null, 40, 41, 106
ones, 24, 36, 105
or, 37, 38, 102
orth, 63, 105
otherwise, 12
part, 26, 29, 82
pause, 5, 15–17, 85, 88, 105
pdiv, 51
pinv, 63, 106
pmodulo, 57, 105
pol2str, 26, 51
pol2tex, 76
polar, 63
poly, 50
print, 69
printf, 69
prod, 57, 106
qr, 63, 106
quit, 5
rand, 24, 106
range, 63
rank, 63, 106
<table>
<thead>
<tr>
<th>Function</th>
<th>113</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcond</td>
<td>63, 106</td>
</tr>
<tr>
<td>read</td>
<td>68, 70, 71</td>
</tr>
<tr>
<td>read4b</td>
<td>68</td>
</tr>
<tr>
<td>readd</td>
<td>68</td>
</tr>
<tr>
<td>real</td>
<td>57, 106</td>
</tr>
<tr>
<td>reshape</td>
<td>70</td>
</tr>
<tr>
<td>residu</td>
<td>51</td>
</tr>
<tr>
<td>resume</td>
<td>5, 18, 85, 88, 106</td>
</tr>
<tr>
<td>return</td>
<td>5, 16–18, 82, 85, 88, 106</td>
</tr>
<tr>
<td>roots</td>
<td>51</td>
</tr>
<tr>
<td>round</td>
<td>57, 106</td>
</tr>
<tr>
<td>rowcompr</td>
<td>51</td>
</tr>
<tr>
<td>save</td>
<td>68, 73, 74</td>
</tr>
<tr>
<td>savewave</td>
<td>69</td>
</tr>
<tr>
<td>scanf_conversion</td>
<td>31</td>
</tr>
<tr>
<td>schur</td>
<td>63, 106</td>
</tr>
<tr>
<td>sci2exp</td>
<td>31</td>
</tr>
<tr>
<td>sci2expr</td>
<td>26</td>
</tr>
<tr>
<td>select</td>
<td>12, 106</td>
</tr>
<tr>
<td>setfield</td>
<td>40, 106</td>
</tr>
<tr>
<td>sfact</td>
<td>51</td>
</tr>
<tr>
<td>sign</td>
<td>57, 106</td>
</tr>
<tr>
<td>simp</td>
<td>51</td>
</tr>
<tr>
<td>simp_mode</td>
<td>38, 51, 52</td>
</tr>
<tr>
<td>sin</td>
<td>60, 106</td>
</tr>
<tr>
<td>sinh</td>
<td>60, 106</td>
</tr>
<tr>
<td>sinh,</td>
<td>61</td>
</tr>
<tr>
<td>sinm</td>
<td>61</td>
</tr>
<tr>
<td>size</td>
<td>25, 26, 40, 42, 106</td>
</tr>
<tr>
<td>spaninter</td>
<td>63</td>
</tr>
<tr>
<td>spanplus</td>
<td>63</td>
</tr>
<tr>
<td>sparse</td>
<td>62, 106</td>
</tr>
<tr>
<td>spchol</td>
<td>62</td>
</tr>
<tr>
<td>spec</td>
<td>63, 103</td>
</tr>
<tr>
<td>speye</td>
<td>62, 106</td>
</tr>
<tr>
<td>spget</td>
<td>62, 104</td>
</tr>
<tr>
<td>spones</td>
<td>62, 106</td>
</tr>
<tr>
<td>sprand</td>
<td>62, 106</td>
</tr>
<tr>
<td>spzeros</td>
<td>62, 106</td>
</tr>
<tr>
<td>sqrt</td>
<td>57, 106</td>
</tr>
<tr>
<td>sqrtm</td>
<td>61, 106</td>
</tr>
<tr>
<td>st_deviation</td>
<td>57, 106</td>
</tr>
<tr>
<td>stacksize</td>
<td>76</td>
</tr>
<tr>
<td>strcat</td>
<td>26, 28</td>
</tr>
<tr>
<td>strindex</td>
<td>26, 30, 104</td>
</tr>
<tr>
<td>string</td>
<td>26, 105</td>
</tr>
<tr>
<td>stripblanks</td>
<td>26, 29, 103</td>
</tr>
<tr>
<td>strsubst</td>
<td>26, 106</td>
</tr>
<tr>
<td>sum</td>
<td>57, 106</td>
</tr>
<tr>
<td>sva</td>
<td>63</td>
</tr>
<tr>
<td>svd</td>
<td>63, 106</td>
</tr>
<tr>
<td>sylm</td>
<td>24, 51</td>
</tr>
<tr>
<td>tan</td>
<td>60, 106</td>
</tr>
<tr>
<td>tanh</td>
<td>60, 107</td>
</tr>
<tr>
<td>tanhm</td>
<td>61</td>
</tr>
<tr>
<td>tanm</td>
<td>61</td>
</tr>
<tr>
<td>testmatrix</td>
<td>24</td>
</tr>
<tr>
<td>texprint</td>
<td>76</td>
</tr>
<tr>
<td>timer</td>
<td>16, 76, 107</td>
</tr>
<tr>
<td>tlist</td>
<td>19, 39, 40, 43, 46</td>
</tr>
<tr>
<td>toeplitz</td>
<td>24, 107</td>
</tr>
<tr>
<td>trace</td>
<td>63, 107</td>
</tr>
<tr>
<td>tril</td>
<td>107</td>
</tr>
<tr>
<td>triu</td>
<td>107</td>
</tr>
<tr>
<td>type</td>
<td>16, 19, 20, 91, 92</td>
</tr>
<tr>
<td>typename</td>
<td>20</td>
</tr>
<tr>
<td>typeof</td>
<td>16, 19, 20, 23, 31, 89, 92</td>
</tr>
<tr>
<td>uint16</td>
<td>22, 107</td>
</tr>
<tr>
<td>uint32</td>
<td>22, 107</td>
</tr>
<tr>
<td>uint8</td>
<td>20, 22, 107</td>
</tr>
<tr>
<td>union</td>
<td>26, 57, 107</td>
</tr>
<tr>
<td>unique</td>
<td>26, 57, 107</td>
</tr>
<tr>
<td>unix</td>
<td>76</td>
</tr>
<tr>
<td>unix_g</td>
<td>76, 77</td>
</tr>
<tr>
<td>unix_s</td>
<td>76</td>
</tr>
<tr>
<td>unix_w</td>
<td>76, 98, 107</td>
</tr>
<tr>
<td>unix_x</td>
<td>76</td>
</tr>
<tr>
<td>varargin</td>
<td>56, 85, 107</td>
</tr>
<tr>
<td>varargout</td>
<td>85, 107</td>
</tr>
<tr>
<td>warning</td>
<td>85, 107</td>
</tr>
<tr>
<td>wavread</td>
<td>68</td>
</tr>
<tr>
<td>wavwrite</td>
<td>69</td>
</tr>
<tr>
<td>where</td>
<td>85, 103</td>
</tr>
</tbody>
</table>
whereami, 85, 103
whereis, 16, 85, 107
while, 12, 107
who, 16, 20, 99, 107
whos, 16, 20, 107
writb, 69
write, 69–73, 77
write4b, 69
xgetfile, 67, 75, 107
xset, 104
zeros, 24, 36, 107

clear global, 103
compan, 103
complex, 21
cond, 103
conj, 103
conv, 103
cos, 103
cosh, 103
cot, 103
coth, 103
cumprod, 103
cumsum, 103
date, 103
dbstack, 103
deal, 9
deblank, 29, 103
det, 103
diag, 103
diary, 76, 103
diff, 6
disp, 103
double, 27, 103
echo, 103
eig, 103
ellipj, 103
else, 12, 103
elseif, 12, 104
end, 22
end [loop], 12, 104
end [matrix], 46, 104
erf, 104
erfc, 104
erfcx, 104
error, 104
eval, 32, 104
evalin, 33
exist, 15, 104
exp, 104
expm, 104
eye, 104
fclose, 104

Index of Matlab Functions
and Variables

[], 102
abs, 102
acos, 102
acosh, 102
all, 37, 102
any, 37, 38, 102
asin, 102
asinh, 102
atan, 59, 102
atan2, 59, 102
atanh, 102
balance, 102
besseli, 103
besselj, 103
besselk, 103
bessely, 103
blanks, 29
break, 12, 103
case, 12, 103
catch, 12, 14
ceil, 103
cell, 6, 30
char, 27, 103
chol, 103
clear, 103
<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>feval</td>
<td>89</td>
</tr>
<tr>
<td>fft</td>
<td>64, 104</td>
</tr>
<tr>
<td>fft2</td>
<td>64, 104</td>
</tr>
<tr>
<td>figure</td>
<td>104</td>
</tr>
<tr>
<td>find</td>
<td>104</td>
</tr>
<tr>
<td>findstr</td>
<td>104</td>
</tr>
<tr>
<td>finish</td>
<td>5</td>
</tr>
<tr>
<td>fix</td>
<td>104</td>
</tr>
<tr>
<td>floor</td>
<td>104</td>
</tr>
<tr>
<td>fopen</td>
<td>66, 75, 104</td>
</tr>
<tr>
<td>for</td>
<td>12, 104</td>
</tr>
<tr>
<td>fprintf</td>
<td>104</td>
</tr>
<tr>
<td>full</td>
<td>104</td>
</tr>
<tr>
<td>function</td>
<td>104</td>
</tr>
<tr>
<td>funm</td>
<td>61</td>
</tr>
<tr>
<td>gamma</td>
<td>104</td>
</tr>
<tr>
<td>gammaln</td>
<td>104</td>
</tr>
<tr>
<td>getfield</td>
<td>104</td>
</tr>
<tr>
<td>global</td>
<td>104</td>
</tr>
<tr>
<td>help</td>
<td>5, 104</td>
</tr>
<tr>
<td>hess</td>
<td>104</td>
</tr>
<tr>
<td>if</td>
<td>12, 104</td>
</tr>
<tr>
<td>ifft</td>
<td>64, 104</td>
</tr>
<tr>
<td>ifft2</td>
<td>64, 104</td>
</tr>
<tr>
<td>imag</td>
<td>104</td>
</tr>
<tr>
<td>input</td>
<td>75, 104</td>
</tr>
<tr>
<td>int16</td>
<td>105</td>
</tr>
<tr>
<td>int32</td>
<td>105</td>
</tr>
<tr>
<td>int8</td>
<td>105</td>
</tr>
<tr>
<td>intersect</td>
<td>105</td>
</tr>
<tr>
<td>inv</td>
<td>105</td>
</tr>
<tr>
<td>isa</td>
<td>20</td>
</tr>
<tr>
<td>iscell</td>
<td>19</td>
</tr>
<tr>
<td>ischar</td>
<td>19</td>
</tr>
<tr>
<td>isempty</td>
<td>105</td>
</tr>
<tr>
<td>isglobal</td>
<td>105</td>
</tr>
<tr>
<td>isinf</td>
<td>105</td>
</tr>
<tr>
<td>isletter</td>
<td>27</td>
</tr>
<tr>
<td>ismember</td>
<td>31, 38</td>
</tr>
<tr>
<td>isnan</td>
<td>105</td>
</tr>
<tr>
<td>isnumeric</td>
<td>19</td>
</tr>
<tr>
<td>isreal</td>
<td>105</td>
</tr>
<tr>
<td>issparse</td>
<td>19</td>
</tr>
<tr>
<td>isstruct</td>
<td>19</td>
</tr>
<tr>
<td>keyboard</td>
<td>17, 88, 105</td>
</tr>
<tr>
<td>length</td>
<td>105</td>
</tr>
<tr>
<td>linspace</td>
<td>105</td>
</tr>
<tr>
<td>load</td>
<td>74</td>
</tr>
<tr>
<td>log</td>
<td>105</td>
</tr>
<tr>
<td>log10</td>
<td>105</td>
</tr>
<tr>
<td>log2</td>
<td>105</td>
</tr>
<tr>
<td>logical</td>
<td>35, 36</td>
</tr>
<tr>
<td>logm</td>
<td>105</td>
</tr>
<tr>
<td>logspace</td>
<td>105</td>
</tr>
<tr>
<td>lookfor</td>
<td>5, 105</td>
</tr>
<tr>
<td>lower</td>
<td>105</td>
</tr>
<tr>
<td>max</td>
<td>57, 105</td>
</tr>
<tr>
<td>mean</td>
<td>105</td>
</tr>
<tr>
<td>median</td>
<td>105</td>
</tr>
<tr>
<td>min</td>
<td>105</td>
</tr>
<tr>
<td>mod</td>
<td>105</td>
</tr>
<tr>
<td>more</td>
<td>105</td>
</tr>
<tr>
<td>nargin</td>
<td>105</td>
</tr>
<tr>
<td>nargout</td>
<td>105</td>
</tr>
<tr>
<td>nmax</td>
<td>105</td>
</tr>
<tr>
<td>nnz</td>
<td>105</td>
</tr>
<tr>
<td>norm</td>
<td>105</td>
</tr>
<tr>
<td>null</td>
<td>105</td>
</tr>
<tr>
<td>num2str</td>
<td>105</td>
</tr>
<tr>
<td>ones</td>
<td>105</td>
</tr>
<tr>
<td>orth</td>
<td>105</td>
</tr>
<tr>
<td>otherwise</td>
<td>12</td>
</tr>
<tr>
<td>pause</td>
<td>105</td>
</tr>
<tr>
<td>pinv</td>
<td>106</td>
</tr>
<tr>
<td>planerot</td>
<td>106</td>
</tr>
<tr>
<td>prod</td>
<td>106</td>
</tr>
<tr>
<td>qr</td>
<td>106</td>
</tr>
<tr>
<td>rand</td>
<td>106</td>
</tr>
<tr>
<td>randn</td>
<td>106</td>
</tr>
<tr>
<td>rank</td>
<td>106</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
</tr>
<tr>
<td>rcond</td>
<td>106</td>
</tr>
<tr>
<td>real</td>
<td>106</td>
</tr>
<tr>
<td>rem</td>
<td>106</td>
</tr>
<tr>
<td>reshape</td>
<td>106</td>
</tr>
<tr>
<td>return</td>
<td>106</td>
</tr>
<tr>
<td>rmfield</td>
<td>106</td>
</tr>
<tr>
<td>roots</td>
<td>52</td>
</tr>
<tr>
<td>round</td>
<td>106</td>
</tr>
<tr>
<td>save</td>
<td>73</td>
</tr>
<tr>
<td>schur</td>
<td>106</td>
</tr>
<tr>
<td>select</td>
<td>106</td>
</tr>
<tr>
<td>setfield</td>
<td>106</td>
</tr>
<tr>
<td>sign</td>
<td>106</td>
</tr>
<tr>
<td>sin</td>
<td>106</td>
</tr>
<tr>
<td>sinh</td>
<td>106</td>
</tr>
<tr>
<td>size</td>
<td>26, 50, 106</td>
</tr>
<tr>
<td>sort</td>
<td>58, 106</td>
</tr>
<tr>
<td>sortrows</td>
<td>58, 106</td>
</tr>
<tr>
<td>spalloc</td>
<td>106</td>
</tr>
<tr>
<td>sparse</td>
<td>106</td>
</tr>
<tr>
<td>speye</td>
<td>106</td>
</tr>
<tr>
<td>spones</td>
<td>106</td>
</tr>
<tr>
<td>sprand</td>
<td>106</td>
</tr>
<tr>
<td>sqrt</td>
<td>106</td>
</tr>
<tr>
<td>sqrtm</td>
<td>106</td>
</tr>
<tr>
<td>sscanf</td>
<td>31, 106</td>
</tr>
<tr>
<td>std</td>
<td>106</td>
</tr>
<tr>
<td>strrep</td>
<td>106</td>
</tr>
<tr>
<td>sum</td>
<td>106</td>
</tr>
<tr>
<td>svd</td>
<td>106</td>
</tr>
<tr>
<td>switch</td>
<td>12</td>
</tr>
<tr>
<td>tan</td>
<td>106</td>
</tr>
<tr>
<td>tanh</td>
<td>107</td>
</tr>
<tr>
<td>tic</td>
<td>107</td>
</tr>
<tr>
<td>toc</td>
<td>107</td>
</tr>
<tr>
<td>toeplitz</td>
<td>107</td>
</tr>
<tr>
<td>trace</td>
<td>107</td>
</tr>
<tr>
<td>tril</td>
<td>107</td>
</tr>
<tr>
<td>triu</td>
<td>107</td>
</tr>
<tr>
<td>try</td>
<td>12, 14, 107</td>
</tr>
<tr>
<td>type</td>
<td>53</td>
</tr>
<tr>
<td>typeof</td>
<td>50, 53</td>
</tr>
<tr>
<td>uigetfile</td>
<td>75, 107</td>
</tr>
<tr>
<td>uint16</td>
<td>107</td>
</tr>
<tr>
<td>uint32</td>
<td>107</td>
</tr>
<tr>
<td>uint8</td>
<td>20, 107</td>
</tr>
<tr>
<td>union</td>
<td>107</td>
</tr>
<tr>
<td>unique</td>
<td>107</td>
</tr>
<tr>
<td>unix</td>
<td>107</td>
</tr>
<tr>
<td>upper</td>
<td>107</td>
</tr>
<tr>
<td>varargin</td>
<td>107</td>
</tr>
<tr>
<td>varargout</td>
<td>107</td>
</tr>
<tr>
<td>version</td>
<td>107</td>
</tr>
<tr>
<td>warning</td>
<td>107</td>
</tr>
<tr>
<td>which</td>
<td>107</td>
</tr>
<tr>
<td>while</td>
<td>12, 107</td>
</tr>
<tr>
<td>who</td>
<td>107</td>
</tr>
<tr>
<td>whos</td>
<td>107</td>
</tr>
<tr>
<td>zeros</td>
<td>107</td>
</tr>
</tbody>
</table>