Scilab Bag Of Tricks

The Scilab-2.5 IAQ (Infrequently Asked Questions)

Lydia E. van Dijk
Hammersmith Consulting

Christoph L. Spiel
Hammersmith Consulting

Scilab Bag Of Tricks: The Scilab-2.5 IAQ (Infrequently Asked Questions)
by Lydia E. van Dijk and Christoph L. Spiel

sci-BOT – the Scilab Bag Of Tricks – is a collection of Scilab experience that come from every day use. We warn of common pitfalls, discuss stylistic issues, shed light on unknown spots, and show many different ways of increasing the performance of Scilab functions.

The document is not meant to be comprehensive or even suitable to a particular level of knowledge. Some sections are at the beginners level, some even surprise long-time users.

Revision History

Revision 0.13 2000-8-2 Revised by: lvd
Minor update
Revision 0.12 2000-6-4 Revised by: lvd
First public release
Revision 0.11 2000-5-1 Revised by: lvd
First semi-public release
Revision 0.10 2000-3-17 Revised by: cls
Translated from 0.1 pod-version

Table of Contents

Preface ... **13**
- 1. Outline ... 13
- 2. Other Formats of sci-BOT ... 14
- 3. Typographic Conventions Used In This Book ... 15
- 4. Acknowledgments ... 16

1. Introduction ... **17**

2. Common Pitfalls ... **19**
- 2.1. The Infamous Dot ... 19
- 2.2. Vector Construction ... 20
- 2.3. Last Newline ... 21
- 2.4. Variable Lifetime And Scoping ... 22
  - 2.4.1. Local Variable Scoping ... 22
  - 2.4.2. Global Variables ... 26
  - 2.4.3. Clearing Variables ... 27

3. Programming Style ... **29**
- 3.1. Spacing and Formatting ... 29
  - 3.1.1. Intra-Expression Spacing ... 29
  - 3.1.2. Line Breaking ... 29
  - 3.1.3. Setting Brackets Apart ... 30
- 3.2. Indentation ... 31
- 3.3. Choice Of Control Structures ... 33
  - 3.3.1. `while`/`for` ... 33
  - 3.3.2. `if`/`select` ... 34
  - 3.3.3. Strict Block Structure/Early Return ... 35
- 3.4. Size of a Function ... 36

4. Unknown Spots ... **39**
- 4.1. Operator Precedence And Associativity ... 39
  - 4.1.1. Numeric Operators ... 39
  - 4.1.2. Relational Operators ... 40
  - 4.1.3. Logical Operators ... 40
- 4.2. Implicit Cast To Boolean ... 41
- 4.3. Functions ... 42
  - 4.3.1. Functions Without Parameters or Return Value ... 42
  - 4.3.2. Named Parameters ... 43
  - 4.3.3. Bulletproof Functions ... 44
  - 4.3.4. Function Variables ... 46
  - 4.3.5. Nested Function Definitions ... 47
  - 4.3.6. Functions as Parameters in Function Calls ... 48
  - 4.3.7. Functions in `tlists` ... 49
  - 4.3.8. `macrovar` ... 49
- 4.4. Miscellaneous Unknown Spots ... 50
  - 4.4.1. Starting **scilex** ... 50
  - 4.4.2. Tuple Assignment ... 52
  - 4.4.3. Omitting Parentheses on Function Call ... 53

5. Performance ... **55**
- 5.1. High-Level Operations ... 55
  - 5.1.1. Vectorized Operations ... 55
  - 5.1.2. Avoiding Indexing ... 56
    - 5.1.2.1. `$`-Constant ... 59
    - 5.1.2.2. Flattened Matrix Representation ... 59
  - 5.1.3. Built-In Vector-/Matrix-Functions ... 60
    - 5.1.3.1. Vector Generation ... 60
      - 5.1.3.1.1. Operator "`:`" ... 60
      - 5.1.3.1.2. `linspace` ... 61
      - 5.1.3.1.3. `logspace` ... 61
    - 5.1.3.2. Whole Matrix Construction ... 61
      - 5.1.3.2.1. `zeros` ... 61
      - 5.1.3.2.2. `ones` ... 62
      - 5.1.3.2.3. `eye` ... 62
      - 5.1.3.2.4. `diag` ... 63
      - 5.1.3.2.5. `rand` ... 64
    - 5.1.3.3. Functions Operating on a Matrix as a Whole ... 64
      - 5.1.3.3.1. `find` ... 64
      - 5.1.3.3.2. `max`, `min` ... 66
      - 5.1.3.3.3. `and`, `or` ... 67
      - 5.1.3.3.4. Operator "`&`", Operator "`|`" ... 67
      - 5.1.3.3.5. `sum`, `cumsum`, `prod`, `cumprod` ... 68
      - 5.1.3.3.6. `gsort` ... 68
      - 5.1.3.3.7. `size` ... 71
      - 5.1.3.3.8. `matrix` ... 71
  - 5.1.4. Evaluation Of Polynomials ... 73
- 5.2. Extending Scilab ... 74
  - 5.2.1. Comparison Of The Link Overhead ... 75
  - 5.2.2. Preparing And Compiling External Subroutines ... 79
    - 5.2.2.1. Fortran-77 ... 79
    - 5.2.2.2. Fortran-9x ... 80
    - 5.2.2.3. (ANSI-) C ... 81

5.2.2.4. C++ 82
5.2.2.5. Ada 84
5.2.3. Pushing It Further 85
5.2.3.1. Scilab as Prototyping Environment 85
5.2.3.2. Scilab to Fortran-77 Compiler 86
5.3. Building an Optimized Scilab 86
6. Scilab Core 89
6.1. Introduction To Pseudo-Ada 89
6.2. Internal Data Structure 90
6.2.1. Parameter Stack And Data Stack 90
6.2.2. Storage of Complex Matrices 90
6.3. Writing Native Scilab Functions 90
6.3.1. Simple Functions 90
6.3.2. Functionals 97
6.3.3. Dispatch Tables 103
6.4. Error Handling 104
6.4.1. Fatal Errors 105
6.4.2. Warnings 105
6.4.3. Messages 106
6.5. Interface to Scilab's Core 106
6.5.1. Query 107
6.5.1.1. `checkrhs` 107
6.5.1.2. `checklhs` 107
6.5.1.3. `lhs` 108
6.5.1.4. `rhs` 109
6.5.2. Access Object 109
6.5.2.1. `getmat` 109
6.5.2.2. `getrmat` 110
6.5.2.3. `getrvect` 111
6.5.2.4. `getvect` 112
6.5.2.5. `getscalar` 112
6.5.2.6. `getexternal` 112
6.5.3. Create Object 115
6.5.3.1. `Cremat` 115
7. Further Information 117
7.1. Coping With Scilab 117
7.1.1. Distribution Size 117
7.1.1.1. CVS 117
7.1.1.2. **locate** 117
7.1.1.3. Glimpse 118
7.1.2. Bug Hunting 118
7.2. Local Documents 119
7.3. Hyperlinks 121
8. Complete Examples 127
8.1. `benchmark.sci` 127
8.2. `listdiff.sci` 130
8.3. `whatis.sci` 132
8.4. Auto-Determination of Precedence and Associativity 137
8.4.1. `assoc.sci` 137
8.4.2. `prec.sci` 138
8.4.3. `parser.sci` 140
8.5. `cat.sci` 141
8.6. `quadpack.sci` 143
A. GNU Free Documentation License 147
0. PREAMBLE 147
1. APPLICABILITY AND DEFINITIONS 147
2. VERBATIM COPYING 148
3. COPYING IN QUANTITY 148
4. MODIFICATIONS 149
5. COMBINING DOCUMENTS 151
6. COLLECTIONS OF DOCUMENTS 151
7. AGGREGATION WITH INDEPENDENT WORKS 152
8. TRANSLATION 152
9. TERMINATION 152
10. FUTURE REVISIONS OF THIS LICENSE 152
B. GNU General Public License 155
0. APPLICABILITY[1] 156
1. VERBATIM COPYING 156
2. MODIFICATIONS 156
3. DISTRIBUTION 157
4. TERMINATION 158
5. ACCEPTANCE 158
6. REDISTRIBUTION 158
7. CONSEQUENCES 159
8. LIMITATIONS 159
9. FUTURE REVISIONS OF THIS LICENSE 160
10. AGGREGATION WITH INDEPENDENT WORKS 160
11. NO WARRANTY 160
12. LIABILITY 160

Bibliography **163**
Index **165**

List of Tables

4-1. Arithmetic Operators 39
4-2. Boolean Operators 41
5-1. Comparison of various vectorization levels 56
5-2. Mode Specifiers for `gsort` 69
5-3. Direction Specifiers for `gsort` 69
5-4. Performance comparison of different polynomial evaluation routines 74
6-1. pAda to Fortran-77 type mappings 89

List of Figures

5-1. Benchmark results for the `mirror` functions 78

List of Examples

2-1. Building a matrix column-by-column and row-by-row 20
2-2. Canonicalization of Scilab files 22
2-3. Shadowing of local variables 22
2-4. Accessing variables from the enclosing scope 23
2-5. Dynamic Scoping 23
3-1. Function `whocat` 31
3-2. Function `mysign` 35
4-1. Function `cat` 44
4-2. Function `tauc` 47
4-3. Manually launching **scilex** 51
5-1. Variants of a matrix mirror function 57
5-2. Naive functions to evaluate a polynomial 73
5-3. Less naive functions to evaluate a polynomial 74
5-4. Sample interface description ("`.desc`") 76
5-5. `Makefile` for static Scilab interfaces via **intersci** 77
6-1. Simple native Scilab function 92
6-2. Scilab functional 99
6-3. Handling of warnings in Scilab 106

Preface

Often we encounter technical problems that we have to solve, to overcome somehow, or just to work around. After having mastered the difficulty, we gladly add it to the knowledge-base in our mind, but from a certain level of difficulty we make notes in one form or the other. These notes then serve for later reference. A collection of related notes can be exploited to gain further insight in the class of problems it describes. Last but not least one can get ambitious to fill the holes of knowledge that an existing set of notes leaves open.

> Richard B. Johnson
>
> An expert in a particular computer language is really an expert in the work-arounds necessary to use this language to perform useful work. An ideal computer language would do exactly what it was told simply from reading a specification. In the absence of a specification, it would ask enough questions to produce such a specification, then it would generate the code necessary to perform the specified functions.
>
> ...
>
> Even C has its shortcomings which have to be handled with assembly language extensions. A Master Carpenter has many tools and is expert with most of them. If you only know how to use a hammer, every problem begins to look like a nail. Stay away from that trap. It bytes (sic).

This is the story of sci-BOT paraphrased. It started with bits of experience gathered in our heads and scattered e-mail correspondence. After more and more e-mails piled up, telling the same old stories one of the authors (lvd) decided to compile the problems and their solutions into a convenient format. Perl (www.perl.org)'s plain old documentation, POD, was chosen for its simplicity paired with a multitude of output formats. However, after 2000+ lines it became clear that POD was missing a feature that would be needed more and more as sci-BOT would grow bigger: cross references. A more powerful documentation format and the associated tools had to be found! A two week web research resulted in one winner: DocBook (http://www.oasis-open.org/docbook/). The downside of the necessary switch of formats was that the previous work done with POD had to be converted into DocBook. Daytime work plus adding new material to sci-BOT plus converting the old work into the new format is too much for a single volunteer. So, a second idiotM-DELauthor was searched and found (cls). His ten years of experience with the TeX/LaTeX typesetting system, his accuracy, and his intensity with which he attacks any obstacle made him the ideal choice for this madnessM-DELproject.

1. Outline

We open up talking about some of the most common syntactic pitfalls when using Scilab in Chapter 2. Finding that some of these syntax problems can be avoided with a clear programming style, the next chapter, Chapter 3, deals with coding issues. In Chapter 4 we then focus on the parts of Scilab that are

not well documented, and therefore widely remain unknown spots. For many users not only enjoy the nice user interface of Scilab, but demand high performance from the interpreter the massive Chapter 5 about performance issues covers these needs. It begins by introducing techniques suitable at a high level like vectorization which do not require low level programming and the dives down into the extension of Scilab by compiled routines. This is a vast field by itself. Therefore we devoted a full chapter, Chapter 6, to the low level API. sci-BOT closes with Chapter 7 containing remarks on compiling and debugging as well as comments on the supplied documentation and available web pages. All of the programming snippets that belong to longer examples which do not fit in the main text have been gathered in Chapter 8, where they show up in full length.

At the end of the document we have put two appendices with the GNU Free Documentation License, and the GNU Public License, a bibliography, and an index.

2. Other Formats of sci-BOT

sci-BOT, the Scilab Bag-of-Tricks is available as SGML, as HTML, or several "printer-ready" versions. Each variant is available in different packing-/compression formats.

Alternate sci-BOT formats

SGML source distribution

The "Real Thing" (tm)! These are our SGML-sources. Building sci-BOT from source requires XML DocBook version 4.x.

- *SGML, tar* (sci-bot-sgml.tar), *MD5* (sci-bot-sgml.tar.md5)
- *SGML, tar.gz* (sci-bot-sgml.tar.gz), *MD5* (sci-bot-sgml.tar.gz.md5)
- *SGML, tar.bz2* (sci-bot-sgml.tar.bz2), *MD5* (sci-bot-sgml.tar.bz2.md5)
- *SGML, tar.Z* (sci-bot-sgml.tar.Z), *MD5* (sci-bot-sgml.tar.Z.md5)
- *SGML, zip* (sci-bot-sgml.zip), *MD5* (sci-bot-sgml.zip.md5)

Web collection

This is sci-BOT in HTML; conveniently bundled for your offline reading pleasure.

- *HTML, tar* (sci-bot-html.tar), *MD5* (sci-bot-html.tar.md5)
- *HTML, tar.gz* (sci-bot-html.tar.gz), *MD5* (sci-bot-html.tar.gz.md5)
- *HTML, tar.bz2* (sci-bot-html.tar.bz2), *MD5* (sci-bot-html.tar.bz2.md5)

- *HTML, tar.Z* (sci-bot-html.tar.Z), *MD5* (sci-bot-html.tar.Z.md5)
- *HTML, zip* (sci-bot-html.zip), *MD5* (sci-bot-html.zip.md5)

Print versions

The printable versions are single files. By the way, you don't have to print it; it looks great with Ghostview, too.

- *PostScript, ps* (sci-bot.ps), *MD5* (sci-bot.ps.md5)
- *PostScript, ps.gz* (sci-bot.ps.gz), *MD5* (sci-bot.ps.gz.md5)
- *PostScript, ps.bz2* (sci-bot.ps.bz2), *MD5* (sci-bot.ps.bz2.md5)
- *PostScript, ps.Z* (sci-bot.ps.Z), *MD5* (sci-bot.ps.Z.md5)
- *PostScript, zip* (sci-bot.ps.zip), *MD5* (sci-bot.ps.zip.md5)
- *Portable Document Format, pdf* (sci-bot.pdf), *MD5* (sci-bot.pdf.md5)
- *Portable Document Format, pdf.gz* (sci-bot.pdf.gz), *MD5* (sci-bot.pdf.gz.md5)
- *Portable Document Format, pdf.bz2* (sci-bot.pdf.bz2), *MD5* (sci-bot.pdf.bz2.md5)
- *Portable Document Format, pdf.Z* (sci-bot.pdf.Z), *MD5* (sci-bot.pdf.Z.md5)
- *Portable Document Format, zip* (sci-bot.pdf.zip), *MD5* (sci-bot.pdf.zip.md5)

3. Typographic Conventions Used In This Book

This section covers the conventions used in this book. Depending on how the version you are currently reading some fonts may look the same.

Typographic Conventions

`filename`

This font designates the name of a file. A filename optionally includes a path.

`user input`

This font is used for the user's input. This refers only to things that can be typed in at the console.

`meta-variable`

This typeface is reserved for placeholders, i.e. stuff that always is replaced with the real input.

`literal piece of code`

We use this font to display literal pieces of code, variables, constant as well as operators.

`variable`

Variables of all kinds are marked up this way.

`function`

Functions or procedures of all kinds are marked up this way.

command

We use this font for shell commands, but also for Scilab commands.

environment-variable

To distinguish environment variables from program variables a separate font is used.

4. Acknowledgments

Lydia van Dijk: To the CCMR system administrators Daniel Blakely and Berry Robinson for providing a rock solid multi-platform environment.

Christoph "Solo" Spiel: First of all thanks go to Lydia. Before working with her I did not know whether I am insane. This project has removed all doubt. "Wonderful girl! Either I'm going to kill her, or I'm going to like her." – Han Solo about Princess Leia in "A New Hope". (Hah, I too can quote from Star Wars!)

To F. Maximilian "Tiger" Pitschi. You have shown me the difference between software engineering and hacking. Kick me again...

Chapter 1. Introduction

"I have read your posting as of ... to the Scilab newsgroup. It was very clear. Can you make a FAQ out of it?" Yes, we can, and here it is!

The hints, tricks, and information put together in sci-BOT come from our own experience (read: daily struggle), problems we have solved for our colleagues, and of course questions answered on the newsgroup. Therefore, this document is a rather loose collection of facts, and should not be read cover to cover.

What this document is not:

- An introduction to Scilab

  There already is an excellent "Introduction to Scilab", the Scilab User's Guide, SCI`/doc/Intro.ps`.

- A replacement for reading the documentation

  IONSHO ("In Our Not So Humble Opinion") folks who do not read the documentation get what they deserve. Scilab's documentation is truly great, so why not using it? To get a command's man-page type **help** at the command line. The same is achieved in the graphical environment with the Help button. If the exact command name is unknown, the powerful cousin of **help**, **apropos** jumps in. It can by used from the command line as well as from the Scilab Help Panel.

- Another FAQ list

  We do not follow the simple Question-and-Answer style. Instead we try to explore Scilab right to its very end.

In the spirit of the Free Software any helpful suggestion or correction concerning this collection will be acknowledged with the author's name and email address. If you want to tell us of a mistake, or want an item added, please drop an email at <`lydia_van_dijk@my-deja.com`>.

Chapter 2. Common Pitfalls

The nice thing about Scilab? It is almost usable!

es

There are several peculiarities in Scilab's way of interpreting an expression that will trip the unwary. Some of them are a result of "compatability" to a certain commercial product of similar sounding name, others are home grown quirks.

2.1. The Infamous Dot

In Scilab a digit in front or after the decimal point is *not* enforced. This is similar to e.g. Fortran and C, but contrary to Ada. Thus, for Scilab the following three numbers are well formed

```
87.492211
.32493
6857.
```

As an aside: `digit+.0`, `digit+.`, and `digit+` e.g. `123.0`, `123.`, and `123` are considered identical.

The last of the three examples, a decimal point at the end of the numeral, baffles users who want to invert a vector or matrix component-wise.

```
->1 ./ [1 2 3]
 ans  =
!   1.    0.5    0.3333333 !
```

Hey, but this is correct! Then, let us squeeze out the spaces in front of the `./` operator.

```
->1./ [1 2 3]
 ans  =
!   0.0714286 !
!   0.1428571 !
!   0.2142857 !
```

Oops! What happened? The last expression is not interpreted as

```
(1) ./ ([1 2 3])
```

but as


```
(1.) / ([1 2 3])
```

where the parentheses have been introduced for clarity. This behavior is described in SCI`/README`, and in the *Scilab FAQ* (http://www-rocq.inria.fr/scilab/faq/index.html).

We suggest to avoid whitespace that influences the calculation by not letting the decimal point stick out on either side. That way expressions with numerals will always be interpreted correctly. For our example this means

```
->1.0./ [1 2 3]
ans  =
!   1.    0.5    0.3333333 !
```

which gives what we had in mind.

2.2. Vector Construction

The square bracket operator `[ ]` is a convenient tool to construct vectors. There even exists an idiom to build a matrix with brackets, which is shown in Example 2-1.

Example 2-1. Building a matrix column-by-column and row-by-row

```
mat = []
for i = 1:n
    row = []
    for j = 1:m
        ...
        expr = ...
        row = [row expr]
    end
    mat = [mat; row]
end
```

Rows are separated by semi-colons or newlines, which actually is straight forward. Columns are separated by commas, or spaces—and here comes trouble.

First, comma and space serve the same purpose, and are interchangeable. Thus, the following expressions have the same result.

```
[1 2 3 4]
[1,2,3,4]
[1 2 3,4]
```

```
[  1, 2   3  ,  4 ]
```

Second, a space is sometimes considered a column-separating space, sometimes a intra-expression space. This can lead to some confusion as the following three matrix definitions demonstrate. Who gets all of them right without peeking at the answers?

```
->m1 = [1+%i -1+%i; -1+%i 1-%i]
 m1  =
!   1. + i    - 1. + i   !
! - 1. + i      1. - i   !

->m2 = [1 +%i - 1 + %i; - 1 + %i 1 - %i]
 m2  =
!   1.        - 1. + 2.i !
! - 1. + i      1. - i   !

->m3 = [1 +%i -1 + %i; - 1 + %i 1 -%i]
 m3  =
!   1.          i    - 1. + i   !
! - 1. + i      1.   - i        !
```

Confusion makes the programmer susceptible to writing code she did not intend. To make the matrix expression clear to you and to Scilab there are at least two possibilities.

1. Using no spaces in the construction of the elements of a matrix. This is e.g. demonstrated in `m1` above, or
2. Putting every compound expression in parentheses, like

   ```
   ->[(1 +%i) (-1 + %i); (- 1 + %i) (1 -%i)]
   ans  =
   !   1. + i    - 1. + i   !
   ! - 1. + i      1. - i   !
   ```

Both ways avoid the ambiguity.

Actually, matrices as simple as the ones shown in the examples can be arranged in a neat way. It is discussed in Section 3.1.2. See also Section 3.1.1 on how to improve the ligibility of Scilab code by the judicious use of whitespace.

2.3. Last Newline

The last line in a Scilab script is ignored if it is not terminated by a newline (`^J` on UNI* systems, but most of the time written in C-style `\n`). This is emphasized at several places in the official Scilab documentation, but it is so common to forget it especially when using **emacs** that we repeat it here. However, **emacs** can be told *always* to add a final newline by adding `(setq require-final-newline t)` to the startup-file, `.emacs`. See “Learning GNU Emacs” [cameron:1996], Table C-8.

Another weapon against this kind of syntax flaw, and a few other pesky things, is e.g. the Perl-script shown in Example 2-2, which fixes part of the format of a Scilab script.

Example 2-2. Canonicalization of Scilab files

```
use Text::Tabs;
while (<>) {
    chomp;                    # remove newline if there is one
    tr/\200-\377/ /;          # map 8-bit chars to spaces
    s[\s+$][];                # kill whitespace at end of line
    $_ = expand $_;           # convert tabs to spaces
    print "$_\n";             # print adding a newline
}
```

2.4. Variable Lifetime And Scoping

2.4.1. Local Variable Scoping

Scilab’s visibility rule for locally defined variables follow those of block structured languages:

> *Variables local to a block shadow all variables of the same name not local this this block.*

Variable `v` “shadows” variable `v'` means that `v'` is not accessible neither for reading nor for writing. What is available for manipulation is variable `v`.

Example 2-3. Shadowing of local variables

```
->deff('y = foo(x)', 'a = 2*x, y = a + 1')
->a = 1.0 // top level
 a  =
```

```
    1.
->foo(3.5)
 ans  =
    8.
->a
 a  =
    1.
->foo(a)
 ans  =
    3.
->a
 a  =
    1.
```

Example 2-3 demonstrates that the variable `a` which is local to function `foo` has no influence on the variable `a` in the surrounding environment. Even calling `foo` with a variable named `a` does not break this rule.

As usual in block structured languages variables from *all* enclosing scopes can be accessed, unless they are shadowed. Example 2-4 shows usage of variable `a` from an enclosing scope.

Example 2-4. Accessing variables from the enclosing scope

```
->deff('y = bar(x)', 'y = a + 1')
->a = 1 // top level
 a  =
    1.
->bar(3.5)
 ans  =
    2.
->bar(-1)
 ans  =
    2.
->a = 2
 a  =
    2.
->bar(-1)
 ans  =
    3.
```

Now what *is* the "enclosing scope"? It is the call stack; Scilab scopes dynamically!

Example 2-5. Dynamic Scoping

```
// scoping in Scilab
deff('first_local()', 'x = ""foo"", second()');
deff('first()', 'second()');
deff('second()', 'disp(x)');

x = 1;
first_local()                    // prints 'foo'
first()                          // prints 1
```

Example 2-5 deserves a close look. Dynamic scoping can be confusing for people used to e.g. C's lexically scoped `auto` variables.

```
/* lexical scoping in C */

void first_local(void);
void first(void);
void second(void);

int x = 1;

int
main(void)
{

    first_local();               /* prints 1 */
    first();                     /* prints 1 */

    return 0;
}


void first_local(void)
{
    int x = 123;                 /* warning: unused variable `x' */
    second();
}

void first(void)
{
    second();
}
```

```
void second(void)
{
    printf("%d\n", x);
}
```

Now compare this to Perl[1]:

```
# dynamical scoping with Perl's local variables

sub first_local { local $x = 'foo'; second(); }
sub first { second(); }
sub second { print "$x\n"; }

$x = 1;
first_local();                    # prints 'foo'
first();                          # prints 1
```

Dynamic scoping is an inherently dangerous feature for it might not be obvious where a variable gets its value.

Let us look at functions which try to change variables from an enclosing scope.

```
->deff('y = baz(x)', 'a = 2*a, y = a + 1')
->a = 3 // top level
 a  =
    3.
->baz(1)
 ans  =
    7.
->baz(1)
 ans  =
    7.
->a
 a  =
    3.
```

Obviously, `a` is unchanged by the calls to `baz`. What happens is the following:

1. A local variable named `a` is created, and the contents of variable `a` from the enclosing scope is copied into it. Within `baz` the local `a` is changed.
2. When the thread of control leaves `baz` the previous value of `a` is restored.

In other words: A local variable *cannot* influence a variable of the same name in any enclosing scope. The only ways to "export" a – possibly modified – value is either via the list of return values, which is the preferred way, or with a `global` variable.

As strange as this may sound to programmers accustomed to languages that require an explicit declaration of all variables, this is a necessary feature in Scilab as variables are created when they are first written to (e.g. as in Python). If a local variable in a function would change a global variable or local variable of the same name in an other function, adding a new function to an existing system or library became a major maintenance headache.

2.4.2. Global Variables

The `global` attribute of a variable *`var`* is often misunderstood. It does *not* place *`var`* in an all encompassing name space so that it could be accessed from everywhere without further ado. Instead, `global` places the variable *`var`* in a separate name space; separate from the interpreter's name space, and separate from all local functions' name spaces. — and this is only the first half of the story.

```
->v = -1
 v  =
  - 1.

->global('v')

->who('global')
 ans  =
 v

->clear v

->who('global')
 ans  =
 v

->deff('y = useglobal()', 'y = v')

->useglobal()
 !-error 4
undefined variable : v
at line 2 of function useglobal called by :
useglobal()
```

As promised, this is only one half. After saying `global` *`var`* the variable lives in its new name space, but it cannot be accessed. Doh! To work with it, we must import it explicitely, using the `global` modifier again. Therefore, a slightly modified version of `useglobal` works.

```
->deff('y = useglobal2()', 'global v, y = v')
```

```
->useglobal2()
 ans  =
  - 1.

->v = 1 + 2*%i
 v  =
    1. + 2.i

->useglobal2()
 ans  =
  - 1.
```

Now what if we want to access `v` from the interpreter level again? It must be imported just as it must be imported into any function.

```
->global('v')

->v
 v  =
  - 1.

->v = 17 + 4
 v  =
    21.

->clear v

->useglobal2()
 ans  =
    21.
```

One last hint: global variables even "survive" a restart. If this is not desired, `clearglobal` should be called in the user's Scilab startup file, `~/.scilab`.

```
clearglobal()
```

will clear all global variables.

2.4.3. Clearing Variables

During everyday programming it is not necessary to explicitely remove variables from the work space. All local variables of a function die on exit from that function anyhow, and the variables in the global name space usually do not need a special treatment.

However, there are conditions under which it is preferable to completely wipe out a variable. This happens if one needs to avoid a pollution of the name space e.g. while working with the list of all variables, `who('local')`. The correct command to kill the non-global variable `v` is

```
clear v
```

Note that there are no parentheses. The assignment

```
v = []
```

sets `v` to the empty matrix. It does *not* remove the variable from the workspace.

Global variables are cleared with the `clearglobal` function, whose syntax is the same as `clear`'s syntax.

There is no need to worry if you do not understand how and why to kill a variable. This feature is only needed in very rare occasions.

Notes

1. The behavior of the C example is reproduced by replacing `local` with `my`.

Chapter 3. Programming Style

The one and only general guideline to good programming style is: "Make it clear!" And one might extend that to

> Make it clear – first of all to you, and then to the poor persons that take over your project (after you have been fired, because of writing illegible code).

Every possible style feature of the language should be used to express the meaning of the code more clearly.

3.1. Spacing and Formatting

Although often underestimated, the format, i.e. the visual layout of the source code itself can greatly help in the understanding of the actions described therein.

3.1.1. Intra-Expression Spacing

We often run into code like this

```
x=a*c+(x-y)^2*b
```

This is not bad, especially when typed at the command line for one-time use. However, the expression is not as clear as it could be. It can easily be improved by making the precedence levels of the operators stand out, as e.g.

```
x = a*c + b*(x-y)^2
```

Now, the assignment is intuitively clear at first glance. We use word "intuitive" here to make the reader alert of the consequences of formatting an expression the wrong way. Then our intuition will mislead us, as in

```
x = a * c+(x-y)^2*b
```

Outch! This expression is evaluated differently from what it is telling us. We should call it a liar.

See also Section 2.2 for the influence of whitespace on the evaluation of dotted operators.

3.1.2. Line Breaking

Breaking a long expression into lines can improve the readability dramatically. It is particularly recommended for matrix definitions with the square bracket operator. See also Section 2.2.

```
m1 = [ 1+%i  -1+%i; ..
      -1+%i   1-%i ]
```

is superior to

```
m1 = [1+%i -1+%i; -1+%i 1-%i]
```

If an arithmetic expression is split into lines the operator at which the split occurs always goes onto the *next* line. Preferred break points occur right before operators of equal precedence.

```
d2 = fact * (a/(a+d)*(b*(1-delta) + d*delta) - d) * (P./K).^theta
```

for example becomes

```
d2 = fact * (a/(a+d)*(b*(1-delta) + d*delta) - d) ..
     * (P./K).^theta
```

or

```
d2 = fact ..
     * (a/(a+d)*(b*(1-delta) + d*delta) - d) ..
     * (P./K).^theta
```

or more dramatic

```
d2 = fact ..
     * ( ..
            a / (a+d) * (b*(1-delta) + d*delta) ..
            - d ..
       ) ..
     * (P./K).^theta
```

The last way of breaking the expression is very Lisp-like.

3.1.3. Setting Brackets Apart

If spaces right inside the parentheses or brackets of an expressions make the subexpression stand out more clearly, they should be used. That way

```
B(k) = a1 * exp(-b1*P(k)/K(k) + b2*Q(k)/K(k))
```

becomes

```
B(k) = a1 * exp( -b1*P(k)/K(k) + b2*Q(k)/K(k) )
```

3.2. Indentation

Heavy indentation does not hurt! No, in fact it is a great help in finding out the control flow quickly. Let us start with a good example this time, Example 3-1.

Example 3-1. Function `whocat`

```
function s = whocat(cat)
// return all local variables, functions,
// etc. that are in category cat.

s = [];
nl = who('local');

for i = 1:size(nl, 1)
        execstr( 'typ=type(' + nl(i) + ')' );
        if typ == cat then
                s = [s; nl(i)];
        end
end
```

The `for` loop, and the `if` branch are immediately recognizable.

There are blank lines between the logical blocks of the function. They too aid the reader's comprehension of `whocat`'s inner workings. As a rule of thumb lines of code that achieve a sub-goal of the computation should be grouped together as sentences are grouped in a paragraph.

In longer functions the indentation becomes essential for the orientation of the maintainer. Here is a excerpt of a longer function, that would be terribly hard to understand if not massively indented.

```
i = 1;
j = 1;
while i <= n1 & j <= n2
        while i <= n1 & j <= n2
                if ~equ(lst1(i), lst2(j)), break, end
```



```
                i = i + 1;
                j = j + 1;
        end
        if i >= n1 | j >= n2, break, end

        icurs = i;
        while icurs <= min(n1, i+fuzz)
                if equ(lst1(icurs), lst2(j)), break, end
                icurs = icurs + 1;
        end
        if icurs <= n1 then
                if equ(lst1(icurs), lst2(j)) then
                        // record element(s) missing from lst1
                        for p = i : icurs-1
                                this_diff = [lst1(p), string(-p)];
                                diff = [diff; this_diff];
                        end
                        // re-sync
                        i = icurs;
                end
        end
        ...
end // while
```

The complete listing of this function can be found in Chapter 8.

The last example also shows that we are switching between several style paradigms:

- Neither the "One statement per line" rule is followed consistently,

  ```
  if equ(lst1(icurs), lst2(j)), break, end
  ```

  could be

  ```
  if equ( lst1(icurs), lst2(j) ) then
      break
  end
  ```

- Nor is the intra-line spacing always consistent with the guidelines presented here:

  ```
  for p = i : icurs-1
  ```

  could be

```
for p = i:icurs-1
```

The Golden Rule is that there are no golden rules... This is best known under the term ‘freedom’.

3.3. Choice Of Control Structures

Though not recognized as that by all programmers the flow control structures themselves are first class indicators of the codes workings. We consider three important cases here.

1. `while` vs. `for`,
2. `if` vs. `select`, and
3. strict block structure vs. premature return.

3.3.1. `while`/`for`

Expressed in words a `for` loop tells us:

- We know exactly how many iterations we shall need before we start looping.
- Nothing in the loop body will change this.

Whereas the `while` loop says:

- We must check whether we should loop at all, and
- we have to re-check after each iteration whether we need another round-trip.

Corollary: The termination condition of a `while` must be influenced in the loop’s body.

Compare the next two code snippets, the first one calculating the average value of a vector of numbers, the second searching zeroes of a given function.

```
values = [1.0, 2.0, 3.0, 4.0, 5.0];
average = 0.0;
n = size(values, 'c');             // line 3
for i = 1:n
    average = average + values(i);
end;
average = average / n
```


From line 3 on, we know the number of iterations, `n`, and we know that nothing will change that. Thus a `for`-loop is adequate.

```
deff('[y, dy] = fun(x)', ..
     'y  = -0.5 + 1.0/(1.0 + x^2), ..
      dy = -2.0 * x / (y + 0.5)^2');

x0 = 0.76;
[y, dy] = fun(x0);
while abs(y) > sqrt(%eps)
    x = y/dy - x0;
    x0 = x;
    [y, dy] = fun(x);
end;
x
```

Assuming that the function `fun`, and the start guess `x0` are supplied by the user, we do not know how many loops it will take for Newton’s algorithm to converge, if it does converge at all. (In the example it does.) Here, the `while`-loop expresses this lack of a-priori knowledge.

3.3.2. `if`/`select`

The relationship between `if` and `select` bears similarity to `while` and `for`, respectively. In a `select` clause the different cases are known – and spelled out explicitely – before the thread of control enters the construct. There is a one to one relationship between the states of the `select`ing expression and the `case` branch taken. The `else` branch in a `select` works exactly as the `else` in an `if`.

```
function f = fibonacci(n)
// return n-th Fibonacci number

select n
case 0 then
    f = 1
case 1 then
    f = 1
else
    f = fibonacci(n - 1) + fibonacci(n - 2)
end
```

The `select`ing expression is not restricted to scalars, vectors for example work too:

```
function s = shape4(m)
// classify a 2x2 matrix according to its shape
```

```
select abs(m) <= %eps
case [%t %t; ..
      %t %t] then
    s = 'empty'
case [%t %f; ..
      %f %t] then
    s = 'diagonal'
case [%f %f; ..
      %t %f] then
    s = 'upper triangular'
case [%t %t; ..
      %f %t] then
    s = 'lower triangular'
case [%f %f; ..
      %f %f] then
    s = dense'
else
    s = 'general'
end
```

An `if` clause is more flexible than a `select` clause, but at the price of being less expressive. Whenever a whole range of values has to be covered the `if` clause is the only way to go.

Example 3-2. Function `mysign`

```
function y = mysign(x)
// re-write of the sign-function, taking
// floating-point precision into account

if abs(x) < %eps
    y = 0.0
elseif x >= %eps
    y = 1.0
else
    y = -1.0
end
```


3.3.3. Strict Block Structure/Early Return

The paradigm of structured programming is: "Every block has one and only one entry point." That's it! Nothing is said about the number of exit points. The purists often misinterpret the paradigm, demanding a single exit point, too. We prefer our freedom, and choose whatever we find adequate to the problem.

Here are two different implementations of an algorithm calculating the factorial of a given integral number.

```
function y = fact_block(x)
// faculty of x; block-structured version

select x
case 0 then
    y = 1
case 1 then
    y = 1
else
    y = x * fact(x - 1)
end
```

The two special cases `0`, and `1` are tested separately, and the general case is handled in the `else` branch.

```
function y = fact_early_ret(x)
// faculty of x; early-return version

if x >= 0 & x <= 1 then
    y = 1
    return
end

y = x * fact(x - 1)
```

This version immediately returns after having treated the special cases, leaving the general case to the "rest" of the function. In this very short function the advantages of the early return are not striking, however they are if many special cases are to be handled. The "rest" of the function can then concentrate on the core of the problem without being obscured by deeply nested conditionals.

3.4. Size of a Function

There is a rule of thumb for the length of a C-function:

L. Torvalds

> Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

It is also true for Scilab functions with the exception that high level functions, or functions that are called from the command line directly should be harnessed, see Section 4.3.3. Therefore, they are usually much longer than just two screenful. Yet their structure decomposes quite naturally into two parts: the argument checking, and the computation part. What remains true is that a Scilab function too should do only one thing and do that well.

For more information about programming style consult “The practice of programming” [kernighan:1999] which is centered around C-like languages, but offers extremely valuable advice throughout. “Programming Perl”, also known as “The Camel”, [wall:1996] has a section called “Efficiency” in chapter 8. It is as insightful as it is fun to read for the authors discuss the various optimization directions. They do not hesitate to put up contradictory suggestions in the different optimization paths.

Conclusion of this section: Whatever makes the code’s workings more obvious to the reader is good. In other words: “If it makes ya high, or saves you taxes, then – by any means – do it!”

Chapter 4. Unknown Spots

In this chapter we shed some light onto widely unknown features. Parts like the operator precedence unconsciously are exploited in every-day programming by all of us. Others like the use of function variables are truly unknown, at least to the average Scilab user. So, read on and become a Yedi^H^H^H^HScilab master.

4.1. Operator Precedence And Associativity

Strange but true, there is no listing of the precedence and associativity of neither class of Scilab's operators anywhere in the documentation. So, we discuss the operator precedence and associativity in detail.

4.1.1. Numeric Operators

Table 4-1 displays a list of all numeric operators up to digraphs[1], sorted in descending order of their precedence. An equal precedence value (column 1) means the operators are evaluated following the given associativity (column 3).

The table is generated with a Scilab script, i.e. we had the interpreter determine its own precedence rules, which is neat. These scripts are listed in Chapter 8.

Table 4-1. Arithmetic Operators

precedence	operator	associativity	comment
21	+	right	unary
20	^	right	
20	.^	right	
19	–	right	unary
8	*	non	
8	/	left	
8	.*	non	
8	./	left	
4	\	left	
4	.\	left	

precedence	operator	associativity	comment
1	+	non	binary
1	–	left	binary

Warning

One line begs for an additional warning, and that is the unary minus ranking at level 19. It looses against the power operator, `^`. Therefore, `-1^2` gives `-1`, and not `1`. In other words Scilab sees `-1^2` as `-(1^2)`.

The association rules follow those of standard algebra. Thus, nobody should be surprised that `a^b^c` is interpreted as `a^(b^c)`.

4.1.2. Relational Operators

Scilab implements the usual gang of relational operators with some syntactic sugar of having two "unequality"-operators <>, and `~=`. The relational operators' precedences rank in between the numeric and the logical operators like they do in many other modern programming languages. This allows for a minimal use of parentheses in larger expressions like

```
if 2.0*n > l+1.0  |  n/3.0 <= k then
    ...
end
```

which evaluates exactly the same way as

```
if ((2.0 * n) > (l + 1.0)) | ((n / 3.0) <= k) then
    ...
end
```

just with much less line-noise.

4.1.3. Logical Operators

There are three logical operators: `&`, `|`, and `~`, meaning "and", "or", and "not". The twiddle, `~` has the unique syntactic property that any number of consecutive twiddles are allowed and evaluated. But unless you want to enter the obfuscated Scilab contest, sticking with one probably is best as e.g. 15 ~ are as good as one, and therefore

```
~~~~~~~~~~~~~~~%t
```

returns `F`.

Table 4-2 shows the complete list of Scilab's logical, also known as boolean, operators sorted according to decreasing precedence.

Table 4-2. Boolean Operators

operator	associativity	comment
~	right	unary
&	non	
|	non	

4.2. Implicit Cast To Boolean

For the logical operators have boolean expressions as their arguments, it is time now to discuss the implicit promotion of numeric types to boolean type, something very familiar to C, Perl, and Python programmers. You have guessed right, the rule is: "Zero is false, everything else is true." Here are some examples of that rule at work:

```
->%t & 0
 ans  =
  F

->%t & 0.1
 ans  =
  T

->6.34 | %f
 ans  =
  T

->6.34 | -0.3
 ans  =
  T
```

Scilab always evaluates boolean expressions completely. No operator is defined with short-circuit evaluation semantics.


```
->deff('b = ret_false()', 'b = %f, disp("ret_false")');

->ret_false() & ret_false()

 ret_false

 ret_false
 ans  =

  F
```

4.3. Functions

Functions are Scilab's the main abstraction feature, thus they deserve a closer look.

4.3.1. Functions Without Parameters or Return Value

The "Introduction to Scilab", SCI`/doc/Intro.ps`, solely explains functions that have one or more parameters, and return one or more values. Yet, Scilab permits all conceivable combinations of number of parameters and return values, including those functions that have no parameters, or no return values.

If only one value is returned the square brackets in the function definition are optional. Therefore, the function head ("declaration")

```
function [y] = foo(x)
```

can be abbreviated to

```
function y = foo(x)
```

However, this is 100% pure syntactic sugar. What is much more important – and a valuable feature – is the possibility of defining a function that returns nothing as

```
function ext_print(x)
printf("%f, %g", x, x)
```

does. In Fortran parlance `ext_print` would be called a `SUBROUTINE`, whereas Ada programmers would term it a `PROCEDURE`.

Of similar importance is the definition of parameterless functions.

```
function t = hires_timer()
```

```
cps = 166e6
t = rdtsc() / cps
```

The parentheses after the function name are optional when defining the function, but not when calling it, thus the declaration of the last function could have been abbreviated to `function t = hires_timer`, but the call to `rdtsc` could not have been written as `t = rdtsc / cps`.

For further information about the omission of parenthesis when calling a function, see Section 4.4.3.

4.3.2. Named Parameters

The associations between the formal parameters of a function and its actual parameters may be positional or named. A positional parameter association is simply an actual parameter. All the positional parameter associations in a function call must precede all the named parameter associations. Thus, in the function call

```
myplot(x, y, pointtype = 4, style = 'linespoints', linetype = 2)
```

the first two parameter associations (`x`, `y`) are positional, and the last three (`style`, `linetype`, `pointtype`) are named. Two points in the previous line of code are worth noting:

- When parameters are associated via their names the formal parameter's position is irrelevant.
- Positional parameter associations have nothing to do with optional parameters. A named parameter can be handled as an optional parameter as well as a positional parameter.

Calling a function with named parameters does not require any special code in the function. Function `myplot` is just an ordinary user-defined function:

```
function myplot(x, y, style, linetype, pointtype)

// a check for optional parameters would go here

select style
case 'lines' then
    plot2d(x, y, linetype)
case 'points' then
    plot2d(x, y, -pointtype)
case 'linespoints' then
    plot2d(x, y, -pointtype, '020')
    plot2d(x, y, linetype, '000')
end
```


To make the two parameters `linetype`, and `pointtype` optional parameters, we add a check for the existence of these parameters in the function's, i.e. the local scope.

```
function myplot(x, y, style, linetype, pointtype)

if ~exists('linetype', 'local')
    linetype = 1
end
if ~exists('pointtype', 'local')
    pointtype = 1
end

select style
case 'lines' then
    plot2d(x, y, linetype)
case 'points' then
    plot2d(x, y, -pointtype)
case 'linespoints' then
    plot2d(x, y, -pointtype, '020')
    plot2d(x, y, linetype, '000')
end
```

Now `myplot` can be called in any of the following forms:

```
myplot(x, y, 'lines')                 // only positional parameters
myplot(x, y, style = 'linespoints')  // 3rd parameter is named
myplot(x, y, 'points', 2, 3)         // override defaults
myplot(x, y, linetype = 5, ..
       style = 'linespoints')        // named params, one override
myplot(x, y, pointtype = 4, ..
       style = 'linespoints', .
       linetype = 2)                 // named params where possible
```

4.3.3. Bulletproof Functions

If we want to write bulletproof Scilab functions, we have to take care that our functions get the right number of arguments which are furthermore of the correct type, and correct dimension. This is due to Scilab's dynamic nature allowing us to pass arguments of different types, dimension etc. to a single function.

We discuss the issues of writing robust function using Example 4-1 as an illustration. The complete function definition is given in Chapter 8.

Example 4-1. Function `cat`

```
function [res] = cat(macname)
// Print definition of function 'macname'
// if it has been loaded via a library.

[nl, nr] = argn(0);                        ❶
if nr ~= 1 then
    error('Call with: cat(macro_name)');
end

if type(macname) ~= 10 then                ❷
    error('Expecting a string, got a ' ..
          + typeof(macname));
end
if size(macname, '*') ~= 1 then            ❸
    sz = size(macname);
    error('Expecting a scalar, got a ' ..
          + sz(1) + 'x' + sz(2) + ' matrix')
end

[res, err] = evstr(macname);               ❹
if err ~= 0 then
    select err
    case 4 then
        disp(macname + ' is undefined.');
        return;
    case 25 then
        disp(macname + ' is a builtin function');
        return;
    else
        error('unexpected error', err);
    end // select err
end // err ~= 0

...
```

❶ First, we check how many actual parameters function `cat` has got. The built-in function `argn` returns the number of left-hand side – or output – variables `nl` (In this example we do not make use of `nl`.), and the number of right-hand side – or input – values `nr`.

Ensuring the correct number of *input* arguments always is the first step. Otherwise we cannot assume whether even accessing a parameter is valid. The number of output values is not as critical,

for calling a function with less output variables than specified in the function's signature causes the extra output values to be silently discarded.

After learning the number of actual parameters, we immediately check whether it is in the right range. In our example simply terminate with an error if the number of arguments is incorrect.

❷ The next thing to address are the types of the arguments. Again we let the function fail with an error if it does not get what it wants, but this is not the only possible way.

It is conceivable that we convert from one type to another, say from numeric to string. Furthermore, it is possible that the type of the arguments determines the algorithm chosen, a feature normally advertised under the name "function overloading".

❸ Finally, we examine the arguments' structure. A function can e.g. allow scalars only, or accept scalars and matrices. Here, we enforce a scalar. In other functions certain dimensional relations of several input parameters must be enforced. E.g. the matrix multiplication *A* * *B* is only defined for `size(`*A*`, 'c') == size(`*B*`, 'r')`.

❹ Now we can start with the real work.

At first glance all this checking gizmos might seem exaggerated. To do it justice we should keep in mind that it is only necessary if a function must work reliably in different environments. All functions that a library exports belong to that class, because the library writer does not know how the functions will be used in the future. Quick-and-dirty functions are a different thing, so are functions that are never called interactively.

4.3.4. Function Variables

Functions are a data type on their own right. Therefore, they themselves can be arguments to other functions, and can be elements in lists.

```
->deff('y = fun(x)', 'if x > 0, y = sin(x); else, y = 1; end')

->fun(%pi/2)
 ans       =
    1.

->fun(-3)
 ans       =
  - 1.

->bar=fun
 bar       =
```

```
[x]=bar(y)

->typeof(bar)
 ans       =
 function

->deff('a = fun(u, v, w)', 'a = u^2 + v^2 + 2*u*v - w^2')
Warning :redefining function: fun

->bar(%pi/4)^2
 ans       =
    0.5

->fun(2, 3, 4)
 ans       =
    9.
```

As the example shows Scilab employs its usual copy-by-value semantics when assigning function-variables, this is consistent with the assignment of any other data type.

4.3.5. Nested Function Definitions

Function definitions can be nested. The usual scoping rules apply. Online nested function definitions are some kind of awkward because of the massive number of quotes, but `deff`s in `function`s are easy to the eye.

Example 4-2 shows a function that defines four functions in its body.

Example 4-2. Function `tauc`

```
function [t, rmin, r0] = tauc(E0, M, s, D)

deff('U = Umorse(r, steepness, depth)', ..
     'e = exp(-r * steepness); ..
      U = depth*(e^2 - 2*e)');

// point of vanishing potential
deff('y = equ0(x)', 'y = Umorse(x, s, D)');

// reflection point
deff('y = equ1(x)', 'y = Umorse(x, s, D) - E0');

deff('tau = integrand(x)', ..
```



```
     'tau = sqrt( M / (2*(E0 - Umorse(x, s, D))) )');

// rationalized units...
units = 10.0e-10 / sqrt(1.380662e-23 / 1.6605655e-27);

// calculate endpoints of definite integral
r0 = fsolve(-10.0, equ0);
rmin = fsolve(-10.0, equ1);

// evaluate definite integral
[t_unscaled, err] = intg(rmin, r0, integrand);
t = 2 * units * t_unscaled;
```

4.3.6. Functions as Parameters in Function Calls

As mentioned above, user-defined functions can be passed as parameters to (usually different) functions. Builtin functions have to be “wrapped” in user-defined functions before they can be used as parameters.

The following example defines a functional that implements a property of Dirac’s delta distribution.

```
->deff('y = delta(a, foo)', 'y = foo(a)')

->delta(cos)
        !-error    25
bad call to primitive :cos

->deff('y = mycos(x)', 'y = cos(x)')

->delta(0, mycos)
 ans  =
    1.
```

The next example is a bit more convoluted, but also closer to the real world. We define a new optimizer function, called `minimize`, which is based on Scilab’s `optim` function. Function `minimize` expects two vectors of data points `xdata` and `ydata`, a vector of initial parameters `p_ini`, the function to be minimized `func`, and an objective functional `obj`.

The advantage of defining separate model and objective functions is an increased flexibility as both can be replaced at will without changing the core minimization function `minimize`.

```
function [f, p_opt, g_opt] = minimize(xdata, ydata, ..
                                      p_ini, func, obj)
```

```
// on-the-fly definition of the objective function
deff('[f, g, ind] = _cost(p_vec, ind)', ..
     '[f_val, f_grad] = func(xdata, p_vec); ..
      [f, g] = obj(f_val - ydata, f_grad)');

[f, p_opt, g_opt] = optim(_cost, p_ini);
```

Function `minimize` needs a model function `func` that returns the value and the gradient at all points `x` for a given vector of parameters `p_vec`. Moreover, we need the objective functional `obj` that gives the "cost", as well as the direction of steepest descent in parameter space.

In this example we choose a quadratic polynomial for the model, `my_model`, and least squares for the objective `lsq`.

```
function [f, g] = my_model(x, p)
g = [ones(x), x, x.*x];
f = p(1) + x.*(p(2) + x*p(3));

function [f, g] = lsq(diff, grad)
f = 0.5 * norm(diff)^2;
g = grad' * diff;
```

Given these definitions, we can call `minimize`:

```
dx = [0.0 1.0 2.0 2.5 3.0]';
dy = [0.0 0.9 4.1 6.1 9.5]';
p_ini = [0.1 -0.2 0.9]';
[f_fin, p_fin, p_fingrad] = ..
    minimize(dx, dy, p_ini, my_model, lsq)

xbasc();
plot2d(dx, dy, -1);          // plot data points ...
xv = linspace(dx(1), dx($), 50)';
yv = my_model(xv, p_fin);
plot2d(xv, yv, 1, '000');    // ... and optimized model function
```

4.3.7. Functions in `tlist`s

Currently the only complex data structure that allows for storage of `function`s is the typed list `tlist`.

FIXME: write it

4.3.8. `macrovar`

The `macrovar` function could be called the functional cousin of the `size` function. The primary purpose of `macrovar` is to support the Scilab-to-Fortran translator, but it can be useful for other purposes, too.

Function `macrovar` reveals five important attributes of a user function. These are the names of all

- input variables,
- output variables,
- global variables,
- functions called, and
- local variables.

One example of an interesting use of `macrovar` is an integration routine that accepts integrand functions with an arbitrary number of arguments, i.e. over arbitrary many dimensions.

```
function vol = int_cube(ifun)
// integrate ifun in an appropriate hypercube
// (0, ..., 0), ..., (1, ..., 1)

ifun_var = macrovar(ifun)
ifun_sz  = size(ifun_var(1)) // names of input arguments
ifun_dim = ifun_sz(1)

for d = 1:ifun_dim
     // integrate in one dimension
end
```

4.4. Miscellaneous Unknown Spots

4.4.1. Starting scilex

For debugging purposes it is sometimes desirable to directly start the main Scilab binary, **scilex**. Scilab is usually launched via the `scilab` shell script. Both, the script and the binary live in the SCI`/bin` directory. The script takes care of setting all environment variables, and finally fires up **scilex**. On the other hand, if one wants to run a debugger, say **gdb**, or **ddd**, or a profiler on Scilab, then a manual invocation is the order of the day. See also Section 7.1.2.

Starting **scilex** is easy as long as the command-line editing goodies are not required, and there is no need for any graphics. Actually, for minimum functionality only the environment variable SCI must be set, then we are all set to call **scilex**. A **bash** sequence to start Scilab "manually" could look as shown in Example 4-3.

Example 4-3. Manually launching scilex

```
lydia@orion:~$ cd /site/X11R6/src/scilab
lydia@orion:/site/X11R6/src/scilab$ SCI=`pwd`
lydia@orion:/site/X11R6/src/scilab$ export SCI
lydia@orion:/site/X11R6/src/scilab$ cd bin
lydia@orion:/site/X11R6/src/scilab/bin$ ./scilex -nw
                           ===========
                           S c i l a b
                           ===========


                          Scilab-2.5
                  Copyright (C) 1989-99 INRIA


Startup execution:
  loading initial environment

->
```

or shorter

```
lydia@orion:~$ export SCI=/site/X11R6/src/scilab
lydia@orion:~$ $SCI/bin/scilex -nw
                           ===========
                           S c i l a b
                           ===========


                          Scilab-2.5
                  Copyright (C) 1989-99 INRIA


Startup execution:
  loading initial environment

->
```


where we are assuming that Scilab is installed in `/site/X11R6/src/scilab`.

4.4.2. Tuple Assignment

The most commonly used form of assignment is single variable assignment. Nonetheless, assigning multiple values in one statement is possible (and no surprise for Perl or Python programmers).

```
->[x1 x2 x3] = (1, 2, 3)
 x3  =
    3.
 x2  =
    2.
 x1  =
    1.
```

Tuple assigment works as expected, performing the whole assigment operation in one step. The online documentation gives the wrong explanatory code

> `[x1,x2,...]=(e1,e2,...)` is equivalent to `x1 = e1`, `x2 = e2`, ...

The correct explanation is

> `[x1, x2, ...] = (e1, e2, ...)` is equivalent to `%t1 = e1`, `%t2 = e2`, ..., and then `x1 = %t1`, `x2 = %t2`, where the variables `%t`*N* are invisible to the user.

Otherwise the following example, swapping values of two variables, would result in `a` and `b` both being equal to `2`.

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

    1.
 b  =

    2.

->[b, a] = (a, b) // swap
 a  =
```

```
    2.
 b  =

    1.
```

See the online documentation, `help parents`.

What one might expect, but what does not work is multiple assignment to parts of matrices (or lists), i.e. the following code snipped does not work as naively expected

```
->v = [0 0 0], a = 0
 v  =
!   0.    0.    0. !
 a  =
    0.

->[a, v(1)] = (1, 2)
 Warning: obsolete use of = instead of ==
            !
                   !-error 41
incompatible LHS
```

The obvious, but ugly workaround is using only scalar variables on the left-hand side of a aggregate assignment, and then assigning these scalars to the appropriate matrix or list parts.

4.4.3. Omitting Parentheses on Function Call

The parentheses of any one-parameter function can be omitted, if the function accepts a string argument. Moreover, the quotes for a literal string argument can be left out, too.

The is especially useful, when working interactively, and loading functions, or scripts. There is no need to type until your fingers bleed by saying

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

as the next two examples work just as well.

```
->getf 'foo.sci'
```

and even

```
->getf foo.sci
```

is OK. Note that this is not only true for built-in, but also for user-defined functions.

Function `exec` is an exception to the rule that a semicolon suppresses any output of the preceeding clause, if it is invoked without parenthesis.[2] In fact, `exec` does echo the commands it executes if used without parenthesis *despite* a trailing semicolon, i.e.

```
->exec script.sci;
```

with semicolon gives same results as

```
->exec('script.sci')
```

without semicolon, whereas

```
->exec('script.sci');
```

does not echo the commands of the script file.

Notes

1. The trigraph operators `.*.`, `./.`, and `.\.` are left out.
2. Thanks to Glenn Fulford for reporting this.

Chapter 5. Performance

Scilab—The fastest thing from France since Django Reinhardt.

cls

In this chapter we discuss how expressions can be written to execute more quickly while doing the same thing. Scilab is powerful and flexible, therefore there are plenty of things one can do to speed up function execution. On the downside there are a lot of things the can be done the wrong way, slowing down the execution to a crawl.

In the first part of this chapter we focus on high-level operations that are executed fast. The main class to name here are vectorized operations. Another class are all functions that are constructing or manipulating vectors or matrices as a whole. The second part of this chapter deals with the extension of Scilab through compiled functions for the sake of increased execution speed. We close with a section on how to compile Scilab itself to increase its performance.

5.1. High-Level Operations

Not using vectorized operations in Scilab is the main source for suffering from a slow code. Here we present performance comparisons between different Scilab constructs that are semantically equivalent.

5.1.1. Vectorized Operations

The key to achieve a high speed with Scilab is to avoid the interpreter and instead make use of the built in vectorized operations. Let us explain that with a simple example.

Say we want to calculate the standard scalar product `s` of two vectors `a` and `b` which have the same length `n`. Naive as we are, we start with

```
s = 0                                   // line 1
i = 1                                   // line 2
while i <= n                            // line 3
    s = s + a(i) * b(i)                 // line 4
    i = i + 1                           // line 5
end                                     // line 6
```

Here Scilab re-interprets lines 3 to 5 in every round-trip, which in total is `n` times. This results in slow execution. The example utilizes no vectorization at all. On the other hand it uses only very little memory memory as no vectors have to be stored.

The first step to get some vectorization is to replace the `while` with a `for` loop.

```
s = 0                                   // line 1
for i = 1:n                             // line 2
    s = s + a(i) * b(i)                 // line 3
end                                     // line 4
```

Line 2 is only interpreted once; the vector `i = 1:n` is set up and the loop body, line 3 is threaded over it. So, only line 3 is re-evaluated in each round trip.

OK, it is time for a really fast vector operation. In the previous examples the expression in the loop body has not been modified, but we can replace it with the element wise multiplication operator `.*`, and replace the loop with the built-in `sum` function. (See also Section 5.1.3.3.)

```
s = sum(a .* b)
```

One obvious advantage is, we have a one-liner now. Is that as good as it can get? No, the standard scalar product is not only a built-in function it is also an operator:

```
s = a * b'
```

We summarize the timing results of a PII/330 Linux-system in Table 5-1.

Table 5-1. Comparison of various vectorization levels

construct	MFLOPS
`while`	0.005
`for`	0.008
`.* and sum`	1.7
`*`	2.8

In other words the speed ratio is 1:1.6:330:550. Of course the numbers vary from system to system, but the general trend is clear.

5.1.2. Avoiding Indexing

Accessing a vector- or matrix-element via indexing is slow. Sometimes the index cannot be avoided, but there are cases where it can. Compare

```
for i = 1:n
    v(i) = i
end
```

and

```
v = []
for i = 1:n
    v = [v, i]
end
```

The second snippet is not only faster, but in some circumstances may be clearer. Again there is a built-in operator that does the same job at lightning speed, the colon `:`, which is described in detail in Section 5.1.3.1.

```
v = 1:n
```

The speed ratio is approximately 1:1.5:5000.

In the next example, Example 5-1, the functions actually try to do something useful: they mirror a matrix along its columns or rows. We show different implementations of `mirrorN` that all do the same job, but utilize more and more of Scilab's vector power with increasing function index *N*.

Example 5-1. Variants of a matrix mirror function

```
function b = mirror1(a, dir)
// mirror matrix a along its
// rows, dir = 'r' (horizontal)
// or along its columns, dir = 'c' (vertical)

[rows, cols] = size(a)
select dir
case 'r' then
    for j = 1 : cols
        for i = 1 : rows
            b(i, j) = a(rows - i + 1, j)
        end
    end
case 'c' then
    for j = 1 : cols
        for i = 1 : rows
            b(i, j) = a(i, cols - j + 1)
        end
    end
else
    error("dir must be "r" or "c"")
end
```



```
function b = mirror2(a, dir)
// same as mirror 1

[rows, cols] = size(a)
b = []
select dir
case 'r' then
    for i = rows : -1 : 1
        b = [b; a(i, :)]
    end
case 'c' then
    for i = cols : -1 : 1
        b = [b, a(:, i)]
    end
else
    error("dir must be "r" or "c"")
end
```

```
function b = mirror3(a, dir)
// same as mirror 1

[rows, cols] = size(a)
select dir
case 'r' then
    i = rows : -1 : 1
    b = a(i, :)
case 'c' then
    i = cols : -1 : 1
    b = a(:, i)
else
    error("dir must be "r" or "c"")
end
```

```
function b = mirror4(a, dir)
// same as mirror 1

select dir
case 'r' then
    b = a($:-1:1, :)
case 'c' then
    b = a(:, $:-1:1)
else
    error("dir must be "r" or "c"");
```

```
end
```

Besides the performance issue discussed here the functions in Example 5-1 demonstrate how much expressiveness Scilab has got. The solutions look quite different, though they give the same results. The benchmark results of all functions are plotted in Figure 5-1, and an extensive discussion is found in Section 5.2.1. In brief the functions get faster from top to bottom, function `mirror1` is the slowest, `mirror4` the fastest.

5.1.2.1. `$`-Constant

The last of the examples, `mirror4`, introduces a new symbol, the “highest index”, `$` along a given direction. The dollar sign is *only* defined in the index expression of a matrix. As `1` always is the lowest (or first) index, `$` always is the highest (or last). The dollar represents a constant, but this constant varies across the expression! More precisely it varies with each matrix dimension. Let us make things clear by stating an example.

```
->m = [ 11 12 13; 21 22 23 ];

->m(2, $)
 ans  =
    23.

->m($, $)
 ans  =
    23.

->m(:, $/2 + 1)
 ans  =
!   12. !
!   22. !
```

5.1.2.2. Flattened Matrix Representation

The `$` sign leads us to the flattened or vector-like representation of a matrix, if we rewrite the third line of the above example to

```
->m(1:$) 1
 ans  =
!   11. !
!   21. !
!   12. !
```



```
!   22. !
!   13. !
!   23. !
```

The expression `u = v(:)` is reshape operation, assigning to `u` the column-representation of `v`. For general reshaping of matrices, see the `matrix` function in Section 5.1.3.3.8.

> **Tip:** Given the vector `v`, the expression `v = v(:)` is a very convenient idiom in a function to force `v` into column (i.e. 1-times-*N*) form.

In general a *n*x*m* matrix *mat* can be accessed in three ways:

- as a unit by saying *mat*,
- by referencing its elements according to their row and column with *mat*(*i*, *j*), or
- via indexing into the flattened form *mat*(*i*).

The following equivalence holds: mat(i, j) == mat(i + (j - 1)*n). Scilab follows Fortran in its way to store matrices in column-major form. See also the discussion of the function `matrix` in Section 5.1.3.3.

5.1.3. Built-In Vector-/Matrix-Functions

Scilab provides many built-in functions that work on vectors or matrices. Knowing what functions are available is important to avoid coding the same functionality with slow iterative expressions.

For further information about contemporary techniques of processing matrices with computers, the classical work “Matrix Computations” [golub:1996] is recommended.

5.1.3.1. Vector Generation

There are two built-in functions and one operator to generate a row-vector of numbers.

5.1.3.1.1. Operator “`:`”

This syntax of the colon operator is

initial [: *increment*] : *final*

with a default *increment* of `+1`. To produce the equivalent piece of Scilab code, we write

```
x = initial
v = [ x ]
while x <= final - increment
    x = x + increment
    v = [v, x]
end
```

where `v` is the result. Note that the last element of the result always will be smaller or equal to the value *`final`*.

5.1.3.1.2. `linspace`

The syntax of `linspace` is

```
linspace(initial, final [, length])
```

using a default of `100` for *`length`*. `linspace` returns a row-vector with *`length`* entries, which divide the interval (*`initial`*, *`final`*) in equal-length sub-intervals. Both endpoints, i.e. *`initial`* and *`final`* are always included.

5.1.3.1.3. `logspace`

`logspace` works much like `linspace`, and the following relation holds

```
logspace(initial, final) == 10^linspace(initial, final)
```

5.1.3.2. Whole Matrix Construction

All of the functions shown in this section are capable to produce arbitrary matrices including the boundary cases of row- and column-vectors.

5.1.3.2.1. `zeros`

As the name suggests this function produces a matrix filled with zeros. The two possible instantiations are with two scalar arguments

```
n = 2
m = 5
mat = zeros(n, m)
```

or with one matrix argument


```
mat1 = [ 4 2; ..
         4 5; ..
         3 5 ]
mat2 = zeros(mat1)
```

The first form produces the `n` times `m` matrix `mat` made up from zeros, whereas the second builds the matrix `mat2` which has the same shape as `mat1`, and is also consisting of zeros.

Single scalar argument to `zeros`

In the case of a single scalar argument `zeros` returns a 1-times-1 matrix, the sole element being a zero.

Furthermore, note that

```
zeros()
```

is not allowed.

5.1.3.2.2. `ones`

The command is functionally equivalent to `zeros`. Instead of returning a matrix filled with `0.0` as `zeros` does, `ones` returns a matrix filled with `1.0`. The only difference is a third form which is permitted for `ones`, and that is calling the function without any arguments:

```
->ones()
 ans  =
    1.
```

5.1.3.2.3. `eye`

The `eye` function produces a generalized identity matrix, i.e. a matrix with all elements a(i, j) == 0.0 for i ~= j, and 1.0 for i == j. This command is functionally equivalent to `zeros`. The only extension is the usage without any argument, where the result automatically takes over the dimensions of the matrix in the subexpression it is used.

```
->a=[2 3 4 3; 4 2 6 7; 8 2 7 4]
 a  =
!   2.    3.    4.    3. !
!   4.    2.    6.    7. !
!   8.    2.    7.    4. !
```

```
->a - 2*eye()
 ans  =
!   0.    3.    4.    3. !
!   4.    0.    6.    7. !
!   8.    2.    5.    4. !
```

5.1.3.2.4. diag

Function `diag` has two different working modes depending on the shape of its argument. Given a vector *v* it constructs a diagonal matrix *mat* from the vector, with *v* being *mat*'s main diagonal, i.e. *mat*(i, i) = *v*(i) for all *v*(i). Given an arbitrary matrix *mat*, `diag` extracts the diagonal as a column-vector.

```
->diag(2:2:8)
 ans  =
!   2.    0.    0.    0. !
!   0.    4.    0.    0. !
!   0.    0.    6.    0. !
!   0.    0.    0.    8. !

->m = [2, 3, 8; 7, 6, -6; 0, -5, -8]
 m  =
!   2.    3.    8. !
!   7.    6.  - 6. !
!   0.  - 5.  - 8. !

->diag(m)
 ans  =
!   2. !
!   6. !
! - 8. !
```

The 2-argument form of the `diag` function

```
diag(v, k)
```

constructs a matrix that has its diagonal `k` positions away from the main diagonal, the diagonal being made up from `v` again. Therefore, `diag(v)` is the special case of `diag(v, 0)`. A positive `k` denotes diagonals above, a negative `k` diagonals below the main diagonal. As for the 1-argument form, extraction of the `k`th super-diagonal (positive `k`, or subdiagonal (negative `k`) is also implemented.

```
->diag([1 1 1 1]) + diag([2 2 2], 1) + diag([-2 -2 -2], -1)
 ans  =
!   1.    2.    0.    0. !
! - 2.    1.    2.    0. !
```


Chapter 5. Performance

```
!   0.  - 2.    1.    2. !
!   0.    0.  - 2.    1. !

->diag(m, -1) // using the same m as above
 ans  =
!   7. !
! - 5. !
```

> **Tip:** Nesting two calls to `diag` is the building block for an interesting idiom to test whether a matrix `m` is a diagonal matrix.
>
> ```
> and( abs(diag(diag(m)) - m) <= %eps * abs(m) )
> ```
>
> The inner call to `diag` extracts `m`'s main diagonal, the outer call taking this column-vector and construction a matrix out of it. The rest of the code simple checks the relative error.

5.1.3.2.5. rand

The `rand` function generates pseudo-random scalars and matrices. Again the function shares its two fundamental forms with `zeros`. Moreover, the distribution of the numbers can be chosen from `'uniform'` which is the default, and `'normal'`. The generator's seed is set and queried with

```
rand('seed', new_seed)
```

and

```
current_seed = rand('seed')
```

5.1.3.3. Functions Operating on a Matrix as a Whole

Although the section title might imply that the following functions apply to matrices only, Scilab's understanding allows for vectors anywhere a matrix is accepted.

5.1.3.3.1. find

In our opinion one of the most useful functions in the group of whole matrix functions is `find`. It takes a boolean expression of matrices, i.e. an expression which evaluates to a boolean matrix, as argument and in form

```
index = find(expr)
```

returns the indices of the array elements that evaluate to true, i.e. `%t` in a vector. See also Section 5.1.2.2.

In the form

```
[rowidx, colidx] = find(expr)
```

it returns the row- and column-index vectors separately. Here is a complete example:

```
->a = [ 1 -4  3; 6  2 10 ]
 a  =
!   1.   - 4.    3.  !
!   6.     2.    10. !

->index = find( a < 5 )
 index  =
!   1.    3.    4.    5. !

->a(index)
 ans  =
!   1. !
! - 4. !
!   2. !
!   3. !

->[rowidx, colidx] = find( a < 5 )
 colidx  =
!   1.    2.    2.    3. !
 rowidx  =
!   1.    1.    2.    1. !
```

The expressions *expr* can be arbitrarily complex. They are not limited to a single matrix at all.

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

->a < 5
 ans  =
! T T T !
! F T F !

->abs(b) >= 4
 ans  =
```



```
! F F F !
! T T T !

->a < 5 & abs(b) >= 4
 ans  =
! F F F !
! F T F !

->find( a < 5 & abs(b) >= 4 )
 ans  =
    4.
```

Last but not least `find` is perfectly OK on the left-hand side of an assignment. So, replacing all odd elements in `a` with `0` simply is

```
->a( find(modulo(a, 2) == 1) ) = 0
 a  =
!   0.   - 4.    0.  !
!   6.     2.    10. !
```

To get the number of elements that match a criterion, just apply `size(`*`idxvec`*`, '*')` to the index vector *`idxvec`* of the `find` operation.

5.1.3.3.2. `max`, `min`

Searching the smallest or the largest entry in a matrix are so common that Scilab has separate functions for these tasks. We discuss `max` only as `min` behaves similarly.

To get the largest value saying

```
max_val = max(a)
```

is enough. The alternate form

```
->[max_val, index] = max(a)
 index  =
!   2.    3. !
 max_val  =
    10.
```

returns the position of the maximum element, too. The form of the index vector is the same as for `size`, i.e. `[`*`row-index`*`, `*`column-index`*`]`. Speaking of `size`, `max` has the forms `max(`*`mat`*`, 'r')`, and `max(`*`mat`*`, 'c')`, too.

```
->[max_val, rowidx] = max(b, 'r')
```

```
 rowidx  =
!   2.    2.    2. !
 max_val  =
!   4.    5.    6. !

->[max_val, colidx] = max(b, 'c')
 colidx  =
!   3. !
!   3. !
 max_val  =
!   3. !
!   6. !
```

These forms return the maximum values of each row or column along with the respective indices of the elements' rows or columns.

The third way of using `max` is with more than one matrix or scalar as arguments. All the matrices must be compatible, scalars are expanded to full matrix size, like `scalmat = scal * ones(mat)`. The return matrix holds the largest elements from all argument matrices.

```
->max(a, b, 3)
 ans  =
!   3.    3.    3.  !
!   6.    5.    10. !
```

5.1.3.3.3. `and`, `or`

Both, `and` and `or` borrow their syntax from the `size` function: without a second argument, or a star, "`*`", as second argument the function is applied to the argument as a whole. A `1` or a "`r`" applies the function seperately to each row, yielding a row-vector as result. Accordingly a `2` or a "`c`" applies the function seperately to each column, yielding a column-vector as result.

The function `and` returns true if all components of the argument are true. Therefore it is related to Fortran-9x's `all` function. Similarly function `or` returns true if any component of its argument is true, mimicking Fortran-9x's `any` function.

One of the fastest ways of testing whether a vector (or matrix) *v* contains any non-zero element uses `or`: `or(v)`. As demonstrated with the `find` function, the arguments to `and` and `or` can take arbitrarily complex boolean expressions. If we like to test whether all components of the vector `v = [1.0 0.95 1.02]` are within 10% of the value 1, we do not need a loop: `and( abs(v - 1.0) < 0.1 )`.

5.1.3.3.4. Operator "`&`", Operator "`|`"

The operators "`&`", and "`|`" perform a component wise logical 'and', or logical 'or' operation. See also Section 4.1.3. The arguments to either operator can be scalars or matrices.

5.1.3.3.5. `sum`, `cumsum`, `prod`, `cumprod`

These are the numeric cousins of the boolean function pair `or` and `and`. Their syntax is identical. The "cum" functions work cumulatively, returning a vector (matrices are processing in their flattened representation).

A fast factorial function?

```
function f = fact(n)

if n < 0 then
    error("fact: domain")
end

if n == 0 then
    f = 1
else
    f = prod(1 : n)
end
```

$1000 at 4.5% over 7 years?

```
->1000.0 * cumprod( (1.0 + 0.045) * ones(7, 1) )
 ans  =
!   1045.      !
!   1092.025   !
!   1141.1661  !
!   1192.5186  !
!   1246.1819  !
!   1302.2601  !
!   1360.8618  !
```

though `1000.0 * (1.0 + 0.045)^(1:7)'` produces the same result and requires less keystrokes.

5.1.3.3.6. `gsort`

> **Warning**
>
> Do not use `sort`! It is buggy in that it sometimes does not return a permutation of the input data. Use `gsort` instead of `sort`.

The `gsort` function is a versatile sorting function for vectors and matrices of real numbers or strings. It sorts into increasing order or decreasing (default!) order, sorts a matrix's rows or columns separately, and can sort the rows or columns lexicographically. The output of `gsort` not only is the sorted matrix *`mat_sorted`* but also the permutation vector *`permutation`* that generates the sorted matrix from the input matrix. The synopsis is `[mat_sorted, permutation] = gsort(mat_input, mode, direction)`, where *`mode`* can have the values shown in Table 5-2, and *`direction`* the values displayed in Table 5-3.

Table 5-2. Mode Specifiers for `gsort`

Specifier	Action	Note
`'g'`	sort flattened matrix	default
`'r'`	column-by-column	
`'c'`	row-by-row	
`'lr'`	rows lexicographically	not `'rl'`!
`'lc'`	columns lexicographically	not `'cl'`!

Table 5-3. Direction Specifiers for `gsort`

Specifier	Action	Note
`'i'`	increasing order or upgrade	
`'d'`	decreasing order or downgrade	default

Let us look at some simple examples. We use a numeric matrix in the example, but a string matrix would do as well.

```
->mat1 = [11 12; 21 22; 31 32]
 mat1  =
!   11.    12. !
!   21.    22. !
!   31.    32. !

->gsort(mat1)
```



```
 ans  =
!   32.    21. !
!   31.    12. !
!   22.    11. !

->gsort(mat1, 'r')
 ans  =
!   31.    32. !
!   21.    22. !
!   11.    12. !

->gsort(mat1, 'c')
 ans  =
!   12.    11. !
!   22.    21. !
!   32.    31. !
```

Applied without parameters `gsort` sorts the flattened (see also Section 5.1.2.2) version, here: `mat(:)`, of its argument into decreasing order. The `'r'`- or `'c'`-options tell `gsort` to sort each column or row seperately.

> **Note:** `'r'` means column wise, and `'c'` means row wise!

The next example points out the difference between simple row- or column-sorting and lexicographical sorting of columns or rows.

```
->mat2 = [6 72 23; 56 19 23; 66 54 21]
 mat2  =
!   6.     72.    23. !
!   56.    19.    23. !
!   66.    54.    21. !

->gsort(mat2, 'r')  // col-by-col
 ans  =
!   66.    72.    23. !
!   56.    54.    23. !
!   6.     19.    21. !

->gsort(mat2, 'lc')  // col lexico
 ans  =
!   72.    23.    6.  !
!   19.    23.    56. !
!   54.    21.    66. !
```

```
->gsort(mat2, 'c')  // row-by-row
 ans  =
!   72.    23.    6.  !
!   56.    23.    19. !
!   66.    54.    21. !

->gsort(mat2, 'lr')  // row lexico
 ans  =
!   66.    54.    21. !
!   56.    19.    23. !
!   6.     72.    23. !
```

Now what is the exact difference between row-by-row sorting and lexicographic row sorting? After row-by-row sorting (in decreasing order) of an *m*-times-*n* matrix *a* the following relation holds: *a*(i, j) >= *a*(i, j + 1) for 1 <= i <= *m* and 1 <= j <= *n* - 1. In other words each row is sorted separately by interchanging its columns. After a lexicographic sort the relation between the rows is: *a*(i, :) >= *a*(i + 1, :) for 1 <= i <= *m* - 1. This time whole rows are compared to each other. Analogous facts hold for column sorting.

In environments not as rich as Scilab `gsort` might be the heart of user-written `min`, `max`, and `median` functions. All three are predefined in Scilab.

5.1.3.3.7. `size`

The `size` function handles all shape inquiries. It comes in four different guises. Assuming that *mat* is a scalar or matrix, `size` can be used as all-info-at-once function as in

```
[rows, cols] = size(mat)
```

as row-only, or column-only function

```
rows = size(mat, 'r')
cols = size(mat, 'c')
```

and finally as totaling function

```
elements = size(mat, '*')
```

5.1.3.3.8. `matrix`

A (hyper-)matrix can be reshaped with the `matrix` command. To keep things simple we demonstrate `matrix` with a 6x2-matrix.


```
->a = [1:6; 7:12]
 a  =
!   1.     2.     3.     4.      5.      6.  !
!   7.     8.     9.     10.     11.     12. !

->matrix(a, 3, 4)
 ans  =
!   1.     8.     4.      11. !
!   7.     3.     10.     6.  !
!   2.     9.     5.      12. !

->matrix(a, 4, 3)
 ans  =
!   1.     3.      5.  !
!   7.     9.      11. !
!   2.     4.      6.  !
!   8.     10.     12. !
```

In contrary to the Fortran-9x function `RESHAPE`, `matrix` neither allows padding, nor truncation of the reshaped matrix. Put another way, for a *m* times *n* matrix *a* the reshaped dimensions *p*, and *q* must obey *m * n = p * q*.

`matrix` works by columnwise “filling” the contents of the original matrix *a* into an empty template of a *p* times *q* matrix. (See also Section 5.1.2.2.) If this a too hard to imagine, the second way to think of it is imagining *a* as a column vector of dimensions (*m * n*) times 1 that is broken down column by column into a *p* times *q* matrix. In fact this is not pure imagination as in many situations there is the identity *a(i, j) == a(i + n*(j - 1))* holds.

```
->a(2,4)
 ans  =
    10.

->a(8)
 ans  =
    10.
```

Moreover, the usual vector subscripting can be used to a matrix.

```
->a(:)
 ans  =
!   1.  !
!   7.  !
!   2.  !
!   8.  !
!   3.  !
```

```
!   9.  !
!   4.  !
!   10. !
!   5.  !
!   11. !
!   6.  !
!   12. !
```

5.1.4. Evaluation Of Polynomials

Once upon a time there was a little Scilab newbie who coded an interface to the `optim` routine to make polynomial approximations easier. On the way an evaluation function for polynomials had to be written. The author was very proud of herself because she knew the Right Thing(tm) to do in this case namely the Horner algorithm. Actually she immediately came up with two implementations.

Example 5-2. Naive functions to evaluate a polynomial

```
function yv = peval1(cv, xv)
// Evaluate polynomial given by the vector its
// coefficients cv in ascending order, i.e.
// cv = [p q r]  ->  p + q*x + r*x^2 at all
// points listed in vector xv and return the
// resulting vector.

yv = cv(1) * ones(xv)
px = xv
for c = cv(2 : $)
    yv = yv + c * px
    px = px .* xv
end


function yv = peval2(cv, xv)
// same as peval1

yv = cv($);
for i = length(cv)-1 : -1 : 1
    yv = yv .* xv + cv(i)
end
```


So what is wrong with that? This code looks OK and it does the job. But from the performance viewpoint it is not optimal! The fact that Scilab offers a separate type for polynomials has been ignored. Even if we are forced to supply an interface with the coefficients stored in vectors the built-in function `freq` is preferable.

Example 5-3. Less naive functions to evaluate a polynomial

```
function yv = peval3(cv, xv)
// same as peval1, using horner()

p = poly(cv, 't', 'coeff')
yv = horner(p, xv)


function yv = peval4(cv, xv)
// same as peval1, using freq()
// The return value yv _always_ is a row-vector.

p = poly(cv, 't', 'coeff')
unity = poly(1, 't', 'coeff')
yv = freq(p, unity, xv)
```

Table 5-4 shows the speed ratios (each line is normalized separately) for a polynomial of degree 4 that we got on a P5/166 Linux system.

Table 5-4. Performance comparison of different polynomial evaluation routines

evaluations	peval1	peval2	peval3	peval4
5	3.5	4.2	1	7.0
1000	1.4	2.5	1	2.5

If we now decide to change our interface to take Scilab's built-in polynomial type the evaluation with `freq` can again be accelerated by a factor of more than 3.

5.2. Extending Scilab

The brute force way of getting a better performance is rewriting an existing Scilab script in a low-level language as C, Fortran, or even assembler. This option should be chosen with care, because the rapid prototyping facilities of Scilab are lost. On the other hand if the interface of the function has settled, its performance is known to be crucial and it is of use in future projects then the translation into compiled code could be be worth the time and the grief.

In the first part of this section we compare different ways of integrating an external function into Scilab. We focus on the ease of integration versus the runtime overhead introduced. The second part deals with writing the low-level functions themselves, especially their interfaces.

5.2.1. Comparison Of The Link Overhead

We revive our matrix mirroring example from Section 5.1.2.

Our Fortran-77 version looks like this:

```
      subroutine mir(n, m, a, dir, b)
*
*     Mirror n*m-matrix a along direction prescribed
*     by dir.  If dir == 'r' then mirror along the
*     rows, i.e. horizontally.  Any other value for
*     dir mirrors along the columns, i.e. vertically.
*     The mirrored matrix is returned in b.
*
      implicit none

*     ARGUMENTS
      integer n, m
      double precision a(n, m)
      character dir*(*)
      double precision b(n, m)

*     LOCAL VARIABLES
      integer i

*     TEXT
      if (dir(1:1) .eq. 'c') then
          do 100, i = 1, m
              call dcopy(n, a(1, m+1-i), 1, b(1, i), 1)
 100      continue
      else
          do 200, i = 1, n
              call dcopy(m, a(n+1-i, 1), n, b(i, 1), n)
 200      continue
      end if

      end
```


The `dcopy`(*n*, *x*, *incx*, *y*, *incy*,) is from BLAS level 1, and copies *n* `double precision` elements from vector *x* in increments of *incx* to *y*, where it uses increments of *incy*.

The only thing missing is the glue code between Scilab and `mir`.

```
function b = mirf(a, dir)
// interface function for 'mir.f'
// Behavior is the same as mirror()

[n, m] = size(a)
b = zeros(n, m)

if dir == 'r' | dir == 'c' then
    b = fort('mir', ..
             n, 1, 'i', m, 2, 'i', a, 3, 'd', dir, 4, 'c', ..
             'out', ..
             [n, m], 5, 'd')
else
    error('dir must be "r" or "c"')
end
```

OK, let's lock-and-load. We are ready to rock!

```
link('mir.o', 'mir')
getf('mirf.sci')
```

The fast alternative to using `fort`, which dynamically creates an interface to a C or Fortran function is using **intersci**, which which creates an interface suitable for static loading.

intersci can create the Fortran glue code for a C or Fortran function to make it callable form the Scilab interpreter. The glue code is compiled (with a Fortran compiler) and linked to Scilab. **intersci** is described very well in the SCI`/doc/Intro.ps`. Anyhow, Example 5-4 shows the description ("`.desc`") file for our current example. Finally it will supply us with a Scilab function called `mirai(a, dir)`.

Example 5-4. Sample interface description ("`.desc`")

```
mirai   a       dir
a       matrix  n       m
dir     string  1
b       matrix  n       m

mir     n       m       a       dir     b
n       integer
m       integer
a       double
```

```
dir      char
b        double

out      sequence        b
*
```

We do not want to go into detail here, but a `desc`-file has three parts separated by blank lines: The description of the Scilab-level function's signature (here: `mirai`), the same for the low-level function (here: `mir`), and finally the results' structure. The signatures resemble Fortran or K&R-style C function definitions with the parenthesis missing. The process of passing a `desc`-file through **intersci**, compiling the low-level routine and the glue code can be automated. Example 5-5, a snippet of our `Makefile.intersci` shows the relevant rules.

Example 5-5. `Makefile` for static Scilab interfaces via intersci

```
ifdef SCI
SCIDIR := $(SCI)
else
SCIDIR := /site/X11R6/src/scilab
endif

%.f.pre: %.desc
        $(SCIDIR)/bin/intersci $*
        mv $*.f $*.f.pre

%.f: %.f.pre
        perl -pe ’s#SCIDIR#$(SCIDIR)#’ $< > $@

%.o: %.f
        $(FC) $(FFLAGS) -c $<
```

Running the automatically generated Fortran code through a filter (here: **perl**) is necessary to fix the lines `include ’SCIDIR/routines/stack.h’`. After everything is compiled a single Scilab command makes the new routine available to the user.

```
addinter([’mirai.o’, ’mir.o’],  // object files
         ’mirai’,               // name of interface routine
         ’mirai’)               // name of new Scilab function
```

The first argument which almost always is a vector of strings tells Scilab the names of the object files to load. One of them is the interface code made by **intersci**. The rest are the user routines. The second

argument specifies name of entry point into the interface routine. The third parameter is the name the new Scilab function will carry.

> **Entry point of interface function**
>
> `addinter`'s second argument must be the name of the *interface routine*, i.e. the one generated by **intersci**. Using the low-level function's entry point here causes Scilab to barf (of course).

Why do we go through that tedious process? After all we are in the performance section, so what we want is speed, high speed, or even better the ultimate speed. Now we can compare all the variants as is done in Figure 5-1.

Figure 5-1. Benchmark results for the `mirror` functions

Performance comparison of `mirror[1-4]`, and `mirai` on a P5/166 Linux box.

Performance comparison of `mirror[1-4]`, and `mirai` on a 2-way PIII/550 Linux box.

If we compare the performance of our three Scilab mirror routines `mirror1`, `mirror2`, and `mirror3` together with the two incarnations of the hard-coded routine `mirf`, and `mirai`, we reach at the following conclusions.

- Scilab code that makes heavy use of indexing, like `mirror1`, is extremely slow no matter what problem size. Thumbs down on that one.
- Well written i.e. index-free Scilab code, like `mirror4`, performs very well. This is especially true for large vectors or matrices.

- The overhead of the `fort`-call in `mirf` is high; it is hard to amortize for that. `fort` is only justified in situations where a significant amount of time is spent in the low-level user-routine. Usually this will be the case for large problem sizes. Of course the cross-over point has to be determined separately in each case.
- Nothing can beat a compiled function that is integrated with `addinter`. `mirai` surpasses all other implementations. For small problem sizes the little overhead in comparison to all the other functions gives this function a factor 10 advantage, though, as the problems size increases `mirai`'s lead is challenged by `mirror4`.

Conclusion: Never underestimate the power of the Emperor^H^H^H^H^H^H vectorized Scilab code.

5.2.2. Preparing And Compiling External Subroutines

In this section we will discuss the interfacing of C, C++, Fortran-77, Fortran-9x, or Ada routines with Scilab via `link` command. We restrict ourselves to the simple case of functions that expect exactly one double precision floating point parameter and return a double precision floating point result. Functions with that signature are required e.g. for the integration routine `intg`, or the root finder `fsolve`.

Before we dive into the language specific descriptions, let us point out the main features of Fortran we have be pay attention to when writing an interface in another language.

Function name mangling

A function named `FOO` (`foo`, or whatever capitalization is chosen) in the Fortran source can become a different symbol in the object file. This is compiler dependent. Most often an underscore "_" is prepended or appended. Sometimes the name is downcased, sometimes it is upcased.

Tip: The **nm**(1) command provides easy access to the symbols in an object file.

Call-by-reference

Fortran never passes the value of a parameter, but always a pointer to the parameter.

Arrays in column-major order

Arrays are stored so that their leftmost index varies fastest.

5.2.2.1. Fortran-77

Fortran-77 or, how do you want to ruin your day?

lvd

Extending Scilab with Fortran-77 is most straightforward. Scilab is writtin in that language, remember? A Fortran-77 source for function `fals` could look like this:

```
      double precision function fals(x)

      double precision x

      fals = sin(10.0d0 * x)

      end
```

After compilation (e.g. **`f77 -c fals.f`**) the compiled code can be linked to Scilab and called with the integration routine.

```
link('fals.o', 'fals');
[res, aerr, neval, info] = ..
    intals(0.0, 1.0, -0.5, -0.5, 'alg', 'fals')
```

5.2.2.2. Fortran-9x

Fortran-90? Don't worry, it can't get much worse.

cls

A bloated, but portable Fortran-90 source for a function could look like this:

```
function fsm(x)
    implicit none
    integer, parameter :: idp = kind(1.0d0)

    ! arguments/return value
    real(kind = idp), intent(in) :: x
    real(kind = idp) :: fsm

    ! text
    fsm = exp(x) / (1.0d0 + x*x)
end function fsm
```

After compilation (e.g. **f90 -c fsm.f90**) the compiled code can be linked to Scilab and called with an integration routine.

```
link('fsm.o', 'fsm');
[ires, ierr, neval] = intsm(0.0, 1.0, 'fsm')
```

5.2.2.3. (ANSI-) C

A simple C function meeting our signature requirements has e.g. this shape:

```
#include <math.h>
#include "machine.h"

double
C2F(fgen)(const double *x)
{
        if (*x > 0.0)
                return 1.0 / sqrt(*x);
        else
                return 0.0;
}
```

After compilation (e.g. **cc -I/site/X11R6/src/scilab/routines -c fgen.c**) the compiled code can be linked to Scilab and called with the integration routine.

```
link('fgen.o', 'fgen', 'c');
[ires, ierr, neval, info] = intgen(0.0, 1.0, 'fgen')
```

There are several ways to get the naming convention differences between Fortran and C right. We show three possible solutions for the case where C uses no decoration at all and Fortran appends one underscore.

```
/* (1) GNU C compiler */
double foo(const double *x) __attribute__((weak, alias ("foo_")));

/* (2) good preprocessor */
#define C2F(name) name##_

/* (3) old preprocessor ;-) */
#define ANOTHERC2F(name) name/**/_
```

None of the above three examples is portable. Therefore, it is prudent to include SCI`/routines/machine.h`, which is automatically generated during the Scilab configuration process

and thus knosw of the name mangling. Among a lot of other macros it supplies a C-to-Fortran name conversion macro called `C2F`.

5.2.2.4. C++

A C++ source for a function could look like this:

```
#include <math.h>

extern "C" {
        double C2F(fgk)(const double *x);
}

double
C2F(fgk)(const double *x)
{
        return 2.0 / (2.0 + sin(10.0 * M_PI * (*x)));
}
```

After compilation (e.g. **c++ -I/site/X11R6/src/scilab/routines -c fgen.c**) the compiled code can be linked to Scilab and called with the integration routine.

```
link('fgk.o', 'fgk', 'c');
[ires, ierr, neval, info] = ..
    intgk(0.0, 1.0, 'fgk', 0, %eps, '15-31')
```

See Section 5.2.2.3 for a discussion of the `C2F` macro.

Further problems arise if the C++ code depends on libraries that have not been linked with Scilab. In the following example `myfct_` is correctly declared, but requires `sqrt` indirectly through a call to `subfct`.

```
// linkcxx.cc
#include <complex>

extern "C" {
    void myfct_(const double *re, const double *im);
}

double_complex subfct(double_complex z);

void
myfct_(const double *re, const double *im)
{
    double_complex u(*re, *im);
```

```
    double_complex v(subfct(u));
    // do something with v
}


double_complex
subfct(double_complex z)
{
    return 1.0 + 0.5 * sqrt(z);
}
```

The problem when linking `myfct_` with Scilab is not the call to `subfct`, but the missing complex `sqrt` function. A listing of the object file's symbols shows the missing function among some functions the (this particular version of g++) compiler silently generates due to `inline` expansion.

```
lydia@orion:/home/lydia/tmp $ nm -C linkcxx.o
000000bd t Letext
00000000 ? __FRAME_BEGIN__
00000000 W complex<double> operator/<double>(complex<double> const &, double)
00000000 W complex<float> operator/<float>(complex<float> const &, float)
00000000 W complex<long double> operator/<long double>(complex<long dou-
ble> const &, long double)
0000008b T main
00000000 T myfct_
         U complex<double> sqrt<double>(complex<double> const &)
00000046 T subfct(complex<double>)
```

It is up to the programmer to supply *all* necessary libraries – in the correct order – when linking. For the previous example the following call would succeed (on a libc6 GNU/Linux system):

```
->link("linkcxx.o -lstdc++-2-libc6.1-1-2.9.0")
linking files linkcxx.o -lstdc++-2-libc6.1-1-
2.9.0  to create a shared executable
shared archive loaded
Link done
 ans  =

    0.
```

In the case that the compiler documentation lacks information abouth which library defines what symbol, the **nm**(1) command is the most useful tool to find out.

Additional Caveats

The inclusion of C++ modules into a project whose `main()` is not written in C++ call for some additional warnings. See also Section 5.3 for a caveat using compilation switches that break the ABI.

Runtime initialization

: When it comes to runtime initialization of his/her code, a C++-programmer depends on the linker as a junkie on his dealer. Either the compiler system does it – and does it right, or you have a very very hard time ahead of you. Sidenote: The GNU linker does the Right Thing(tm)!

`exceptions`

: In brief: Get them – *all*! If the C++ to be linked with Scilab is known to throw exceptions, all interfaced functions of which an exception possibly could escape have to be wrapped in C++-functions that catch these exceptions and translate them into error codes e.g. à la LAPACK. Otherwise Scilab is terminated with an `abort()` call.

5.2.2.5. Ada

For GNAT/Ada the package's interface part pulls in the Fortran interface definitions. Is the simplest case the mathematical functions are only instantiated with the type `Double_Precsion`. Ada requires to export every function's interface separately, as is clear from the following example.

```
with Interfaces.Fortran;
use Interfaces.Fortran;
with Ada.Numerics.Generic_Elementary_Functions;

package TestFun is
    package Fortran_Elementary_Functions is new
        Ada.Numerics.Generic_Elementary_Functions(Double_Precision);
    use Fortran_Elementary_Functions;

    function foo(x : Double_Precision) return Double_Precision;
    pragma Export(Fortran, foo);
    pragma Export_Function(Internal => foo,
                           External => "foo_",
                           Mechanism => Reference,
                           Result_Mechanism => Value);
end TestFun;
```

According to the interface specification the package body looks like this:

```
package body TestFun is
```

```
    function foo(x : Double_Precision) return Double_Precision is
    begin
        return exp(x) / (1.0 + x*x);
    end foo;
end TestFun;
```

The package is compiled as usual **gnatmake -O2 testfun.adb**.

Hint: Make sure that there is a GNAT runtime library `libgnat-3.12p.so`. Your version number may be different, but only the ending (“`so`”) is critical, as `libgnat-3.12p.so.1.7` will not make `dlopen`(3) happy. From now on everything is downhill, and the function can be `link`ed almost as usual.

```
link('testfun.o -L/site/gnat-3.12p/lib -lgnat-3.12p', 'foo')
```

Again, the path to your gnat-library and the version numbers can differ.

In the case of several functions in the package it is preferable to rely on the extended `dlopen`(3) mechanism, and link the package/library combo with remembering the id of the shared library.

```
adacode = link('testfun.o -L/site/gnat-3.12p/lib -lgnat-3.12p', 'foo')
```

Linking further functions from the library happens by referencing the number of the library.

```
link(adacode, 'bar')
```

This saves space (Scilab’s TRS) and time (to execute the `link`). Speaking about saving... Users with a loader e.g. GNU **ld**, capable of incremental linking (e.g. `-i`, `-r`, `-relocatable`) can of course link `testfun.o` with the (gnat-)library before linking everything to Scilab. To complete the example, here comes the command-line:

ld -i -o testfun-lib.o testfun.o -L/site/gnat-3.12p/lib -lgnat-3.12p

In Scilab the arguments to `link` then reduce to

```
link('testfun-lib.o', 'foo')
```

5.2.3. Pushing It Further

What? What are you doing in this section? Still not satisfied with your functions’ performance?—Sorry, but there are no conventional ways to get more out of Scilab. Tinkering with the interface routines is not worth the effort. Some completely new approach is necessary.

5.2.3.1. Scilab as Prototyping Environment

If a problem is too tough, Scilab still can serve as a rapid prototyping environment. One sister program of Scilab, namely *Tela* (http://www.geo.fmi.fi/prog/tela.html) has been written for exactly this purpose. Prototyping with an interpreted language is currently going through a big revival with C (and C++) developers discovering Python.

As whenever optimization is the final goal, an extensive test suite is the base for success. So one way to proceed could be to develop test routines and reference implementation completely in Scilab. The next step is rewriting the routines *still* in Scilab to match the signatures of for example BLAS/LAPACK routines as closely as possible. The test suite can remain untouched in this step. The final step is to migrate the Scilab code to Fortran, C, or whatever, while making extensive use of BLAS/LAPACK. Ideally the test suite remains under Scilab and can be used to exercise the new standalone code.

5.2.3.2. Scilab to Fortran-77 Compiler

FIXME: write it!

5.3. Building an Optimized Scilab

One relatively easy way to to increase Scilab’s performance is recompiling it with a good compiler and an optimized BLAS library[2]. See Section 7.3 to find out what optimized BLAS kernels are available, and where to get them.

Our experience only suffices to explain the compilation on ia32 GNU/Linux systems. Here, *gcc* (http://gcc.gnu.org) or *pgcc* (http://www.goof.com/pcg/) are the compilers of choice.

The following options are a good starting point for further exploration. They apply to compiling Fortran as well as C code.

`-march=`*`arch`*

: This option instructs gcc to generate code specifically for architecture *`arch`*. Among other things it sets `-mcpu=`*`arch`*. Furthermore, it forces `-malign-loops`, `-malign-jumps`, `-malign-functions`, and `-mpreferred-stack-boundary` to their optimum values for the selected architecture *without* braking the ABI. Therefore, it can be considered an optimization switch.

`-malign-double`

For systems with an original 1ntel P5 or above processor this option is an absolute *must*. It forces the aligment of 64 bit floating point numbers (also known as double, double precision, and IEEE754) to a 64 bit boundary. Though *it breaks the ABI*, the gain in speed due to avoiding the misalignment penalty on each memory access is tremendous, even on PPro (and derivative) systems with all write back caches enabled.

> **Warning**
>
> `-malign-double` breaks the ABI!
>
> Code using double compiled with [p]g++-2.95 and `-malign-double` is known to cause segmentation faults under some circumstances.

`-O2`

The workhorse optimization switch, `-O2`, activates a lot of optimizations. See node "Optimize Options" in gcc's info file, e.g. **info -f /usr/info/gcc.info.gz -n "Optimize Options"**

The optimizations toggled on by `-O2` are well tested and do not produce excessively long text.

`-funroll-all-loops`

This switch increases the text size by unrolling as many loops as possible, thereby speeding them up. YMMV.

`-fschedule-insns2`

Although the gcc info page states that this optimization is switched on by `-O2`, this might not be true for all versions of gcc floating around. The switch should be particularly helpful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

Notes

1. Remember that the colon operator returns a row-vector.
2. Simply linking with an optimized BLAS library generally is not enough. Patches (e.g. "fast-blas", and "big patch") to fix part of this problem exist. Check *Scilab Stuff* (http://www.lightlink.com/lydia/scilab.html).

Chapter 6. Scilab Core

Aerosmith video “Love In An Elevator”, “Pump” (1989).

Good morning Mister Tyler! Going down?

We are going down all the way right to the core, the core of Scilab. Though this is the most technical and most complex chapter, it is by no means true that writing a native Scilab function is unmanageable by for ordinary mortals. A strict programming discipline, patience, persistence, and a thorough knowledge of what makes up the stack-structures involved, let us overcome the difficulties.

6.1. Introduction To Pseudo-Ada

Instead of simply repeating the Fortran-77 statements that make up the Scilab stack, the API, etc., we introduce a new language that is better suited for this job: a pseudo Ada[1], pAda, which is much more expressive. The syntax follows Ada, and the pAda types are mapped onto Fortran-77 types as listed in Table 6-1. What might look like an artificial complication, the introduction of new types, actually is a major simplification (Three cheers for Ada!). First, the name of the type now makes clear exactly what it is used for. Second, distinct types designate distinct things, stuff that never should be mixed up. Third, the valid ranges of the sub-types are explicity mentioned.

Table 6-1. pAda to Fortran-77 type mappings

pAda	Fortran-77
`Integer`	`INTEGER`
`Float`	`DOUBLE PRECISION, REAL*8`
`Boolean`	`LOGICAL`
`Character`	`CHARACTER`
`type String is array (1..`*`N) of Character`*	`CHARACTER*`*`N`*
`type ComplexFlag is (RealVariable, ComplexVariable)`	`INTEGER = 0, 1`
`subtype Natural is Integer range 0..Integer’Last`	`INTEGER`
`type ParameterStackAddress is new Integer range 1..Integer’Last`	`INTEGER`
`type DataStackIndex is new Integer range 1..Integer’Last`	`INTEGER`

6.2. Internal Data Structure

FIXME: explain the parameter stack, data stack, etc.

6.2.1. Parameter Stack And Data Stack

FIXME: follow the docu in Internals

6.2.2. Storage of Complex Matrices

FIXME: explain separate storage of two DOUBLE PRECISION parts instead of one DOUBLE COMPLEX

6.3. Writing Native Scilab Functions

In the following two sections we shall treat the “anatomy” of native, i.e. low-level Scilab functions. This will confront us with all the gory details of the stack, the low-level API, and the calling conventions. Having the “Guide for Developers”, `Internals.ps` (see also Section 7.2) ready is a good idea. Where the developer guide is at the end of its wits, a study of the source code is appropriate, especially the file SCI`/routines/interf/stack1.f`

We start out discussing simple functions. Simple in the sense that they are self-contained and only take non-function parameters as their arguments. In the second part we shall consider functions that take other functions (either Scilab functions or externals) as arguments.

6.3.1. Simple Functions

A typical native Scilab function proceeds as follows:

1. Check the number of input and output parameters.

2. Get the “pointers” to actual input parameters; supply default values for optional parameters; issue warnings or errors as appropriate if too many or too few parameters are supplied.
3. Allocate space for temporary variables, “workspace(s)”, etc.
4. It might be necessary to translate the input variables which are in Scilab format into the appropriate format for the worker routine. This is happens for example if the worker routine uses Fortran-77’s `double complex` (or equivalently `complex*16`) variables. See Section 6.2.2 for details.
5. Perform the calculations or transformations that *really* make up the procedure.
6. As in Step 4, it might be necessary to transform the results, now from the worker routine’s format back into Scilab format.
7. If necessary, allocate space for the return value(s) on the Scilab stack, and copy result(s) to this space.

Now that the general outline is clear, and we are ready to dissect a simple function: `ortho`. It takes exactly one argument a, that is a real or complex m times n matrix. The single output parameter is a matrix of the same shape and type as is the input matrix. The duty of `ortho` is to transform the columns of the input matrix into orthonormal form; to achieve this we employ the following LAPACK functions:

```
type Complex is record
    Re, Im : Float'Base;
end record;

type FloatVector   is array (Positive range <>) of Float;
type ComplexVector is array (Positive range <>) of Complex;
type FloatMatrix   is array (1..Lda, Positive range <>) of Float;
type ComplexMatrix is array (1..Lda, Positive range <>) of Complex;

procedure dgeqrf
    (M     : in     Natural;        - number of rows of A
     N     : in     Natural;        - number of cols of A
     A     : in out FloatMatrix;    - M-by-N matrix
     Lda   : in     Natural;        - leading dimension of A
     Tau   :    out FloatVector;    - scalar factors of elem. refl.
     Work  :    out FloatVector;    - workspace
     Lwork : in     Integer;        - size of workspace Work
     Info  :    out Integer);       - error indicator

procedure dorgqr
    (M     : in     Natural;        - number of rows of A
     N     : in     Natural;        - number of cols of A
     K     : in     Natural;        - number of elem. refl.
     A     : in out FloatMatrix;    - M-by-N matrix
     Lda   : in     Natural;        - leading dimension of A
```



```
     Tau   :    out FloatVector;    - scalar factors of elem. refl.
     Work  :    out FloatVector;    - workspace
     Lwork : in     Integer;        - size of workspace Work
     Info  :    out Integer);       - error indicator

procedure zgeqrf
    (M     : in     Natural;        - number of rows of A
     N     : in     Natural;        - number of cols of A
     A     : in out ComplexMatrix;  - M-by-N matrix
     Lda   : in     Natural;        - leading dimension of A
     Tau   :    out ComplexVector;  - scalar factors of elem. refl.
     Work  :    out ComplexVector;  - workspace
     Lwork : in     Integer;        - size of workspace Work
     Info  :    out Integer);       - error indicator

procedure zungqr
    (M     : in     Natural;        - number of rows of A
     N     : in     Natural;        - number of cols of A
     K     : in     Natural;        - number of elem. refl.
     A     : in out ComplexMatrix;  - M-by-N matrix
     Lda   : in     Natural;        - leading dimension of A
     Tau   :    out ComplexVector;  - scalar factors of elem. refl.
     Work  :    out ComplexVector;  - workspace
     Lwork : in     Integer;        - size of workspace Work
     Info  :    out Integer);       - error indicator

procedure dcopy
    (N     : in     Natural;        - number of elements to copy
     X     : in     FloatVector;    - input vector
     IncX  : in     Integer;        - input stride
     Y     :    out FloatVector;    - output vector
     IncY  : in     Integer);       - output stride
```

The `dgeqrf`- and `zgeqrf`-functions compute a QR-factorization of a real or complex m-by-n matrix a, while the `dorgqr`-, and `zungqr`-functions generate an m-by-n real or complex matrix q with orthonormal columns, relying on the QR-factorization of `dgeqrf` or `zgeqrf`. Function `dcopy` copies N elements (of type Float) of the vector X in increments of *IncX* to the vector Y using increments of *IncY* on that side. For a detailed description please consult the LAPACK User Guide, or the appropriate manual pages.

Example 6-1 is one of the longest examples in the running text, but don’t be scared as we will explain line-by-line and variable-by-variable what is where and why.

Example 6-1. Simple native Scilab function

```
      subroutine ortho
```
– *Native functions are parameterless*

```
      implicit none
```
– *Switch into weeny mode :-)*

```
*     CONSTANTS
      integer realtype
      parameter (realtype = 0)
```
– *See Table 6-1 for type association*

```
*     LOCAL VARIABLES
      character*6 fname
```
– *name of the routine as string*

```
      logical checklhs, checkrhs, cremat, getmat
```
– *Scilab API functions*

```
      integer topk
      integer n, m, mattyp
      integer tausz, worksz, info
      integer areadr, aimadr, badr, tauadr
      integer wrkadr, rreadr, rimadr, dumadr

*     EXTERNAL FUNCTIONS/SUBROUTINES
      external checklhs, checkrhs, cremat, getmat
```
– *Scilab API functions*

```
      external error

      external dcopy, dgeqrf, dorgqr, zgeqrf, zungqr
```
– *LAPACK/BLAS worker subroutines*

```
*     HEADER
      include '/site/X11R6/src/scilab/routines/stack.h'
```
– *Scilab API header*

```
*     TEXT
      fname = 'ortho'
```
– *Function name (for error messages)*

```
      topk = top
```
– `top` *is defined in* `stack.h`

```
      rhs = max(0, rhs)

      if (.not. checkrhs(fname, 1, 1)) return      ❶
      if (.not. checklhs(fname, 1, 1)) return
```



```
*     fetch input parameters                         ❷
      if (.not. getmat(fname, topk, top - rhs + 1,
     $        mattyp, m, n, areadr, aimadr)) return

      if (n * m .eq. 0) return
```
– *Quick return on empty matrix*

```
      tausz = min(m, n)
```
– *Prescribed by man-page*

```
      worksz = max(1, n)
```
– *... same here*

```
      if (mattyp .eq. realtype) then
*     real case

*     allocate temporary variables; all are real  ❸
          if (.not. cremat(fname, top + 1, realtype, tausz, 1,
     $            tauadr, dumadr)) return
          if (.not. cremat(fname, top + 2, realtype, worksz, 1,
     $            wrkadr, dumadr)) return
          if (.not. cremat(fname, top + 3, realtype, m, n,
     $            badr, dumadr)) return

*     prepare worker routines' input parameters   ❹
          call dcopy(n * m, stk(areadr), 1, stk(badr), 1)

*     call worker routines                          ❺
          call dgeqrf(m, n, stk(badr), m, stk(tauadr),
     $            stk(wrkadr), worksz, info)
          if (info .ne. 0) then
```
– *Any error is considered fatal*

```
              buf = fname // ' dgeqrf failed'
              call error(999)
              return
          endif

          call dorgqr(m, n, tausz, stk(badr), m, stk(tauadr),
     $            stk(wrkadr), worksz, info)
          if (info .ne. 0) then
```
– *Any error is considered fatal*

```
              buf = fname // ' dorgqr failed'
              call error(999)
              return
          endif

      else
```

```
*     complex case; mattyp != realtype

*     allocate temporary variables,
*     use two REAL*8 for one COMPLEX*16           ❻
          if (.not. cremat(fname, top + 1, realtype, 2 * tausz, 1,
     $            tauadr, dumadr)) return
          if (.not. cremat(fname, top + 2, realtype, 2 * worksz, 1,
     $            wrkadr, dumadr)) return
          if (.not. cremat(fname, top + 3, realtype, 2 * m, 2 * n,
     $            badr, dumadr)) return

*     prepare worker routines' input parameters, joining
*     two REAL*8 arrays into one COMPLEX*16 array ❼
          call dcopy(n * m, stk(areadr), 1, stk(badr),     2)
          call dcopy(n * m, stk(aimadr), 1, stk(badr + 1), 2)

*     call worker routines                        ❽
          call zgeqrf(m, n, stk(badr), m, stk(tauadr),
     $            stk(wrkadr), worksz, info)
          if (info .ne. 0) then                             –
Any error is considered fatal
              buf = fname // ' zgeqrf failed'
              call error(999)
              return
          endif

          call zungqr(m, n, tausz, stk(badr), m, stk(tauadr),
     $            stk(wrkadr), worksz, info)
          if (info .ne. 0) then                             –
Any error is considered fatal
              buf = fname // ' zorgqr failed'
              call error(999)
              return
          endif

      endif


*     get ready to exit
      if (lhs .ge. 1) then                        ❾
          if (.not. cremat(fname, top, mattyp, m, n,
     $            rreadr, rimadr)) return
          if (mattyp .eq. realtype) then          (10)
              call dcopy(m * n, stk(badr), 1, stk(rreadr), 1)
          else
```



```
              call dcopy(m * n, stk(badr),     2, stk(rreadr), 1)
              call dcopy(m * n, stk(badr + 1), 2, stk(rimadr), 1)
          endif
      endif

      end
```

❶ Check the number of input and output parameters. The task is easy as we receive one and return exactly one parameter. This line and the next correspond to Step 1.

❷ Get the addresses – as mentioned in Step 2 – of the real, and imaginary part of the matrix passed as (only) parameter to `ortho`. Note that `getmat` will return `False` if the parameter at the given parameter stack position is not a matrix of numbers.

Function `getmat` is called with the second parameter, `topk`, holding the value of the parameter stack pointer when the control flow entered `ortho`. This as well as the function name passed in `fname` is necessary for the cleanup and messaging in case of an error.

The only parameter we use sits on top of the parameter stack for `top - rhs + 1` equals `top` in our case.

On successful return `getmat` not only sets the data stack addresses `areadr`, and `aimadr` of the real and imaginary parts, but also tells us via `mattyp` whether the matrix is real complex, and via `m`, and `n` how large the matrix is.

The following lines directly depend on the sizes passed back form the core interface, calculating the necessary space for two scratch arrays.

❸ Allocating space for the temporary variables `tau`, `work`, and `b` on the data stack is a realization of Step 3. The variables `tau` and `work` are necessary because of the LAPACK routines used; `b` is a copy of `a` as the LAPACK routines, `dgeqrf`, and `zgeqrf`, modify the matrix in place, i.e. would mangle the input variable `a`. The temporaries are accessed the same way parameters are accessed: through indices into the data stack. These indices are `tauadr`, `wrkadr`, and `badr`. Their positions on the parameter stack are `top + 1`, `top + 2`, and `top + 3`, respectively.

We request a purely real storage for each of the three temporary variables, the third parameter being `realtype = 0`. Therefore, the index for the imaginary part is a dummy index, called `dumadr`.

❹ There is no "translation" to do in the real case. So Step 4 is quite simple. The input variable – of which we definitely know that it is real – is simply copied to the scratch space that we have allocated on the data stack.

Note how the *address* of the matrices is passed. The idiom is `stk(`*`index`*`)`, where *`index`* has been obtained through one of the get*-, or cre*-functions. The memnonic "stk" means data stack.

❺ Everything is set up correctly and initialized. We have reached Step 5. The worker routines can take over now.

❻ In the complex case the allocation of the temporaries variables requires a bit more thought, although it is again just Step 3. We know that the LAPACK routines need the complex vectors/matrices in packed form. Thus, we allocate *one* real (double precision) vector/matrix of twice the size each time thereby accommodating the storage requirement of complex (double complex, or complex*16) variables. Otherwise this step proceeds as in the real case.

❼ Because of the different handling of complex variables in Scilab and LAPACK, Step 4 requires two calls to the copy function.

```
      call dcopy(n * m, stk(areadr), 1, stk(badr),     2)
      call dcopy(n * m, stk(aimadr), 1, stk(badr + 1), 2)
```

The first line says: "Copy `m` times `n` elements from the first position of the double precision variable `stk(areadr)` taking each entry (3rd parameter, read stride = 1) to the double complex output variable `stk(badr)` filling every other entry (5th parameter, write stride = 2)." The second line does almost the same, but starts off writing at the second element `stk(badr + 1)`, therefore filling in the imaginary parts of `stk(badr)`. This corresponds to Step 4.

❽ Again we have reached Step 5; everything is set up correctly and initialized. The worker routines can take over.

❾ If there is an output variable, we copy the results into it. Otherwise, we skip the expensive copy operation.

(10) At this point a purely real result, `stk(badr)`, can simply be copied to the output parameter, `stk(rreadr)`.

The situation is a bit more complicated for a complex result, as we have to de-splice the `double complex` result from LAPACK into two `double precision` matrices. Here are the crucial lines again:

```
      call dcopy(m * n, stk(badr),     2, stk(rreadr), 1)
      call dcopy(m * n, stk(badr + 1), 2, stk(rimadr), 1)
```

The first line says: "Copy `m` times `n` elements from the first position in the double complex result `stk(badr)` taking every other entry (3rd parameter, read stride = 2) into the double precision output variable `stk(rreadr)` filling each entry (5th parameter, write stride = 1)." The second line does almost the same, but starts off at the second element, `stk(badr + 1)`, therfore copying the imaginary parts into `stk(rimadr)`. This way we are merging Step 6, and Step 7 into one.

6.3.2. Functionals

Func what? What are you talking about? Functionals – what is this? Glad you asked! Functions operate on numbers or variables, which themselves are not functions. The square root function for example is

usually applied to numbers (like: `sqrt(2)`) or more generally to variables (like: sqrt(x) for any real x). Functionals operate on other functions. Prominent examples are differentiation

where f as to fulfill certain continuity requirements at the point x_0; integration:

where again f has to fulfill certain (interesting) requirements; and Fourier transformation:

for suitable functions f, and integrals I.

The question how to write native Scilab functions that take arbitrary non-function parameters as their arguments has been discussed in the previous section. Now we focus on Scilab functions that take other Scilab functions as their arguments. If the reader does not feel familiar with native Scilab functions, she should reconsider Section 6.3.1.

In a similar manner as in the last section, we introduce an example. The example is taken from our Scilab/Quadpack interface available on the web. Among others it features the integrator `dqng` for sufficiently smooth functions, which has the follwoing signature:

```
type SimpleFunctionType is access
    function(X : in Float) return Float;

procedure dqng
```

```
(Function            : in     SimpleFunctionType;
 LowerIntervalEnd    : in     Float;
 HigherIntervalEnd   : in     Float;
 EpsilonAbsolute     : in     Float;
 EpsilonRelative     : in     Float;
 Result              :    out Float;
 ErrorAbsolute       :    out Float;
 NumberOfEvaluations :    out Natural;
 ErrorIndicator      :    out Natural);
```

Here comes the complete example.

Example 6-2. Scilab functional

```
      subroutine intsm
*
*     name:        intsm.f  -  Scilab/F77 interface to QUADPACK's dqng
*     author:      Lydia van Dijk <lydia_van_dijk@my-deja.com>
*     last rev.:   Wed Mar 15 23:49:45 UTC 2000
*     scilab ver.: 2.5
*     compiler:    g77-0.5.25 (Linux 2.3.49)
*
*     Scilab signature:
*         [res, abs_err, n_eval] = intsm(a, b, f, eps_abs, eps_rel)
*
*     Return Values:
*         res:      value of the integral
*         abs_err:  estimate of the absolute error
*         n_eval:   number of function evaluations
*
*     Arguments (mandatory):
*         a:        lower bound of integral
*         b:        upper bound of integral
*         f:        function to integrate with signature y = f(x),
*                   x, y real scalars
*
*     Arguments (optional):
*         eps_abs:  desired absolute error; default: 0.0
*         eps_rel:  desired relative error; default: 1e-8

      implicit none                                          –
Switch into weeny mode :-)

      include 'stack.h'
```



```
      common /cintg/ namef

      external bintg, fintg                                            –
gateways, see Section 6.3.3
      external setfintg

*     LOCAL VARIABLES
      character*6 namef                                                –
Name of the routine as string
      character*6 fname                                                –
Name of function to be integrated
      character*8 errstr

      logical getexternal, getscalar
      logical type, cremat

      integer iadr, sadr, neval, ifail, l, idxf, idxa
      integer topk, lr, lra, lrb, iipal, dummy

      double precision epsa, epsr, a, b, val, abserr

      include 'errnum.inc'                                             –
Error numbers are defined here

*     STATEMENT FUNCTIONS
      iadr(l) = l + l - 1                                              –
Accessor for integers on real*8 stack
      sadr(l) = l/2 + 1                                                –
Accessor for real* on integer stack

*     TEXT
      fname = 'intsm'                                                  – Name of this function
      if(rhs .lt. 3 .or. rhs .gt. 5) then           ❶
         call error(39)
         return
      endif
      topk = top                                                       – Remember stack position

*     pop optional parameters off the stack
      if(rhs .eq. 5) then                          ❷
         if (.not. getscalar(fname, topk, top, lr)) return
         epsr = stk(lr)
         top = top - 1
      else
         epsr = 1.0d-8                                                 – Scilab default
```

```
      endif

      if (rhs .ge. 4) then
          if (.not. getscalar(fname, topk, top, lr)) return
          epsa = stk(lr)
          top = top - 1
      else
          epsa = 0.0d0                                  – Scilab default
      endif

*     pop mandatory parameters off the stack
      namef = '      '
      type = .false.
      if (.not. getexternal(fname, topk, top, namef, type, setfintg)) ❸
     $        return
      idxf = top                                        –
Remember stack position of function f
      top = top - 1

      if (.not. getscalar(fname, topk, top, lrb)) return
      b = stk(lrb)
      top = top - 1

      if (.not. getscalar(fname, topk, top, lra)) return
      a = stk(lra)
      idxa = top                                        –
Remember stack position of argument a
      top = topk + 1                                    – Reset stack index

*     call integration routine
      if (type) then                          ❹
*     compiled external
          call dqng(fintg, a, b, epsa, epsr, val, abserr, neval, ifail)
      else                                    ❺
*     Scilab macro
          iipal = iadr(lstk(top))                       –
Start building a var. desc.
          istk(iipal) = 1                               –
          istk(iipal + 1) = iipal + 2                   –
          istk(iipal + 2) = idxf                        –
          istk(iipal + 3) = idxa                        –
          lstk(top + 1) = sadr(iipal + 4)               –
          call dqng(bintg, a, b, epsa, epsr, val, abserr, neval, ifail)
      endif
```



```
      if (ifail .eq. 1) then                    ❻
          buf = fname // ': max. number of steps reached; '
     $            // 'integral too difficult for int_sm'
          call error(emaxdiv)
          return
      endif
      if (ifail .eq. 6) then
          buf = fname // ': invalid error bounds'
          call error(ebounds)
          return
      endif
      if (ifail .ne. 0) then
*     catch all other errors
          write(errstr, '(I10)') ifail
          buf = fname // ': unknown error ' // errstr
          call error(eunknown)
          return
      endif

*     return values #1, and #2 (val, abserr) replace arguments #1, and
*     #2 (a, b).
      top = topk - rhs + 1
      stk(lra) = val                            ❼
      if (lhs .ge. 2) then
          top = top + 1
          stk(lrb) = abserr
      endif
*     return value #3, neval, needs space on the stack
      if (lhs .ge. 3) then                      ❽
          top = top + 1
          if (.not. cremat(fname, top, 0, 1, 1, lrb, dummy)) return
          stk(lrb) = dble(neval)                        –
neval is int, stk() is double precision
      endif

      end
```

❶ Here, we do not rely on the predefined number-of-arguments checking functions, checklhs and checkrhs, but set up out own scheme. `intsm` will require three mandatory aruments, *a*, *b*, *f*, and two optional ones, *eps_abs*, *eps_rel*, making a total of five.

❷ Take care of the optional parameters: fetch them from the stack, or use a default value if the actual parameter is omitted.

❸ Fetch mandatory parameters from the stack. The stack index, `top` is decremented with each parameter. This is a slight variation of the code shown in Example 6-1, where we keep the stack index fixed and add an appropriate offset when fetching the parameter from the stack.

❹ `getexternal` returns the type of the external after a successful call. An external, i.e. object code linked to Scilab, sets `type = 1`, a macro – difined via `deff`, or `function` – sets `type = 0`.

The case of an external is easy to handle as `getexternal` has already taken care of initializing the address to be called `fintg`. A call to `setfintg` accomplishes this.

❺ Calling a Scilab macro is much more involved.

❻ The return code form the integration routine `dqng` is checked, and errors are handled as described in Section 6.4.

❼ For `intsm` returns a scalar *and* the first arument is a mandatory scalar too, we do not need to reserve space for the value of the integral, `val`. The result is simply copied into the argument's stack position.

Almost the same holds for the second return value, `abserr`, though we only can use its slot if there actually is a return variable we are able to assign to.

❽ The third return value is a scalar, but the third argument is a function, so we cannot apply our previous technique again. `cremat` reserves the space for `neval`.

6.3.3. Dispatch Tables

FIXME: write it!

Also called "gateways".

```
/*
 * name:      quadqack-gw.c  -  gateway for all QUADPACK
 *                               interface routines
 * author:    Lydia van Dijk <lydia_van_dijk@my-deja.com>
 * last rev.: Wed Mar 15 02:22:02 UTC 2000
 * compiler:  pgcc-2.95.2 19991024 (Linux 2.3.49)
 */


#include <stack-c.h> /* lives in $SCI/routines */


typedef void (*gatef_t) (void);
```



```
extern void C2F(intals)(void);
extern void C2F(intcau)(void);
extern void C2F(intexc)(void);
extern void C2F(intfou)(void);
extern void C2F(intgen)(void);
extern void C2F(intgk)(void);
extern void C2F(intinf)(void);
extern void C2F(intosc)(void);
extern void C2F(intsm)(void);


static gatef_t gftab[] = {
    C2F(intals),
    C2F(intcau),
    C2F(intexc),
    C2F(intfou),
    C2F(intgen),
    C2F(intgk),
    C2F(intinf),
    C2F(intosc),
    C2F(intsm)
};


int
C2F(quadpack_gw)(void)
{
    (*gftab[Fin - 1])();
    return 0;
}
```

Scilab part ...

```
quadpacklibs = ['/site/src/netlib/quadpack/libquadpack-1.0.so', ..
                '/site/src/netlib/quadpack/intersci/libqpif-1.0.so']
gateway = 'quadpack_gw'  // name of the gateway function
interfaces = ['intals', 'intcau', 'intexc', 'intfou', ..
              'intgen', 'intgk',  'intinf', 'intosc', ..
              'intsm']

addinter(quadpacklibs, gateway, interfaces)
```

The complete example can be found in Section 8.6.

6.4. Error Handling

We briefly discuss how to produce the three possible classes of errors: fatal, warning, and message in Scilab.

6.4.1. Fatal Errors

To signal a fatal error condition in an interface procedure, call `error` with the appropriate code. The codes can be looked up in SCI`/routines/system/error.f`.

Here is a code snippet that does this.

```
if (ifail .eq. 2) then
    call error(1232)
    return
endif
```

If there is no suitable error message, place you own message (length <= 80 chars) in the global variable `buf`, and call `error` afterwards.

Warning

The string placed in `buf` must not be longer than 80 characters.

```
if (ier .eq. 6) then
    buf = 'invalid limits'
    call error(32253)
    return
endif
```

Sideffect of calling `error`: The Scilab stack is cleaned up, i.e. put back in the state it was just before the user routine has been entered.

On the Scilab interpreter level an error terminates the evaluation of whatever is currently evaluated (expression, file, or string), unless the trapping of errors has been modified by `errcatch`. See also: `errclear`, and `iserror`.

6.4.2. Warnings

To signal non-fatal error conditions (also known as soft-errors, or warnings), place a negative integer in `err2`, and call `out` to display the warning message. Depending on the situation a `return` may be issued after that. The Scilab stack is *not* cleaned up, which means all return values from the interface routine are

passed back normally. This is the solution of choice if the user can decide how to proceed, based on the return values.

Again, here is a small piece of code for demonstration.

```
if (fail .eq. 1) then
    err2 = -6343
    call out('reached table limit')
    return
endif
```

On interpreter level it is now mandatory to call `iserror` after a call to a routine that issues warnings like this. In the user-level error handler the error code *must* be reset by `errclear` to allow for further warnings to be reveived.

A typical way of coping with these soft-errors in the interpreter level is shown in Example 6-3.

Example 6-3. Handling of warnings in Scilab

```
[z, n, info] = abraxas(a, b, foo, limit)
if iserror(-19) then
        errclear(-19)
        limit = limit / 2    // make it easier
        [z, n, info] = abraxas(a, b, foo, limit)
        if iserror(-19) then
                errclear(-19)
                error(failed even with easy limit');
        end
end
```

6.4.3. Messages

Messages are the least severe class of errors. Sometimes they are not really errors, but just additional information that something unexpected is going on. No news is good news.

We have already seen the appropriate subroutine in action. It is `out`.

```
if (iter .gt. 1000) then
    call out('iterating excessively')
endif
```

6.5. Interface to Scilab's Core

The interface to Scilab's core is widely undocumented. To save the reader frequent lookups in the defining file, SCI`/routines/interf/stack1.f`, we have compiled the most important ones in the following sections: **query**, **access**, and **creation** of objects.

6.5.1. Query

The functions in this group retrieve information about the parameters a function has been called with, and about the properties of objects on the stack.

6.5.1.1. `checkrhs`

Synopsis

```
function CheckRhs
    (SelfName        : in String;
     MinNumParameter : in Integer;
     MaxNumParameter : in Integer)
    return Boolean;
```

Description

Check the number of actual parameters on the right-hand side to be in the range *`MinNumParameter`* `:` *`MaxNumParameter`*. Return true if it is in the range, otherwise raise error 77 associated with *`SelfName`*.

Note that a function that is called without any parameters, like *`Foo`*`()`, is assigned an `Rhs` of `-1`.

The semantics of `CheckRhs` are slightly goofy. If the number of actual input parameters is in the specified range, `CheckRhs` returns `True`, but it never returns `False` as it raises an error in this case.

Example

Ensure that at least 2, but not more than 5 parameters are passed to the function:

```
if (.not. checkrhs(fname, 2, 5)) return
```

We have assumed that `fname` is set to the function's name.

See also

`CheckLhs`, `Lhs`, `Rhs`

6.5.1.2. `checklhs`

Synopsis

```
function CheckLhs
    (SelfName        : in String;
     MinNumParameter : in Integer;
     MaxNumParameter : in Integer)
    return Boolean;
```

Description

Check the number of output variables, i.e. arguments on the right-hand side to be in the range *`MinNumParameter`* `:` *`MaxNumParameter`*. Return true if it is in the range, otherwise raise error 78 associated with *`SelfName`*.

Note that it is no error to supply less output parameters than the function actually returns. The extra values are silently discarded. This is true for the case of zero output values, too; then `ans` gets the first output value. Thus, a function called without any output parameters is assigned an `Lhs` of `1`.

The semantics of `CheckLhs` are slightly goofy. If the number of actual output parameters is in the specified range, `CheckLhs` returns `True`, but it never returns `False` as it raises an error in this case.

Example

Ensure that there are not more than 2 output parameters, when the function is called:

```
if (.not. checklhs(fname, 1, 2)) return
```

We have assumed that `fname` is set to the function's name.

See also

`CheckRhs`, `Rhs`, `Lhs`

6.5.1.3. `lhs`

Synopsis

```
Lhs : Integer
```

Description

The number of actual output parameters, i.e. those on the left-hand side of the assignment operator, is stored in the global variable `Lhs`.

See also

`CheckLhs`, `CheckRhs`, `Rhs`

6.5.1.4. `rhs`

Synopsis

```
Rhs : Integer
```

Description

The number of actual input parameters, i.e. those on the right-hand side of the assignment operator, is stored in the global variable `Rhs`.

See also

`CheckRhs`, `CheckLhs`, `Lhs`

6.5.2. Access Object

The functions in this section grant the programmer access to parameters that are stored on the Scilab stack. In general all of these functions work alike: An index to the current (i.e. as on entry of the function) top of the parameter stack, "BasePointer", and an index to the desired argument, "StackPointer", are passed to the API. On return the user gets all necessary information about the argument like sub-type, dimension as well as the indices, "*Foo*Index" that index into the Scilab heap. The indices act like pointers to the actual contents. This way only meta-data is passed, saving time-consuming copy operations.

6.5.2.1. `getmat`

Synopsis

```
function GetMat
    (SelfName       : in     String;
     BasePointer    : in     ParameterStackAddress;
     StackPointer   : in     ParameterStackAddress;
     IsComplex      :    out ComplexFlag;
     Rows           :    out Natural;
     Columns        :    out Natural;
     RealIndex      :    out DataStackIndex;
     ImaginaryIndex :    out DataStackIndex)
    return Boolean;
```

Description

Retrieve the address(es) and dimensions of a real or complex matrix from the parameter stack. The *`BasePointer`* must be set to the parameter stack pointer's value on entry of the *calling* function. *`StackPointer`* points to the desired parameter on the parameter stack. If successful, `GetMat` returns `True`, and *`IsComplex`*, *`Rows`*, *`Columns`*, and *`RealIndex`* are valid. If *`IsComplex`* `= ComplexVariable` then *`ImaginaryIndex`* is valid, too. If the parameter indexed by *`StackPointer`* is not a matrix `GetMat` returns `False`.

The output parameter *`IsComplex`* indicates whether the matrix on the data stack is purely real or complex. In the first case *`RealIndex`* points to the matrix, in the second case *`RealIndex`* points to the real part of matrix, and *`ImaginaryIndex`* points to the imaginary part. In any case *`Rows`* and *`Columns`* are the number of rows and columns of the matrix.

Example

Fetch the addresses of a possibly complex `m`-times-`n` matrix from position `top` of the parameter stack.

```
if (.not. getmat(fname, topk, top, iscmpx, m, n, are, aim)) return
```

It is assumed that `fname` has been set to the function's name, and `topk` carries the position of the stack on entry to the calling function.

See also

`GetRMat`, `GetRVect`, `GetVect`

6.5.2.2. getrmat

Synopsis

```
function GetRMat
    (SelfName     : in     String;
     BasePointer  : in     ParameterStackAddress;
     StackPointer : in     ParameterStackAddress;
     Rows         :    out Natural;
     Columns      :    out Natural;
     RealIndex    :    out DataStackIndex)
    return Boolean;
```

Description

Function `GetRMat` works like function `GetMat`, but restricts the accepted matrices to purely real ones.

See also

`GetMat`, `GetRVect`

6.5.2.3. getrvect

Synopsis

```
function GetRVect
    (SelfName     : in     String;
     BasePointer  : in     ParameterStackAddress;
     StackPointer : in     ParameterStackAddress;
     Rows         :    out Natural;
     Columns      :    out Natural;
     RealIndex    :    out DataStackIndex)
    return Boolean;
```

Description

Function `GetRVect` works like function `GetRMat`, but restricts the accepted matrices to either single rowed (1-times-N) or single columned (N-times-1).

See also

`GetVect`, `GetRMat`

6.5.2.4. getvect

Synopsis

```
function GetVect
    (SelfName       : in     String;
     BasePointer    : in     ParameterStackAddress;
     StackPointer   : in     ParameterStackAddress;
     IsComplex      :    out ComplexFlag;
     Rows           :    out Natural;
     Columns        :    out Natural;
     RealIndex      :    out DataStackIndex;
     ImaginaryIndex :    out DataStackIndex)
    return Boolean;
```

Description

Function `GetVect` works like function `GetMat`, but restricts the accepted matrices to either single rowed (1-times-N) or single columned (N-times-1).

See also

`GetMat`, `GetRVect`

6.5.2.5. getscalar

Synopsis

```
function GetScalar
    (SelfName     : in     String;
     BasePointer  : in     ParameterStackAddress;
     StackPointer : in     ParameterStackAddress;
     Index        :    out DataStackIndex)
    return Boolean;
```

Description

Retrieve the address and dimensions of a real or complex scalar from the parameter stack. The *`BasePointer`* must be set to the parameter stack pointer's value on entry of the *calling* function. *`StackPointer`* points to the desired parameter on the parameter stack. If successful, `GetScalar` returns `True`, and *`Index`* is valid. If the parameter indexed by *`StackPointer`* is not a scalar `GetScalar` returns `False`.

See also

`GetVect`, `GetMat`

6.5.2.6. `getexternal`

Synopsis

```
type FortranIdentifier is
    array (1 .. 6) of Character;  - Fortran's 6 char limit

- SimpleFunctionType is just an example
type SimpleFunctionType is access
    function(X : in Float) return Float;

type InstallerProcedureType is access
    procedure(FunctionName       : in     FortranIdentifier;
              FunctionEntryPoint : in     SimpleFunctionType);

function GetExternal
    (SelfName     : in     String;
     BasePointer  : in     ParameterStackAddress;
     StackPointer : in     ParameterStackAddress;
     FunctionName : in out FortranIdentifier;
     IsExternal   :    out Boolean;
     Installer    : in     InstallerProcedureType)
    return Boolean;
```

Description

The first three parameters *`SelfName`*, *`BasePointer`*, and *`StackPointer`* work exactly the same as in the other functions, so does the return value. They explanations are not repeated here, see e.g. Section 6.5.2.1.

The fourth parameter, *`FunctionName`* is the name of the function to be called. This parameter can designate an external function, i.e. code that has been compiled seperately and then linked to Scilab via, e.g. `link`, or the name of a Scilab function that has been defined with `function` or `deff`. In any case it is simply the name of the function. On return the *`IsExternal`* parameter signals `True` if the function is a external and `False` otherwise.

The last parameter is very special. It specifies the installation procedure *`Installer`* that manipulates a dispatcher function *`DispatchFunction`*. After a call to *`Installer`* the dispatcher points to the user function given by *`FunctionName`*.

Note: The programmer *should not* issue such a call; it is already done by `GetExternal`.

Examples of dispatcher functions, the associated hooks and dispatch tables are e.g. found in SCI`/routines/default/FTables.{h,c}`.

Support Functions

The `GetExternal` function relies heavily on various support functions. The supporting function have to be set up previous to the first call. Usually they are installed by the user's extention package unless she/he decides to (ab)use existing dispatcher tables and functions.

```
package ExternalSupport is
    procedure InstallProcedure
        (FunctionName       : in     FortranIdentifier;
         FunctionEntryPoint : in     SimpleFunctionType);
end ExternalSupport;

package body ExternalSupport is

    with Ada.Characters.Latin_1;

    type FunctionTableEntry is
        record
            FunctionName    : FortranIdentifier;
            FunctionAddress : SimpleFunctionType;
        end record;

    type FunctionTableType is
        array (Positive range <>) of FunctionTableEntry;


    - SimpleExample is of type SimpleFunction
    function SimpleExample(X : Float) return Float;
    begin
        return X;
    end Hook;


    FunctionTable : FunctionTableType(1 .. 2) :=
        (1 => (FunctionName    => "exampl",
               FunctionAddress => SimpleExample'Access),
         2 => (FunctionName    => (others => Ada.Characters.Latin_1.NUL),
               FunctionAddress => null));
```

```
    DispatchFunction : SimpleFunctionType;


    - function Hook is the hard-coded target for all
    - internal calls

    function Hook(X : Float) return Float;
    begin
        return DispatchFunction.all(X);
    end Hook;


    - bend hook function to point to FunctionEntryPoint

    procedure InstallProcedure
        (FunctionName       : in     FortranIdentifier;
         FunctionEntryPoint : in     SimpleFunctionType);
    begin
        DispatchFunction := SetFunction(FunctionName,
                                        FunctionEntryPoint,
                                        FunctionTable);
    end InstallProcedure;

end ExternalSupport;
```

Example

For a self-contained example, see Section 6.3.2.

See also

Functionals, Dispatch Tables

6.5.3. Create Object

The object creation functions are mainly used to setup temporary variables for the current procedure or the procedures to be called; they bear a lot of resemblance with the object access functions (see also Section 6.5.2). The difference is that a new object is created and therefore stack space is used.

6.5.3.1. `Cremat`

Synopsis

```
function CreMat
    (SelfName       : in     String;
     StackPointer   : in     ParameterStackAddress;
     WantComplex    : in     ComplexFlag;
     Rows           : in     Natural;
     Columns        : in     Natural;
     RealIndex      :    out DataStackIndex;
     ImaginaryIndex :    out DataStackIndex)
    return Boolean;
```

Description

Notes

1. We apologize to all Ada programmers for the abuse of the language, but Ada’s expressiveness and clarity are unmatched.

Chapter 7. Further Information

FIXME: need text here

7.1. Coping With Scilab

Scilab is a large package no doubt about that. The source for version 2.5 comprises of more than 48 MB, and builds to over 88 MB on an ia32 GNU/Linux system.

7.1.1. Distribution Size

We use several tools to cope with Scilab's size and complexity. The most important ones are introduced in the following section.

7.1.1.1. CVS

CVS (http://www.sourcegear.com/CVS) is one of the most commonly used version control systems. A set of source files (which can be binary) is put under revision control by "checking it in". The important difference to an older version control system, RCS is the notion of a module which refers to the complete set of sources. Usually the set consists of a whole directory tree, as e.g. all Scilab sources.

Also check out Pascal Molli's *information and FAQ* (http://www.loria.fr/~molli/cvs-index.html) on CVS. In larger development environments CVS with its relaxed rules might not be the adequate tool. In these cases *Aegis* (http://www.pcug.org.au/~millerp/aegis/aegis.html) can be used.

7.1.1.2. locate

The **locate**(1) command is the fast brother of the **find**(1) command. More precisely, **locate** accesses a precomputed database of filenames (usually `/var/lib/locatedb`; for its structure see `locatedb`(5)). The database is generated by **updatedb**(1) with a `find / -print` and then mangled for faster access.

We have found a local filename database very useful for the work with large projects. Therefore, we have set up two aliases that create and access a project-specific list of filenames.

```
alias upd='updatedb -output=./.locatedb -localpaths=.'
alias loc='locate -database=./.locatedb'
```

The **upd** sequence is typically run after a CVS checkout, add, or remove in the directory SCI.

We run **loc** whenever we are looking for a file in the Scilab distribution. This is much faster than running **find** every time, especially when working with a slow file server. The only inconvenience remaining is that **loc** must be executed in the directory where the database resides, here: SCI. However this is more than compensated by the fact that **locate** does a substring search, i.e. given the filename *`fpat`* it returns all file- and directory names matching the regulat expression `.*`*`fpat`*`.*`.

If we want to scan the complete database and postprocess the output with our tools-of-choice, issuing a `loc .` and piping the output through the desired filters does the job.

7.1.1.3. Glimpse

What the **updatedb/locate** pair is for filenames the **glimpseindex/glimpse** pair is for file contents. **glimpseindex**(1) generates a database that is accessed by the user via **glimpse**(1). So,

```
glimpse pattern
```

corresponds to the non-database backed command, namely a recursive **grep** over a set of directories like

```
find . -print | xargs grep pattern
```

assuming that the database has been generated for "`.`". Again the fast version is so helpful that we have defined two aliases.

```
alias glidx='if test -f .glimpse_index; then
                 glimpseindex -H . -o -f -B .;
             else
                 glimpseindex -H . -o -B .;
             fi'
alias gl='glimpse -H .'
```

The first alias, **glidx**, is *one* line. It has been broken into several lines only to make its workings clear; namely if an index file already exists it is updated (`-f` option), otherwise it is generated from scratch.

Like our **locate** aliases everything is happening in the current directory which means that **glidx** should be called from SCI.

Glimpse is not part of most of the standard GNU/Linux distributions (at least not *SuSE* (http://www.suse.de)-6.2, and *RedHat* (http://www.readhat.com)-6.1, the ones we checked). The University of Arizona currently hosts the *Glimpse home page* (http://webglimpse.org/), and Glimpse can also be downloaded from *SC0's software archive* (ftp://ftp.sco.com/skunkware/src/fileutil/), which is mirrored by *Sunsite UK* (ftp://ftp.sunsite.org.uk/Mirrors/ftp.sco.com/skunkware/src/fileutil/).

7.1.2. Bug Hunting

In preparation of this document (lvd), and in our daily work (cls) we have found it very useful to have more than one Scilab. What? More than one running process? – No, more than one binary of `scilex`! In fact three different versions all come in handy depending on the task:

`scilex` binaries

Code optimized for execution speed

The common name is "production quality code", but Scilab is so far away from production quality that we shall not use that term.

This `scilex` is built with all compiler optimizations enabled. Furthermore all compiler switches and options are specifically tuned for the machine the code will run on in the future (see Section 5.3). Maximum performance is the only goal and no attempt is made to retain any debugging information.

Debugging Code

This `scilex` is not optimized, instead it carries complete debug information. Thus, it is ideally suited for interactive debugging sessions, and single-line tracing.

Profiling Code

The third variant is a profiling version of `scilex`. It is not optimized for speed either.

Profiling is the first step of any tuning. Furhtermore, during our work with and on Scilab we have found it very helpful to be able to answer the notorious question: "Where is it buring the cycles?" Profiling – done right – is much faster than timing individual "suspects", although analyzing the profiler output requires some skill.

See also Section 4.4.1.

7.2. Local Documents

Following documents come with every source distribution of Scilab. They live in the directory SCI`/doc`.

Standard Documentation

`Comm.ps` – Communication Toolbox

Description of geci an interactive communication manager.

Whenever Scilab is started with the **scilab**-script in fact geci takes over and starts **scilex** as a process on the local machine. geci is not limited to local processes; remote machines can be accessed transparently.

`Comm.ps` describes the commands available from within Scilab to communicate with other applications via geci. Moreover it elaborates how to write C-applications that communicate with Scilab via geci and the associated library.

`Internals.ps` – Guide for Developers

`Internals.ps` is the terse breakdown of the innermost core of scilex. It contains descriptions of the most holy variables like `stk`, the stack structure, and the internal variable representation.

The last third treats interfacing user routines with the core and consists mostly of scarcely documented Fortran program listings.

This is a document for gurus, and certainly not suited for casual reading.

`Intro.ps` – Users Guide

This is not a must-read this is more, it is a must-print-and-store-near-the-computer. The Intro is *much* more than just an introduction to Scilab.

The chapter breakdown is as follows:

- Introduction,
- Data Types,
- Programming,
- Basic Primitives,
- Graphics, and
- Interfacing.

The appendices cover a Demo Session, System Interconnection, and a brief section about converting Scilab code to F77 code.

Most interesting for beginners is the chapter “Introduction”. Combining it with browsing over “Data Types” and “Graphics” the novice should be all set for her/his first steps.

The advanced user will want to come back the “Data Types” regularly and also study “Programming” in detail. Any chapter but “Interfacing” should be understandable at that level of knowledge. It is no shame to come back to the Intro over and over again as Scilab is rich in data types and graphics commands.

Far beyond the introductory stuff the chapter on “Interfacing C or Fortran routines” wraps up the Intro. The topics treated here are for the aspiring Scilab master. Dynamic and static extensions are discussed in depth. The companion program intersci for automatic Fortran77-interface generation is treated in detail.

`Lmi.ps` – LMI-Optimization Toolbox

FIXME: someone please write it!

`Manual.ps` – Reference Manual

The `Manual.ps` is an automatically generated compilation of all Scilab user-variables and user-commands. It is a compilation in the truest sense of the word as all the help texts available online through the **help**-command are catenated to one huge (over 700 pages) file.

`Metanet.ps` – Graphs and Networks Toolbox

FIXME: someone please write it!

`Scicos.ps` – Dynamic System Builder And Simulator Toolbox

FIXME: someone please write it!

`Signal.ps`– Signal Processing

FIXME: someone please write it!

7.3. Hyperlinks

Here are a few links that are useful in connection with Scilab.

Links

INRIA (http://www-rocq.inria.fr) official Scilab pages

- *Scilab Home page* (http://www-rocq.inria.fr/scilab/)
- *Parallel Scilab Home page* (http://www.ens-lyon.fr/~desprez/FILES/RESEARCH/SOFT/SCILAB/)
- *Scilab FAQ* (http://www-rocq.inria.fr/scilab/faq/index.html)
- *Scilab FTP Site* (ftp://ftp.inria.fr/INRIA/Projects/Meta2/Scilab/distributions/)

Pages Of Scilab Enthusiasts (alphabetically)

- *Lydia van Dijk’s Scilab page* (http://www.lightlink.com/lydia/scilab.html)
- *Stéphane Mottelet’s Scilab page* (http://www.dma.utc.fr/~mottelet/scilab/)
- *Bruno Pinçon’s Scilab page* (http://www.iecn.u-nancy.fr/~pincon/scilab/scilab.html) featuring a nice French introduction to Scilab.
- *Enrico Segre’s page* (http://www.polito.it/~segre/)

Free Numerical Libraries

Netlib (http://www.netlib.org/index.html)

Netlib gathers a huge amount of free numerical libraries. Due to the nature of the business most of them are written in Fortran-77.

GAMS (http://math.nist.gov/)

The Guide to Available Mathematical Software. This is the lazy man’s entry point to Netlib. If you know your problem then just follow the decision tree until you reach the module that deals with it. Can it get any better than that?

Lapack (http://www.netlib.org/lapack/index.html)

The all time classic of the numerical libraries features linear equation, linear least squares, singular value, and (even generalized) eigenvalue solving in an orthogonal design: 4 precisions (real, double precision, complex, double complex), several matrix storage schemes (rectangular dense, symmetric/hermitian positive definite, banded, tridiagonal, ...).

Optimized BLAS (and sometimes more) Libraries

ATLAS (http://www.netlib.org/atlas/)

ATLAS is an acronym for Automatic Tuning of Linear Algebra Software. A novel approach to optimize the BLAS library for an arbitrary architecture with deep memory hierarchie and pipelined functional units. In other words for any modern machine.

ASCI RED (http://www.cs.utk.edu/~ghenry/distrib/)

This is an acronym for Advanced Strategic Computing Initiative. In the course of their research they develop highly optimized BLAS libraries for PPro and later 1ntel processors.

DIGITAL™'s *Extended Math Library* (http://www.digital.com/info/hpc/software/dxml.html)

FIXME: Someone please say something about it!

Intel™'s *Math Kernel Library* (http://developer.intel.com/vtune/perflibst/mkl/) (MKL)

This is the one for the poor souls being trapped on the Dark Side. The MKL runs with the software-from-hell (aka from Redmond/WA).

IBM™'s *Essential Scientific Software Library* (http://www.austin.ibm.com/software/sp_products/esslspec.htm) (ESSL)

FIXME: Someone please say something about it!

SUN™'s *Fortran High-Performance Library* (http://www.sun.com/workshop/fortran/) (PerfLib)

FIXME: Someone please say something about it!

SGI™'s *Scientific Library* (http://www.sgi.com/software/scsl.html) (SCSL)

FIXME: Someone please say something about it!

Free Mathematical Software

Pari (http://www.parigp-home.de/)©

An extremely fast arbitrary precision calculator with a library that can be linked with user programs.

MuPAD (http://math-www.uni-paderborn.de/MuPAD/index.html)©

Symbolic algebra program which is Map1e alike. It features one of the most comprehensive integration libraries currently available.

Octave (http://www.che.wisc.edu/octave/)©

Matlab© compatible matrix package, whose core is written in C and C++; relies on classical Fortran libraries like LAPACK. The Octave libraries can be integrate easily in new user programs.

Tela (http://www.geo.fmi.fi/prog/tela.html)©

Tela is the short form for tensor language; it is written in C++. Currently tensors up to rank 4 are supported. Tela is less compatible to the original than Scilab or Octave as it uses a Pascal-like syntax with C-like identifiers. The strength of Tela is its tremendously fast interpteter (in some cases more than a factor of 100 faster than Octave, and more than 40 times faster than Scilab).

The Tela language supports modules like Perl does. Functions can have optional arguments and mandatory return values (to implement in-place modification). Moreover so-called C-tela files allow the user to write C-functions using extensions from tela to produce extremely efficient extensions.

Free Plotting Software

GNUPlot (http://WWW.cs.dartmouth.edu/gnuplot_info.html)©

Although the prefix is a pure coincidence with the *GNU project* (http://www.gnu.org) it is one of the best 2d-plotting programs that are available with source code.

PlotMTV (ftp://ftp.x.org/contrib/applications/)©

An older, but still excellent program for 2d- and 3d-plots. The “philosophy” is different from GnuPlot as plotting instructions and data share the same file.

Chapter 8. Complete Examples

Welcome to our attic! Following the style of the bag-of-trick, the examples gathered here are an unsorted collection of hacks that has piled up over the years. A few functions are used or discussed in the earlier section, but were truncated to emphasized the important parts. Here you only find complete versions.

8.1. `benchmark.sci`

This example shows a benchmark function that tries hard to do better than others. In the first step the timer resolution is determined. Next the function under test is executed in a loop and the time taken is estimated. This time in turn is used for the final test. The number of loop iterations is chosen according to the preliminary test. Finally, the median of the timings is returned.

```
function res = calibrate(max_len, n_avg, log_inc)
// determine the resolution of Scilab's built-in timer
// Return vector with measured timer resolution(s)

[nl, nr] = argn()
if nr <= 2, log_inc = 1.1, end
if nr <= 1, n_avg = 31, end
if nr == 0, max_len = 100000, end

r = []
n = 1
while n <= max_len
    //disp(n)
    tv = []
    iter = 0:n
    for k = 1:n_avg
        timer()
        for i = iter, end
```



```
        t = timer()
        tv = [tv; t]
    end
    tv = sort(tv)
    r = [r; [n, tv($/2 + 1)]]
    n = log_inc * n
end

// xbasc(); plot2d(r(:,1), r(:,2), -1)

delta = [r(:, 2); r($, 2)] - [r(1, 2); r(:, 2)]
idx = find(delta > %eps)
res = delta(idx)

function tpl = benchmark()
// return the time for one loop round trip

verbose = %t
min_test = 10  // minimum multiple of the timer
               // resolution to run coarse test
std_test = 200 // as min_test but for real test
n_avg = 31     // number of samples to calculate median
log_inc = 2.0  // logarithmic increment in coarse test

// inquire timer properties
disp('+++ calibrating timer')
resol = calibrate()
if size(resol, '*') <= 2 then
    error('calibration failed')
end
if resol(1) ~= resol(2) then
    warning('calibration botched; proceeding anyway...')
end
t_resol = resol(1);
if verbose
    disp('timer resolution is ' + string(t_resol) + 's')
end

// rough estimate of time
disp('+++ calibrating test')
np = 1
timer()
mytest()
t = timer()
while t < min_test * t_resol
```

```
    //disp(np, t, min_test * t_resol)
    np = log_inc * np
    timer()
    for i = 1:np
        mytest()
    end
    t = timer()
end
if verbose then
    disp('coarse, ' + string(np) + ' round trips in '
         + string(t) + 's')
end
if np == 1 then
    warning('slow procedure under test - time may be excessive')
end

// run real test
disp('+++ running test')
tc = t / np
ne = ceil(std_test * t_resol / tc)
if verbose then
    disp('fine, test will take about '
         + string(ceil(tc * ne * n_avg)) + 's')
end

r = []
for k = 1:n_avg
    timer()
    for i = 1:ne
        mytest()
    end
    t = timer()
    r = [r; t]
end
if verbose then
    disp('fine, ' + string(ne) + ' round trips in '
         + string(t) + 's')
end

// get median and return
r = sort(r)
//disp(r)
tpl = r($/2 + 1) / ne

function mytest()
```



```
exact = -2.5432596188;
z = abs( exact - intg(0, 2 * %pi, f) )

function y = f(x)
//y = x * sin(30 * x) / sqrt( 1 - ((x / (2 * %pi))^2) )
y = x
```

8.2. listdiff.sci

The function `listdiff` returns the differences of two vectors in the style of the **diff**(1) command. It is a funny example of doing something completely non-numerical with Scilab.

```
function diff = listdiff(lst1, lst2, equ)
// implement a diff(1) like function for vectors.
// The caller can supply a bool equ(x, y) function
// that will be used in all comparisions, otherwise
// operator '==' is used.
//
// RETURN VALUE
//   2-column vector describing the differences.
//   Column 1 contains the element and column 2
//   the element's index.  A '+' in front of the
//   index means: 'Extra element in lst2', a '-'
//   means missing element in lst1.
//
// AUTHOR
//   Christoph L. Spiel
//
//   Copyright 1999 by Christoph Spiel
//   listdiff is free software distributed under the terms
//   of the GNU General Public License, version 2.
//!

[nl, nr] = argn(0);
select nr
case 0 then
    error('Too few arguments.  Got 0, require 2 or 3.');
case 1 then
    error('Too few arguments.  Got 1, require 2 or 3.');
case 2 then
    deff('b = equ(s1, s2)', 'b = s1 == s2');
case 3 then
```

```
    // caller supplied equ()
    if type(equ) ~= 13 then
        error('Function expected, got a ' + typeof(equ) + '.');
    end
else
    error('Too many arguments.  Got ' + string(nr) + ' require 2 or 3.');
end

if type(lst1) ~= 1 & type(lst2) ~= 1 then
    // none of the lists is empty
    if type(lst1) ~= type(lst2) then
        error('Both lists must be of the same type, or be empty.');
    end
end

fuzz = 10;

diff = [];
n1 = size(lst1, 1);
n2 = size(lst2, 1);


// special cases

if n1 == 0 & n2 == 0, return, end

if n1 == 0 then
    p = 1 : n2;
    diff = [lst2, '+' + string(p')];
    return;
end

if n2 == 0 then
    p = 1 : n1;
    diff = [lst1, string(-p')];
    return;
end


// general case (neither list is empty)

i = 1;
j = 1;
while i <= n1 & j <= n2
    while i <= n1 & j <= n2
```



```
        if ~equ(lst1(i), lst2(j)), break, end
        i = i + 1;
        j = j + 1;
    end
    if i >= n1 | j >= n2, break, end

    icurs = i;
    while icurs <= min(n1, i+fuzz)
        if equ(lst1(icurs), lst2(j)), break, end
        icurs = icurs + 1;
    end
    if icurs <= n1 then
        if equ(lst1(icurs), lst2(j)) then
            // record element(s) missing from lst1
            for p = i : icurs-1
                this_diff = [lst1(p), string(-p)];
                diff = [diff; this_diff];
            end
            // re-sync
            i = icurs;
        end
    end

    jcurs = j;
    while jcurs <= min(n2, j+fuzz)
        if equ(lst1(i), lst2(jcurs)), break, end
        jcurs = jcurs + 1;
    end
    if jcurs <= n2 then
        if equ(lst1(i), lst2(jcurs)) then
            // record extra element(s) in lst2
            for p = j : jcurs-1
                this_diff = [lst2(p), '+' + string(p)];
                diff = [diff; this_diff];
            end
            // re-sync
            j = jcurs;
        end
    end
end
```

8.3. whatis.sci

```
function rv = whatis(name_arr)
// NAME
//   whatis - listing of variables in extended format
//
// CALLING SEQUENCE
//   whatis()
//   whatis(name_arr)
//
// PARAMETER
//   name_arr : array of variables names
//
// DESCRIPTION
//   whatis returns a column-vector with the names,
//   types, and sizes of all local variables
//   (first form), or only of the variables whose
//   names (as strings!) are given in the matrix
//   name_arr (second form).
//
// EXAMPLES
//   whatis()
//   whatis('my_mat')
//   whatis(['foo' 'bar' 'baz'; 'foobar 'morefoo' 'foobaz'])
//
// SEE ALSO
//   who, whos
//
// AUTHORS
//   Enrico Segre, Lydia van Dijk
//
//   Copyright 1999 by Enrico Segre and Lydia van Dijk.
//   whatis is free software distributed under the terms
//   of the GNU General Public License, version 2.
//!

// LAST REVISION
// lvd, Fri Dec  3 01:01:45 UTC 1999

// TO DO/TO FIX
//
// - Accepting a regexp as an argument would be nice.  This in turn
//   leads to complete boolean expressions doing the variable selection
//   resembling what the UNI* find utility does.  Example:
//   All vars ending in a 'v' that are complex and larger than
```



```
//   1000 words.
// - The behavior with undefined variables is unsatisfactory.


[nl, nr] = argn(0);
clear nl;
if nr == 0 then
    // no arguments -> take all variables like whos() does
    clear nr;
    name_arr = sort(who('get'));
end
clear nr;

if type(name_arr) ~= 10 then
    error('Expecting a string or an array of strings, got a ' ..
            + typeof(name_arr) + '!');
    return;
end

[namev, memv] = who('get'); // get memory usage of all local vars

// define isreal() for hypermatrices
deff('b = %hm_isreal(hm)', ..
     'if size(hm, "*") == 0 then b = %t; ..
      else ..
          b = isreal(hm(1)); ..
      end');

deff('b = hm_isbool(hm)', ..
     'if size(hm, "*") == 0 then ..
          b = %t; ..
      else ..
          b = type(hm(1)) == 4; ..
      end');

deff('b = hm_isstring(hm)', ..
     'if size(hm, "*") == 0 then ..
          b = %t; ..
      else ..
          b = type(hm(1)) == 10; ..
      end');

deff('b = hm_isint(hm)', ..
     'if size(hm, "*") == 0 then ..
          b = %t; ..
```

```
      else ..
          b = type(hm(1)) == 8; ..
      end');

rv = [];
for name = matrix(name_arr, 1, size(name_arr, '*')) do
    if isdef(name) then
        clear var;
        var = evstr(name);     // convert var's name back into var
        //
        // type classification
        //
        ty = type(var);        // type number
        select ty              // type 16 and 17 are not recognized
        case 16 then           //   by the function typeof()
            tgenp = %f;        // we know the tlist's type for these
            lab = var(1);      // vector of labels
            select lab(1)      // 1st label defines the type
            case 'ar' then
                tnam = 'ARMAX process';
            case 'des' then
                tnam = 'descriptor system';
            case 'linpro' then
                tnam = 'linear programming data';
            case 'lss' then
                tnam = 'linear system';
            case 'r' then
                tnam = 'rational';
            case 'scs_tree' then
                tnam = 'SCICOS navigator data';
            case 'xxx' then
                tnam= 'SCICOS menu data';
            else
                tnam = 'generic tlist ' + lab(1);
                tgenp = %t;
            end // select lab(1)
        case 17 then
            tnam = 'hypermatrix';
        else
            tnam = typeof(var);         // type name, a string
        end // select ty
        if ty==1 | ty==2 | ty==5 | ty==17 then
            // boolean, string, integral, real, or complex,
            // possibly sparse matrix or hypermatrix (yuck!)
            if hm_isbool(var) then
```



```
                tnam = 'boolean ' + tnam;
            elseif hm_isstring(var) then
                tnam = 'string ' + tnam;
            elseif hm_isint(var) then
                tnam = 'int ' + tnam;
            else
                if isreal(var) then
                    tnam = 'real ' + tnam;
                else
                    tnam = 'complex ' + tnam;
                end
            end
        end
        tmp = name + ':  ';
        //
        // size determination
        //
        if ty==1 | ty==2 | ty==4 | ty==5 | ty==8 | ty==10 | ty==17 then
            // any kind of matrix
            sz = size(var);           // var's dimensions
            tmp = tmp + string(sz(1));
            for j = 2:length(sz)
                tmp = tmp + 'x' + string(sz(j));
            end
            tmp = tmp + ' ';
        elseif ty==16
            // user-defined aka generic tlist
            if tgenp then
                tmp = tmp + string(size(var(1), '*')) + ' element ';
            end
        else
            // function, library, or other non-atomic object
        end
        //
        // memory usage
        //
        i = find(namev == evstr('name')); // index of var's entry
        tmp = tmp + tnam + ', ' + string(memv(i)) + ' words';
    else
        tmp = "'" + name + "' is not defined';
        warning('variable ' + tmp);
    end
    rv = [rv; tmp];
end
```

8.4. Auto-Determination of Precedence and Associativity

`assoc.sci`, `prec.sci`, and `parser.sci` are the scripts that determine the precedence and the associativity of the arithmetic Scilab operators. The results are used in Section 4.1.

8.4.1. assoc.sci

```
function a = assoc(oper, typ)
// Return the associativity a of
// operator oper which accepts type typ.
// oper can be a matrix of operators.
//
// typ can be 'n' for numeric, or 'b' for boolean.
// If typ is omitted, numeric is assumed.

[nl, nr] = argn()
if nr == 1 then
    typ = 'n'
end

select typ
case 'n' then
    args = string([1.1 1.2 1.5])
    deff('b = equal(x, y)', 'b = abs(x - y) < 1.2*%eps')
case 'b' then
    args = string(['%f' '%t' '%f'])
    deff('b = equal(x, y)', 'b = x == y')
else
    error('unknown type ' + typ)
end

a = []
for op = oper
    expr = '[' + args(1) + op
               + args(2) + op + args(3) + ',' ..
               + '(' + args(1) + op + args(2) + ')'
               + op + args(3) + ',' ..
               + args(1) + op + '(' + args(2)
               + op + args(3) + ')]'
    //disp(expr)
    r = evstr(expr)
    //disp(r)
```



```
    if equal(r(2), r(3)) then
        a = [a 'non']
    elseif equal(r(1), r(2)) then
        a = [a 'left']
    elseif equal(r(1), r(3)) then
        a = [a 'right']
    else
        error('could not determine associativity')
    end
end
```

8.4.2. prec.sci

```
function p = prec(op1, op2)
// determine the relative precedence of operator op1 vs op2
// If operator op1 has a higher precedence than op2 then p = -1.
// In the opposite case p = 1.  If both have the same precedence
// level p = 0

args = string([1.1 1.2 1.5])
deff('b = equal(x, y)', 'b = abs(x - y) < 1.2*%eps')

expr = '[' + args(1) + op1 + args(2) + op2 + args(3) + ',' ..
           + '(' + args(1) + op1 + args(2) + ')' + op2 + args(3) + ',' ..
           + args(1) + op1 + '(' + args(2) + op2 + args(3) + ')]'

//disp(expr)
r = evstr(expr)
//disp(r)

if equal(r(2), r(3)) then
    p = 0
elseif equal(r(1), r(2)) then
    p = -1
elseif equal(r(1), r(3)) then
    p = 1
else
    error('could not determine precedence level')
end

function p = prec1(uop, op)
// determine what relative precedence the unary operator uop has
// with respect to operator op.  The return values are like those
```

```
// of prec()

args = string([1.1 1.2])
//args = string([(1.1+0.9*%i) (1.2-0.8*%i)])
deff('b = equal(x, y)', 'b = abs(x - y) < 1.2*%eps')

expr = '[' + uop + args(1) + op + args(2) + ',' ..
           + '(' + uop + args(1) + ')' + op + args(2) + ']'

//disp(expr)
r = evstr(expr)
//disp(r)

if equal(r(1), r(2)) then
    p = -1
else
    p = 1
end

function p = lprec(op1, op2)
// determine relative precedence of the
// logical operators op1 and op2

v = ['%f' '%t']
for i = 1:2
    for j = 1:2
        for k = 1:2
            args = string([v(i) v(j) v(k)])
            expr = '[' + args(1) + op1 + args(2) + op2 + args(3) + ',' ..
                       + '(' + args(1) + op1 + args(2) + ')' + op2 + args(3) + ',' ..
                        + args(1) + op1 + '(' + args(2) + op2 + args(3) + ')]'

            //disp(expr)
            r = evstr(expr)
            //disp(r)

            if r(2) == r(3) then
                p = 0
            elseif r(1) == r(2) then
                p = -1
                return
            elseif r(1) == r(3) then
                p = 1
                return
            else
```



```
                error('could not determine precedence level')
            end
        end
    end
end
```

8.4.3. parser.sci

```
// determine properties of Scilab's parser:
// associativity and precedence level of operators

getf('assoc.sci');
getf('prec.sci');

numop1 = ['+' '-'];
numop2 = ['+' '-' '*' '/' '\' '^' '.*' './' '.\' '.^'];
logop1 = ['~'];
logop2 = ['&' '|'];

// inquire associativity
an = assoc(numop2, 'n');
ab = assoc(logop2, 'b');

// figure out the relative precedence of binary numeric operators
pm2 = [];
for i = numop2
    row = [];
    for j = numop2
        row = [row prec(i, j)];
    end
    pm2 = [pm2; row];
end
[lev, idx] = sort( sum(pm2, 'r') );
lev = lev - min(lev) + 1;               // minimum := 1

nop2 = numop2;
for op = numop1  // mark binary oparators that have a unary twin
    patch = find(op == nop2);
    nop2(patch) = op + '/2';
end

relp2 = [string(lev); nop2(idx); an(idx)]';
```

```
relp1 = [];
for i = numop1
    row = [];
    for j = numop2
        row = [row, prec1(i, j)];
    end
    hop = numop2(find(row > 0.5)); // operators with higher precedence
    minhop = 0;
    for op = hop
        minhop = max( minhop, find(relp2(:, 2) == op) );
    end
    // now minhop is the index of the lowest precedence binary operator
    // that has a higher precedence than the unary operator i, or 0 if
    // there is none
    if minhop == 0
        uop = evstr(relp2(1, 1)) + 1;
    else
        uop = evstr(relp2(minhop, 1)) - 1;
    end
    relp1 = [relp1; [string(uop), i+'/1', 'right']];
end
//relp1

// Merge unary operators into matrix of binary operators
relp = [relp1; relp2];
[dummy, idx] = sort(evstr(relp(:, 1)));
relp(idx, :)
```

8.5. cat.sci

`cat.sci` defines the function `cat` which prints the source of a macro (function) if it is available. The argument-, type-, and size-checking part is used in Example 4-1.

```
function [res] = cat(macname)
// Print definition of function 'macname'
// if it has been loaded via a library.

// AUTHOR
//   Lydia van Dijk
//
```



```
//   Copyright 1999, 2000 by Lydia van Dijk.
//   cat is free software distributed under the terms
//   of the GNU General Public License, version 2.

[nl, nr] = argn(0);
if nr ~= 1 then
    error('Call with: cat(macro_name)');
end
if type(macname) ~= 10 then
    error('Expecting a string, got a ' + typeof(macname));
end
if size(macname, '*') ~= 1 then
    sz = size(macname);
    error('Expecting a scalar, got a ' ..
      + sz(1) + 'x' + sz(2) + ' matrix')
end

[res, err] = evstr(macname);
if err ~= 0 then
    select err
    case 4 then
        disp(macname + ' is undefined.');
        return;
    case 25 then
        disp(macname + ' is a builtin function');
        return;
    else
        error('unexpected error', err);
    end // select err
end // err ~= 0

name = whereis(macname);
//disp('name = <' + name + '>');
if name == [] then
    disp(macname + ' is defined, but its definition is unaccessible');
    clear ans;
    return;
end

cont = string(evstr(name)); // path (1) and contents (2..$) of library
fpath = cont(1);
if part(fpath, 1:4) == 'SCI/' then
    fpath = SCI + '/' + part(fpath, 5:length(fpath));
end
fname = fpath + macname + '.sci';
```

```
[fh, err] = file('open', fname, 'old');
if err ~= 0 then
    error('Could not open file ' + fname, err);
end
text = read(fh, -1, 1, '(a)');
file('close', fh);
write(%io(2), text);
```

8.6. quadpack.sci

Here is the complete example from Section 6.3.3. Function `quadpack` loads, unloads, or queries the load status of a Scilab extension. In this case the extension is the famous QuadPack library.

The foremost goal in the design of `quadpack` was user friendliness. Therefore, we condensed the function's interface to its minimum, providing only three different actions:

`load`

Link library and glue code with Scilab; do nothing if the library/glue code already has been linked. Return the actual linkage status afterwards.

`unload`

Unload library and glue code; do nothing if the library/glue code was not linked with Scilab before. Return the actual linkage status afterwards.

`query`

Do nothing, but return the actual linkage status.

Additional goodies are that `quadpack` defaults to action `query` if the function's argument is omitted, that the case of the argument does not matter, and that a minimal prefix of the argument is enough to select the right action.

```
function state = quadpack(action)
// name:        quadpack.sci  -  load/unload QUADPACK or query
//                                 the load-status
// author:      Lydia van Dijk <lydia_van_dijk@my-deja.com>
// last rev.:   Sat Mar 18 19:23:58 UTC 2000
// Scilab ver.: 2.5

// The variable 'quadpacklibs' is the *only* one that needs
// adjustment on a per-system basis.  It is safe to leave
```



```
// all the other stuff untouched.

quadpacklibs = ['/site/src/netlib/quadpack/libquadpack-1.0.so', ..
                '/site/src/netlib/quadpack/intersci/libqpif-1.0.so']

//
// No user servicable parts below this line.
//

gateway = 'quadpack_gw'  // name of the gateway function

// The order of the interface names in
// interfaces *MUST* be the same
// as in qptab in file 'quadpack-gw.c'!
interfaces = ['intals', 'intcau', 'intexc', 'intfou', ..
              'intgen', 'intgk',  'intinf', 'intosc', ..
              'intsm']

[lhs, rhs] = argn()
if rhs == 0 then
    action = 'query'                                  – Default if no args
end

if rhs >= 2 then
    error('Too many arguments; expecting 0 or 1')
end
if type(action) ~= 10 then // 10 means string
    error('Expecting a string in argument 1')
end
if size(action, '*') ~= 1 then
    error('Expecting a scalar in argument 1')
end

action = code2str( abs(str2code(action)) )            – Convert to lowercase
[state, number] = c_link(gateway)

if strindex('query', action) == 1 then                – Check prefex
    // do nothing
elseif strindex('load', action) == 1 then
    if state == %t then
        disp('already loaded; no action taken')
        return
    end
    addinter(quadpacklibs, gateway, interfaces)
elseif strindex('unload', action) == 1 then
```

```
    if state == %f then
        disp('not loaded; no action taken')
        return
    end
    ulink(number)
else
    error('Expecting "query", "load" or "unload" in arg 1')
end
state = c_link(gateway)                                  – Always return actual status
```

Appendix A. GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the

Appendix A. GNU Free Documentation License

Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is

included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See *GNU Copyleft* (http://www.gnu.org/copyleft/).

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Appendix A. GNU Free Documentation License

Appendix B. GNU General Public License

Version 2, June 1991

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330
Boston, MA 02111-1307
USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU General Public License

Terms And Conditions For Copying, Distribution And Modification

0. APPLICABILITY[1]

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. VERBATIM COPYING

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. MODIFICATIONS

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. DISTRIBUTION

You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

Appendix B. GNU General Public License

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. TERMINATION

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. ACCEPTANCE

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. REDISTRIBUTION

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. CONSEQUENCES

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. LIMITATIONS

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. AGGREGATION WITH INDEPENDENT WORKS

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. NO WARRANTY

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. LIABILITY

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL

ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Notes

1. The titles of the sections have been added by the authors. They do not occur in the original GNU General Public License. Everything else has been copied in verbatim.

Appendix B. GNU General Public License

Bibliography

[cameron:1996] Debra Cameron, Bill Rosenblatt, and Eric Raymond, *Learning GNU Emacs*, second edition, O'Reilly&Associates, 1996, 1-56592-152-6.

[golub:1996] Gene H. Golub and Charles F. van Loan, *Matrix Computations*, third edition, The Johns Hopkins University Press, 1996, 0-8018-5413-X.

[kernighan:1999] Brian W. Kernighan and Rob Pike, *The practice of programming*, Addison Wesley Longman, Inc., 1999, 0-201-61586-X.

[wall:1996] Larry Wall, Tom Christiansen, and Randal L. Schwartz, *Programming Perl*, Second edition, O'Reilly&Associates, 1996, 1-56592-149-6.

Index

Symbols

$ constant, 59
& operator, ?
.
  (See dot)
./ operator
  (See division, element wise)
  (See multiplication, element wise)
: operator, 57, ?
geci , 120
[ ] operator
  (See vector, construction)
| operator
  (See & operator)

A

Ada
  pseudo
    (See pseudo Ada)
Ada extensions
  (See subroutines, external, Ada)
array ordering
  Fortran-77, 79
assigment
  tuple, ?

B

benchmark
  mirror functions, 78

C

C extensions
  (See subroutines, external, C)
C++ extensions
  (See subroutines, external, C++)
calling convention
  by-reference, 79
column-major ordering
  (See array ordering, Fortran-77)
control structures
  block structure, ?
  choice of, ?
  early return, ?
  for, ?
  if, ?
  select, ?
  while, ?
conventions, typographic, ?
CVS, ?

D

Debugging
  Scilab, ?
desc-file
  (See file, interface description)
Dirac distribution, 48
dispatch tables, ?
division
  element wise, 19
documentation
  local, ?
    Comm.ps, 120
    Internals.ps, 120
    Intro.ps, 120
    Lmi.ps, 121
    Manual.ps, 121
    Metanet.ps, 121
    Scicos.ps, 121
    Signal.ps, 121
dot
  decimal, ?

E

environment variables
  SCI, 50
error
  generation
    API routines
      (See Scilab, error handling)
evaluation
  boolean, 41
  short-circuit
    (See evaluation, boolean)

F

fatal error
  (See Scilab, error handling, fatal errors)
Fibonacci function, 34
file
  interface description, 76
find
  (See locate)
formats, other of sci-BOT, ?
Fortran-77 extensions
  (See subroutines, external, Fortran-77)
Fortran-9x extensions
  (See subroutines, external, Fortran-9x)
free mathematical software
  (See mathematical software, free)
free plotting software
  (See plotting software, free)
function
  Dirac
    (See Dirac distribution)
  Fibonacci
    (See Fibonacci function)
  Fortran-77
    name mangling
      (See name mangling, Fortran-77 function)
  mirror and variants, 57
  size of, ?
functions
  API
    error, 105
    out, 105, 106
  as members of tlist, ?
  as parameters, ?
  as variables, ?
  builtin
    addinter, 77
    and, ?
    cumprod
      (See function sum)
    cumsum
      (See function sum)
    diag, ?
    eye, ?
    find, ?
    fort, 76
    freq, 74
    fsolve, 79
    gsort, ?
    horner, 74
    intg, 79
    link, 79, 85
    linspace, ?
    logspace, ?
    macrovar, ?
    matrix, ?
    max, ?
    min
      (See function max)

ones, ?
or
(See function and)
poly, 74
prod
(See function sum)
rand, ?
size, ?
sort
(See function gsort)
sum, 56, ?
zeros, ?
bulletproof, ?
call without parenthesis, ?
gateway
(See dispatch tables)
native
(See Scilab, native functions)
nested, ?
parameters
named, 43
optional, 43
safe
(See functions, bulletproof)
user defined, ?
without parameters, 42
without return value, 42

G

gateway functions
(See dispatch tables)
Glimpse, ?
GNU
Free Documentation License (FDL), 147
General Public License (GPL), 155

H

hyperlinks, ?

I

index
highest
(See $ constant)
last
(See $ constant)
indexing
avoiding, ?
INRIA, 122
interface description file
(See file, interface description)
intersci, 121
example Makefile, 77
program, 76

L

libraries
numerical, 122
ASCI RED, 123
ATLAS, 123
Essential Scientific Software Library (ESSL) , 123
Extended Math Library, 123
Fortran High-Performance Library (PerfLib) , 123
GAMS, 122
Lapack, 122
Math Kernel Library (MKL) , 123
Netlib, 122
optimized BLAS, 123
Scientific Library (SCSL) , 123
links
(See hyperlinks)
locate, ?

M

mathematical software
free, 124
MuPAD, 124
Octave, 124
Pari, 124
Tela, 124
matrix
construction, ?
flattened representation, ?
operations, ?
shaping a
(See functions, builtin matrix)
multiplication
element wise, 56

N

name mangling
Fortran-77 function, 79
native functions
(See Scilab, native functions)
newline
missing last, ?
numbers
decimal
(See dot)

O

operation
vectorized, ?
operator
./
(See division, element wise)
precedence and associativity, ?
logical operators, ?
numeric operators, ?
relational operators, ?
[ ]
(See vector, construction)
Optimizing
Scilab
(See scilex, binaries, building optimized)
overhead
link, 75
runtime, 75

P

Parallel Scilab
home page, 122
parameters
named
(See functions, named parameters)
optional
(See functions, optional parameters)
plotting software
free, 124
GNUPlot, 124
PlotMTV, 125
polynomials
evaluation of, ?
prototyping, 86
pseudo Ada, ?
types, 89

R

RCS, 117

runtime overhead
- (See overhead, runtime)

S

scalar product, 55
SCI (environment variable), 50
Scilab
- API, ?
  - checklhs, 93, 108
  - checkrhs, 93, 107
  - cremat, 93, 116
  - error, 93, 99
  - getexternal, 99, 113
  - getmat, 93, 110
  - getrmat, 111
  - getrvect, 111
  - getscalar, 99, 112
  - getvect, 112
  - lhs, 93, 99, 109
  - rhs, 93, 99, 109
- compiler to Fortran-77, 86
- debugging
  - (See Debugging, Scilab)
  - (See Debugging, Scilab)
- enthusiasts
  - Mottelet, Stéphane, 122
  - Pinçon, Bruno, 122
  - Segre, Enrico, 122
  - Van Dijk, Lydia Ellen, 122
- error handling, ?
  - fatal errors, ?
  - messages, ?
  - warnings, ?
- extending, ?
- FAQ, 122
- FTP site, 122
- home page, 122
- internal data structure, ?
  - complex matrices, ?
  - data stack
    - (See Scilab, internal data structure, stack)
  - parameter stack
    - (See Scilab, internal data structure, stack)
  - stack, ?
- native functions, ?
  - functionals, ?
  - simple, ?

scilab shell script, 50
scilex
- binaries, 119
  - building optimized, 86
  - debugging code, 119
  - optimized code, 119
  - profiling code, 119
- starting, ?

scope
- dynamic, 23
- enclosing, 23
- lexical, 24

session
- restart
  - persistent global variables, 27

sorting
- lexicographical, 69

space
- white
  - (See whitespace)

spacing
- emphasizing brackets, ?
- in an expression, ?
- indentation, ?
- line breaks, ?

starting scilex
- (See scilex, starting)

style
- formatting, ?
- spacing
  - (See style, formatting)

subroutines
- external
  - Ada, ?
  - C, ?
  - C++, ?
  - compiling, ?
  - Fortran-77, ?
  - Fortran-9x, ?

T

tlist
- functions as elements of, 49

tuple
- assigment
  - (See assigment, tuple)

type
- cast
  - implicit, ?
- promotion
  - implicit
    - (See type, cast implicit)

typographic conventions
- (See conventions, typographic)

V

variable
- clearing, ?
  - global, 27, 28
  - local, 27
- global, ?
- local, 25
- scoping
  - local, ?
- shadowing, 22
- visibility rule, 22

variables
- API
  - buf, 105
  - err2, 105

vector
- construction, ?
- generation, ?

W

warnings
- (See Scilab, error handling, warnings)

whitespace
- around dotted operator, 19
- as column separator, 20
- last newline, 22