FUDGIT

A Multi-Purpose
Data-Processing
and
Fitting Program

User's Manual

Version 2.32
May 1993

Martin-Daniel Lacasse
Center for the Physics of Materials
and
Department of Physics
McGill University
Montréal, Québec, Canada
<isaac@physics.mcgill.ca>

© Martin-Daniel Lacasse, 1993

Contents

1 Introduction **1**

1.1 What is FUDGIT ? . . . 1
1.2 Future . . . 2
1.3 Supported Architectures . . . 2
1.4 Bugs . . . 2
1.5 Credits . . . 2

2 Tutorial **5**

2.1 Prerequisite Knowledge . . . 5
2.2 The Different Modes . . . 5
2.2.1 The Fitting Mode . . . 6
2.2.2 The C-calculator Mode . . . 6
2.2.3 The Plotting Mode . . . 6
2.3 Changing from one Mode to Another . . . 6
2.4 Vectors, Parameters, Variables, Strings, and Constants . . . 6
2.4.1 Scope of Variables . . . 7
2.4.2 Vectors . . . 7
2.4.3 Parameters . . . 8
2.4.4 Scalar Variables . . . 8
2.4.5 String Variables . . . 8
2.4.6 Constants and String Constants . . . 9
2.5 C-calculator Mode Essentials . . . 9
2.5.1 Computational Errors . . . 9
2.5.2 Flow Control . . . 10
2.5.3 Functions and Procedures . . . 10
2.5.4 Printing from C-calculator Mode . . . 10
2.5.5 Implicit Loops . . . 11
2.6 Plotting Mode Essentials . . . 11
2.7 Fitting Mode Essentials . . . 11
2.7.1 Environment Variables . . . 11
2.7.2 Script Files . . . 11
2.7.3 Signals . . . 11
2.7.4 Input/Output . . . 12
2.7.5 Variable Expansion . . . 13
2.7.6 Macros and Aliases . . . 13
2.7.7 Flow Control . . . 13
2.7.8 Fourier Transforms . . . 13
2.7.9 Cubic Spline Interpolation . . . 14
2.7.10 Basic Statistics and Integration . . . 14
2.7.11 Fitting Commands . . . 15
2.8 Interactive Command Line Shell . . . 17

2.9 Dynamic Loading 18

3 Reference Manual **19**
3.1 & 19
3.2 \ 19
3.3 ! 19
3.4 ? 20
3.5 $ 20
3.6 _dumplot 21
3.7 _killplot 21
3.8 adjust 21
3.9 alias 22
3.10 append and save 22
3.10.1 append history 22
3.10.2 append macros 23
3.10.3 append parameters 23
3.10.4 append variables 24
3.10.5 append vectors 24
3.11 auto 24
3.12 break 25
3.13 C 25
3.14 cd 26
3.15 cmode 27
3.16 comments 30
3.17 continue 30
3.18 data files 30
3.19 echo 30
3.20 else 31
3.21 end 31
3.22 endif 31
3.23 environment 31
3.24 exec 31
3.25 exit 32
3.26 fft 32
3.27 fit 33
3.28 fmode 33
3.29 for 33
3.30 foreach 34
3.31 free 35
3.32 func 35
3.33 help 36
3.34 history 36
3.35 if 36
3.35.1 C-calculator mode if 37
3.35.2 Fitting mode if 37
3.36 in 38
3.37 install 38
3.38 invfft 41
3.39 let 41
3.40 line editing and history 41
3.41 load 42
3.42 lock 42
3.43 ls 43

3.44 macro . . . 43
3.45 math functions . . . 44
3.45.1 math function abs . . . 44
3.45.2 math function acos . . . 44
3.45.3 math function acosh . . . 44
3.45.4 math function asin . . . 44
3.45.5 math function asinh . . . 44
3.45.6 math function atan . . . 44
3.45.7 math function atan2 . . . 44
3.45.8 math function atanh . . . 44
3.45.9 math function besj0 . . . 44
3.45.10 math function besj1 . . . 45
3.45.11 math function besjn . . . 45
3.45.12 math function besy0 . . . 45
3.45.13 math function besy1 . . . 45
3.45.14 math function besyn . . . 45
3.45.15 math function cbrt . . . 45
3.45.16 math function ceil . . . 45
3.45.17 math function cos . . . 45
3.45.18 math function cosh . . . 45
3.45.19 math function cot . . . 45
3.45.20 math function coth . . . 45
3.45.21 math function csc . . . 45
3.45.22 math function csch . . . 46
3.45.23 math function erf . . . 46
3.45.24 math function erfc . . . 46
3.45.25 math function exp . . . 46
3.45.26 math function floor . . . 46
3.45.27 math function hypot . . . 46
3.45.28 math function int . . . 46
3.45.29 math function interp . . . 46
3.45.30 math function lgamma . . . 46
3.45.31 math function ln . . . 46
3.45.32 math function log . . . 47
3.45.33 math function max . . . 47
3.45.34 math function min . . . 47
3.45.35 math function rand . . . 47
3.45.36 math function rint . . . 47
3.45.37 math function scan . . . 47
3.45.38 math function sec . . . 47
3.45.39 math function sech . . . 47
3.45.40 math function sin . . . 48
3.45.41 math function sinh . . . 48
3.45.42 math function sqrt . . . 48
3.45.43 math function srand . . . 48
3.45.44 math function sum . . . 48
3.45.45 math function tan . . . 48
3.45.46 math function tanh . . . 48
3.45.47 math function trunc . . . 48
3.46 pause . . . 48
3.47 plot . . . 49
3.48 pmode . . . 49

3.49 print . . . 49
3.50 proc . . . 50
3.51 pwd . . . 51
3.52 quit . . . 51
3.53 quotes . . . 52
3.54 read . . . 52
3.55 reinstall . . . 53
3.56 return . . . 53
3.57 save . . . 53
3.58 set . . . 53
3.58.1 set comment . . . 53
3.58.2 set data . . . 53
3.58.3 set debug . . . 54
3.58.4 set error . . . 54
3.58.5 set expand . . . 55
3.58.6 set format . . . 55
3.58.7 set function . . . 55
3.58.8 set input . . . 57
3.58.9 set iteration . . . 57
3.58.10 set method . . . 57
3.58.11 set noexpand . . . 58
3.58.12 set output . . . 58
3.58.13 set pager . . . 58
3.58.14 set parameters . . . 58
3.58.15 set plotting . . . 59
3.58.16 set prompts . . . 59
3.58.17 set samples . . . 59
3.58.18 set vformat . . . 60
3.59 shell . . . 60
3.60 show . . . 60
3.60.1 show comment . . . 60
3.60.2 show data . . . 60
3.60.3 show debug . . . 61
3.60.4 show error . . . 61
3.60.5 show input . . . 61
3.60.6 show iteration . . . 61
3.60.7 show fit . . . 61
3.60.8 show format . . . 62
3.60.9 show function . . . 62
3.60.10 show macros . . . 62
3.60.11 show memory . . . 62
3.60.12 show method . . . 62
3.60.13 show output . . . 63
3.60.14 show pager . . . 63
3.60.15 show parameters . . . 63
3.60.16 show plotting . . . 63
3.60.17 show prompts . . . 63
3.60.18 show samples . . . 64
3.60.19 show setup . . . 64
3.60.20 show table . . . 64
3.60.21 show variables . . . 64
3.60.22 show vectors . . . 65

3.60.23 show vformat . . . 65
3.61 smooth . . . 65
3.62 special . . . 65
3.63 spline . . . 66
3.64 startup . . . 66
3.65 stop . . . 67
3.66 string functions . . . 67
3.66.1 string function DirName . . . 67
3.66.2 string function FileName . . . 67
3.66.3 string function Read . . . 68
3.66.4 string function Scan . . . 69
3.67 system . . . 69
3.68 then . . . 69
3.69 unalias . . . 69
3.70 unlock . . . 70
3.71 unmacro . . . 70
3.72 version . . . 70
3.73 vi . . . 70
3.74 while . . . 70
3.74.1 C-calculator while . . . 71
3.74.2 Fitting mode while . . . 71

4 More Examples **72**
4.1 Example 1 . . . 72
4.2 Example 2 . . . 72
4.3 Example 3 . . . 73
4.4 Example 4 . . . 73
4.5 Example 5 . . . 74

A Using History Interactively **78**
A.1 History Interaction . . . 78
A.1.1 Event Designators . . . 78
A.1.2 Word Designators . . . 78
A.1.3 Modifiers . . . 79

B Command Line Editing **80**
B.1 Introduction to Line Editing . . . 80
B.2 Readline Interaction . . . 80
B.2.1 Readline Bare Essentials . . . 80
B.2.2 Readline Movement Commands . . . 81
B.2.3 Readline Killing Commands . . . 81
B.2.4 Readline Arguments . . . 82
B.3 Readline Init File . . . 82
B.3.1 Readline Init Syntax . . . 82
B.3.2 Readline Vi Mode . . . 84

Chapter 1

Introduction

1.1 What is FUDGIT ?

No more *fudging*! Despite its name, FUDGIT is a double-precision multi-purpose fitting program. It can manipulate complete columns of numbers using vector arithmetic. FUDGIT is also an expression language interpreter understanding most of C grammar. It supports most functions from the C math library. Finally, FUDGIT is a front end for any plotting program supporting commands from stdin. It is a nice mathematical complement to GNUPLOT, for example.

The main features of FUDGIT are:

- Command shell including history;
- Possible abbreviation of all the *fitting mode* commands;
- Possible plural when it makes sense too;
- User-definable macros;
- Aliases;
- Shell flow control statements such as if, else, while, foreach;
- On-line help;
- On-line selectable plotting program;
- Fourier transforms;
- Cubic spline interpolation;
- Double-precision built-in calculator;
- Built-in interpreter supporting most of C language including flow control (if, else, while, for, break, continue);
- User-definable functions and procedures;
- User-defined objects dynamically linkable as functions or procedures;
- Double-precision vector arithmetic;
- Access to the complete C math library;
- Built-in fitting series such as:

+ power series (polynomial);
+ sine series;
+ cosine series;
+ Legendre polynomials;
+ series of Gaussians;
+ series of exponentials;

- User-definable fitting functions;
- Totally dynamical allocation of variables and parameters;
- Possible selection of fitting ranges.

FUDGIT has a collection of fitting routines including:

- straight line (linear) least squares;

- straight line (linear) least absolute deviation;
- general linear least squares using QR decomposition;
- general linear least squares using singular value decomposition;
- nonlinear Marquardt-Levenberg method.

1.2 Future

FUDGIT can be easily enlarged to include other built-in univariable manipulations such as fancy integration, derivative, statistics, etc. However, dynamical linking allows the user a lot of flexibility. Dynamical linking should be ported to other vendors.

If anyone is interested in going in this direction then be my guest, or send me the routines you want to see included! The interpreter is built so that the inclusion of extra modules is fairly straightforward.

On the other hand, the dynamical loader should allow FUDGIT to become a nice interface between the user, the data files, and mathematical routines. An implementation of matrix arithmetic would however be required. This might happen one day, if time permits... Users would then only have to build (by constructing and/or gathering routines) a tool box that would be totally configurable and reusable. FUDGIT would then be stripped from spline/fitting/fft/etc. routines which would come in a separate tool box. Anyone interested?

1.3 Supported Architectures

As it stands now, FUDGIT can be compiled on AIX, DATA GENERAL, HPUX, linux, IRIX, NeXT, OSF, SUNOS, and ULTRIX. Ports can be easily made to other vendors. As it stands, the dynamic loading feature only compiles on IRIX, ULTRIX (vax only), and SUNOS. Ports of dynamic loading to HPUX and AIX are not foreseen in a near future unless someone indicates me how to implement dynamic loading on these operating systems. If you know something about this let me know. Dynamic loading should be part of new operating systems in a near future: stay tuned.

1.4 Bugs

There are probably too many to enumerate. However, the program was fairly stable on a lot of testing and routine applications. Be careful with possible circular aliases; busting a vector by unlocking `data`. Also, some malloc() libraries are not always happy when they work a lot. As with most interpreters (e.g. csh), `if, else, endif` statements can be difficult to debug and are not much flexible. However, FUDGIT's flow control error messages are more indicative than the one from *csh* and *sh*.

The program has been written with robustness in mind and not optimization. Some parts of the code could therefore be improved for speed, but a deliberate choice has been made on robustness. (As an example, the default behavior is to have all divisions checked for a null quotient.)

Character by character terminal mode (cbreak) is not available through the plotting program pipe. A (less portable) pseudo terminal could be used for the plotting program.

The stack and machine could probably be allocated dynamically. However, recursive loops would make the program to be killed by init for envading the swap space.

Redefined functions do not free the part they were previously occupying on the machine space, but this memory leak is not significant over a normal session.

1.5 Credits

Parts of FUDGIT were built from some existing facilities. I would like to give full credits for ideas or even segments of code that have been taken from the following sources.

- For parts of the user interface:

The help facility and the line editor were taken and adapted respectively from GNUPLOT, and READLINE. READLINE was written by Brian Fox. The help facility is originally from John D. Johnson.

• For the C-calculator:

The calculator is inspired from HOC calculator which was debugged and largely augmented to support vector algebra, memory management and extra operators. The source of the basic program is reproduced for educational purposes in *The Unix Programming Environment* by Brian W. Kernighan and Rob Pike, Prentice-Hall (1984).

• For the fitting functions:

Some of the included fitting routines are based on the algorithms found in chapter 14 of *Numerical Recipes in C* by W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vettering, Cambridge University Press (1988), of which some were in turn adapted from LINPACK. I had to adapt all the algorithms to include elegant error recovery and to perform all calculations on vectors outside the fitting loops, since their implementation would not permit the use of run-time user selectable functions. FUDGIT would not have been possible without the valuable help of that book. Since I strongly recommand that you have a copy of a fitting book, I strongly suggest you have a copy of this one. Not only this book will describe all the methods used in FUDGIT but it will also give you unvaluable insights to get to the state of the art of fitting. These routines are copyrighted and cannot be separated from FUDGIT.

Copyright (C) 1987, 1988 Numerical Recipes Software. Reproduced by permission from the book *Numerical Recipes: The Art of Scientific Computing* published by Cambridge University Press.

• For the fft routine:

The fft routine was first derived from an original Pascal version written in *Simple Calculations with Complex Numbers* by David Clark in DDJ 10/84 and then translated to C by R. Hellman (02/21/86). I rewrote the C version to use vectors of alternated double real and imaginary instead of the original (slow) vector of pointers to complex structures. I also merged all functions in one to prevent unnecessary function calls. I was astonished to see that the resulting version was almost identical to the one found in Numerical Recipes with the exception of a trigonometric recursion. In conclusion, given an algorithm, I think that is is a hard task to try to write a code much better than the one found there. This is normal since the space of code possibilities gets narrower as the constraints (optimization) acting on an implementation in a given language are increased. The one included is from N.R. which was adapted from N. Brenner.

Copyright (C) 1987, 1988 Numerical Recipes Software. Reproduced by permission from the book *Numerical Recipes: The Art of Scientific Computing* published by Cambridge University Press.

• For the help file:

I would like to thank Ross Thompson for proofreading part of the documentation and also for giving me constructive feedback in course of the program development.

• For the IRIX supported dynamic loading package:

The `dl` Dynamic Loading package is from Jack Jansen <Jack.Jansen@cwi.nl> from the *Centrum voor Wiskunde en Informatica*. This package only works on IRIX for now.

• For the SUNOS, ULTRIX supported dynamic loading package:

The implementation of `dl` for SUNOS and ULTRIX is from Guido van Rossum <guido@cwi.nl> from the *Centrum voor Wiskunde en Informatica*. I modified part of it to allow multiple routine loading from the same object. The other part of the puzzle is the `dld` loader from Wilson Ho <how@cs.ucdavis.edu> which is based on GNU ld(1) and is under GNU license.

• For compilers not having `alloca()`:

The included public domain version of alloca is from D. A. Gwyn.

• For systems not having `putenv()`:

The version of putenv was adapted from Dave Taylor's elm who adapted it from cnews.

• For a lot of ideas:

Many thanks to Steve Hornes for stimulating e-mail discussions. Steve is responsible for the idea of implementing dynamic loading in the final development of FUDGIT.

• For the port to linux:

The port to linux was made by Thomas Kroenig <ig25@rz.uni-karlsruhe.de>. Thanks a lot Thomas!

• For the rest of the code:

Copyright (C) 1993 Martin-D. Lacasse

See the Copyrights file for more detail, or the 'README' help topic.

Permission to use, copy, and distribute this software and its documentation for any peaceful purpose and without fee is hereby granted, provided that the above notices appear in all copies and that both those notices and this permission notice appear in supporting documentation. No part of this can be used for commercial purposes.

Send bugs, comments or suggestions to
<isaac@physics.mcgill.ca>.

Disclaimer:

This software is provided *as is* without express or implied warranty.

Chapter 2

Tutorial

The following sections will deal with several applications of FUDGIT. The main strength of FUDGIT resides in its capacity of fitting and splitting multiple files. This kind of applications cannot be done by a mouse driven program since commands can rarely be automated in such programs. FUDGIT allows the user to build scripts which can automate a process that performs the available functions or user-defined functions or macros over several data files. The syntax is very close to C-shell and most of C-shell users will only have to intuitively apply their knowledge to the current fitting mode shell. Parallel to this shell is a built-in calculator which in fact is a basic C interpreter supporting most of the C math library, plus some additional features. The particularity of FUDGIT comes from the fact that global variables are common to both the C-calculator and the command shell. Once again, a minimum knowledge of C will suffice to write short programs using the C-calculator.

To make a rough picture, one can say that FUDGIT uses C-shell syntax to deal with files, C syntax to deal with vectors, numbers and strings, and the language of your favorite plotting program for plotting.

2.1 Prerequisite Knowledge

To read this manual, the reader should have a minimal knowledge of C programming. Specifically, the user should understand the basics of `if`, `else`, `while` and `for` constructions. A basic knowledge of C operators would also be preferable.

A familiarity with UNIX *csh* command would be helpful, particularly if you know the C-shell `if`, `while` and `foreach` constructions, as well as aliases, and string variable substitutions. Users already familiar with (and perhaps addicted to) *tcsh*- or *bash*-EMACS- style line editing will feel at home.

However, the novice reader should only refer to a C book to understand operators specific to C. All the rest of what should be known is fully covered by this manual.

It is also understood that you already know the plotting program you are going to link to FUDGIT.

2.2 The Different Modes

FUDGIT is composed of three different modes. These modes can be thought of as (1) a C-shell like interpreter linked with (2) a C-calculator, sharing the same variables in memory, and with (3) a plotting program of your choice.

Most of the time, commands of different modes will be interlaced as they are given on the command line. However, the present manual starts by giving a separate description of the different modes. In the reference manual, we assume that the different modes are well understood so that the commands are described without distinction, as they would be used, and independently of their originating mode.

2.2.1 The Fitting Mode

The C-shell like interpreter is called the *fitting mode*. It is the central mode and is the one from which all accesses to disk are done. This mode has a range of commands allowing the user to read vectors from or save vectors to a data file, to read a command script, save the command history, do the Fourier transform of a vector, make a linear or nonlinear least square fit, etc...This mode also allows the user to define macros and aliases and to perform basic flow control over the statements. All the commands in the fitting mode can be abbreviated. It is worth mentioning that in the fitting mode, the command line parsing is done by analyzing words separated by blanks (spaces or tabs), as in an interactive csh.

2.2.2 The C-calculator Mode

The *C mode* or *C-calculator mode* is a language interpreter supporting most of C grammar except pointers. It also supports most of the double-precision C math library. Thus, recognized keywords cannot be abbreviated, and the different tokens need not be separated. Most of the C operators and keywords are understood and a few extra operators have been added. This mode does essentially all the possible calculations on scalar variables and vectors. Recursive functions and procedures can be defined as well. Since all numeric variables are double precision numbers, C `switch` constructions were not implemented.

2.2.3 The Plotting Mode

The *plotting mode* is a channel talking directly to the plotting program of your choice. Therefore, FUDGIT can serve as a front end to any plotting program able to accept input from stdin. This way, vectors can be built from the C-calculator and then plotted by your favorite plotting program (e.g. super mongo, irisplot, SGIPLOT, plot, GNUPLOT, etc...). The default plotting program is GNUPLOT.

2.3 Changing from one Mode to Another

The fitting mode is the central mode from which the two other modes are accessed. Thus, one cannot change from the C-calculator mode to the plotting mode without going through the fitting mode. The mode changing commands are mnemonically called `cmode` for changing to the C-calculator mode, `pmode` for going to the plotting mode and `fmode` to go back to the fitting mode from any of the two previous modes. In interactive use, typing ^D is equivalent to `fmode` and the program will return to the fitting mode.

Each mode has a different type of parsing. The fitting mode analyzes the input line by breaking it in blank separated words, as the standard C shell. Commands can thus be abbreviated in this mode. On the other hand, the C-calculator mode parses the input line by breaking it into tokens and keywords of different types as does a C or FORTRAN compiler. Nothing can be abbreviated, but the different tokens need not be separated by white spaces. Finally, the plotting mode does no parsing at all, since the plotting program is responsible for analyzing the input line.

The commands `cmode` and `pmode` can be followed by a list of commands on the same line, in which case the rest of the line will be parsed according to the mode the command refers to, and then processed in the mode specified. To make life easier, the `let` command is equivalent to `cmode` and anything following it will be passed to the C-calculator mode parser and then processed.

2.4 Vectors, Parameters, Variables, Strings, and Constants

This section describes the different types of variables supported by FUDGIT. They are **vectors**, **variables**, **constants**, **string variables** and **string constants**. To this list, we should add the last class of objects consisting of only one member: **parameters**. Vector and parameter names consist of upper case letters uniquely. Variable and constant names consist of lower case letters uniquely. String variable and string constant names consist of a mixture of upper case and lower case letters. Any number of digits or underscores

can be part of a name: the remaining characters will then decide on the class of object the name describes. Vectors, parameters, variables and constants are all double precision numbers.

2.4.1 Scope of Variables

All variables are allocated as they are introduced on the C-calculator mode command line. All such variables have a global scope so that they can be accessed from anywhere outside or inside functions or procedures. Automatic variables can be defined with the `auto` keyword. The scope of such variables is delimited by braces as in standard C. The `auto` definition has to be immediately after an opened brace. When defined inside functions or procedures, automatic variables cannot have a name already used by an argument or the function or procedure. Automatic variables are located on the stack and are freed when the brace matching the end of the scope is encountered.

2.4.2 Vectors

FUDGIT supports full vector arithmetic. This means that once a vector has been read, it can be modified using any of the supported math functions. A vector can also be created by a simple assignment or a `while` or `for` loop over its elements. Any vector part of an assignment will be created and allocated if it does not already exist. The allocation size is specified by the `set samples` command. The default size is 4000. A constant called `data` will be set to the number of elements that have been read. Obviously, `data` will always be smaller than the sample size. Operations relating vectors are possible. In this case, a loop over the vector elements will be done from the first element to the `data`th element. The value of this constant is therefore crucial to all vector operations. We should go over an example.

Suppose that vectors X, Y, Z and $ENERGY$ have been read from a file using a `read` statement (described below). Now consider the following assignments:

```
let X=ln(X)
let Y=sin(X+Z)
let TIME=ENERGY^2
let CONST=3
let GAME=rand()
let K/=GAME
```

which all are examples of a single equation relating `data` equations, since each vector element will be assigned individually. Although all the elements of vector `CONST` will be equal to 3, the elements of vector `GAME` will be random numbers independently generated by the function `rand()`.

Vector elements can also be referenced individually using the usual C syntax. Thus, $A[2]$ refers to the second element of vector A. Vector operation can thus be done using `for` loops or `while` loop constructions. Note that unlike C, the first element of any vector is 1.

Be careful with assignments involving a vector element and the same whole vector as in:

```
let VEC *= VEC[3]
```

which would multiply the vector `VEC` by its third element. However, the value `VEC[3]` will not be constant over the implied loop and the result will not be the one expected. Therefore, such assignments should be written using temporary variables as in:

```
let tmp = VEC[3]
let VEC *= tmp
```

so that variable `tmp` carries the value that `VEC[3]` had before the assignment through the whole loop.

Vectors can be saved, appended to a file, with the `append vectors` and `save vectors` command. They can be displayed at any time using the `show vectors` command.

2.4.3 Parameters

The short vector containing the fitting parameters is isolated in a class by itself. Parameters can be assigned globally or element by element. Clearly, parameters cannot be mixed with vectors but elements of it can. Therefore, statements like

```
set parameters FITPAR 2
let FITPAR = 3
let VECTOR = 3
let FITPAR[2] = 3*sin(pi) * VECTOR[4]
```

all are examples of the possible declaration and usage of parameters.

As with vectors, parameters can be displayed, saved, appended using the same range of commands **append**, **save**, **show parameters**.

2.4.4 Scalar Variables

Scalar variables have lower case names and are allocated as they are introduced on the command line. For example,

```
let x++
let y=Y[4]
let z=x*y
```

are all legal commands. Note that the first example increments variable x only if it already exists. In terms of assignment, a vector or parameter element is considered as a variable. Variables can also be mixed with vectors such as in

```
let X=3^x
let Y=x+Z
let N=n++
let Z=cos(X/a + Z[x])
```

but statements as

```
let x=X+Z
```

are clearly not legal.

Variables can be saved, appended to a file, with the **append variables** and **save variables** commands. They can be displayed at any time with the **show variables** command.

2.4.5 String Variables

String variables can also be created as needed. Any name consisting of both lower case and upper case letters will be considered as a string variable. Statements like

```
let String = "Hello"
let Other_String = String
let String1 = "Good"
let String2 = " Bye!"
let String3 = String1 + String2
let String4 = String1 - "od"
let Input = Read()
let y = scan(Read(), "%lf")
let x = (Other_string == String)
let y = (Other_string != "What is going on?  \n")
```

are thus permitted. The only operations permitted on string variables are the equality operators, the addition operator, the subtraction operator, and to be arguments of string functions such as **scan** function. In the previous example, x and y will both be set to 1. Note that the **Read** function returns a string.

2.4.6 Constants and String Constants

A constant is a variable that is locked to its value. It can be used at the same place a variable can be except that changing its value will result in a parsing error. FUDGIT contains 4 built-in scalar constants: `pi` = π, `e` = Neperian number e, and `chi2` which contains the value of χ^2 as obtained from the last fit. Finally, `data` contains the effective size of all vectors.

There are also 3 built-in string constants: `ReadFile` holding the name of the last file read by the `read` command, `Tmp` holding a temporary filename consisting of */tmp/fudgitPID*, where *PID* holds for a unique number describing the current process, and finally `Cwd` containing the name of the current working directory. Constants and variables (either scalars or strings) can be transformed in one another using `lock` and `unlock` commands so that the user can build his own constants at any time.

The linked list of all vectors, parameters, variables, constants, functions and procedures can be displayed by the `show table` command.

2.5 C-calculator Mode Essentials

The C-calculator mode is the mode dealing with the mathematical transformations of the variables. It has access to the C functions of the mathematical library such as $\sin(x), \cos(x), \tan(x), \ln(x), \log(x), \exp(x)$ and more fancy functions such as random number generators, Bessel functions of the first and second kind, error function, Gamma function and so on...

All operators of C are supported with the exception of pointer and bitwise operators. An extra ^ exponentiation operator has been implemented.

2.5.1 Computational Errors

All mathematical functions, exponentiation operator and divisions are checked for error. Computational errors involve infinity values, not a number, out of range, and out of domain. The user can select what kind of checks she or he wants to have. The first two are values whereas the last two are error numbers set by the various mathematical functions. This is done using the `set error` command. The user has the choice among the following checks:

Infinity A value of that kind results from a division by zero or if the value resulting from a mathematical function cannot be represented by the computer. The number representation limit is 1.0e+308 for most 64 bits double machines. To give an idea, the exponential e^{710} cannot be represented. Note that the infinity value is signed on some machines, e.g, $\log(0) = -\infty$.

Not a Number This value is returned by a function called with an argument out of the domain, such as a negative number to a log() of a number smaller that 1 for acosh(). This value is also returned when trying to divide 0 by itself, or the Infinity by itself.

Out of Range This is a flag (error number) that is raised by the mathematical functions returning a value that cannot be represented by a double C type (double precision FORTRAN).

Out of Domain This flag is raised whenever the argument is unacceptable for a given mathematical function. Typical examples are $x < 0$ for $\log(x), \ln(x), x^r$ for any r, $x \notin [-1, 1]$ for $\mathrm{acos}(x)$, $\mathrm{asin}(x)$, etc.

By setting the error to some of these, the user keeps control over the whole calculation. If a computational error occurs in a loop, the element number will be indicated along with the error message. The default is to have all error checks active.

2.5.2 Flow Control

The C-calculator mode supports the `if-else`, `while`, and `for` constructions. The syntax is very similar to the one in C. They can be embedded one in another, with no other limit that the compiled code does not blow FUDGIT's internal machine capacity.

The basic C-calculator mode `if` syntax is

```
if (conditions) {
cmode-statements
} else if (conditions) {
cmode-statements
} else {
cmode-statements
}
```

or, for single line statements, or semicolon separated list of statements typed on the same line, the following syntax

```
if (conditions)
cmode-line-statement
else
cmode-line-statement
```

is perfectly legal. The same C type of construction is valid for both `for` and `while`. Note that semicolons are separators and not terminators as they are in C. Therefore, empty statements are defined by empty braces { }.

2.5.3 Functions and Procedures

FUDGIT's C-calculator supports recursive functions and procedures. Therefore, more complex functions can be defined using the mathematical functions provided by the C library. I refer the reader to the `func` and `proc` items of the reference manual where some examples are given. These commands are only accessible in the C-calculator mode. Programming in FUDGIT's C-calculator mode is very close to programming a simple mathematical application in standard C. All variable types can be passed as arguments. Scalar variables are passed by value whereas vectors and string variables are passed by pointer.

2.5.4 Printing from C-calculator Mode

Variables and constants (either scalars or strings) can be displayed by typing their name on the command line. A printing list is a coma separated list of variables. Thus

```
cmode
"values", x, y, X[3]
```

will display the values of string "values", x, y, and X[3], tab separated and appended with a newline, to the standard output. The `print` command is a bit different. It will print the given printing list *without an appended newline* to a file selected by the fitting mode `set output` command. The value of `output` is *stdout* by default. It can be set to any filename or to either of the strings `stdout` and `stderr`. The following fragment of code will give an output similar to the ones obtained above:

```
fmode
set output stdout
cmode
print "values", x, y, z, "\n"
fmode
```

2.5.5 Implicit Loops

All vector or parameter assignments imply a loop over the vector or parameter elements (the loop is from 1 to `data`). In such case, a vector name can be used in place of an expression, and the relation will run over all elements, from one to `data`, the latter representing the effective size of vectors. This is also valid for user-defined functions (see `func`).

2.6 Plotting Mode Essentials

The *plotting mode* has no real command as such. It is a pipe connected to the stdin of a plotting program of your choice. Note that all $*var-name* construction are still recognized and expanded as you talk to the plotting program. Moreover, all the interactive features of FUDGIT, such as filename completion and command line history are still active. Even if you don't need all the features of FUDGIT, it can still be used as a front end to your plotting program, especially if you get addicted to interactive command shells having filename completion and variable substitution mechanism.

2.7 Fitting Mode Essentials

We shall now describe the central mode of FUDGIT: the *fitting mode*. The fitting mode allows the user to define `macros`, to `alias` commands, and to have flow control statements as `while`, `foreach` loops and `if`, `else` similar to those in C-shell. These features allow the user to deal with several files and collect information from fitting successive files and putting the results in a unique output file. This feature is essential when one has to do a fit on the results obtained from several other fits. This is what FUDGIT was written for. All the fitting routines, FFT routines, and I/O accesses are done from the fitting mode.

This section gives a brief description of the range of commands and operations accessible to the fitting mode.

2.7.1 Environment Variables

The fitting mode is sensitive to the following environment variables:

PAGER This variable refers to the pager to use when displaying vectors of size larger than 24 of while interactively displaying long output.

SHELL The value of this environment variable is used whenever the user opens a shell from fudgit.

HOME Your home variable is used to translate the '~' symbol and when `cd` command is called without argument.

FUDGIT also loads a file called *~/.fudgitrc* every time it starts. This is useful to configure the program according to your needs.

2.7.2 Script Files

The `load` command is one of the most handy command. All the commands are generally put in a file with the editor and then loaded in FUDGIT using the `load` command. Thus, complicated commands can be recalled and modified at will. In order to avoid confusion between your data files and the script files you write, we recommand that you use the *.ft* extension for your script files.

2.7.3 Signals

FUDGIT responds to signals in a context sensitive way. This discussion will be restricted to signals sent from the keyboard in interactive mode since FUDGIT will behave normally when in non-interactive mode, i.e., when it has been called with some input file names on the shell command line input.

An *Interrupt* signal will be sent by the key combination `Control` and `C` what we will denote `^C`. An *Interrupt* will abort any calculation or loop while in the fitting mode or the C-calculator mode. When in the plotting mode, the *Interrupt* signal will be sent to the plotting program and ignored by FUDGIT. The behavior of the plotting program will depend from one program to another. In GNUPLOT, the current action is interrupted and the program waits for new input.

The key combination `^Z` will suspend the program in its current state. Even if this is done in the middle of an input line, FUDGIT will remember where you were and redisplay the line when the program will be called back in foreground. This is done (only available in *csh* job control) by typing `fg` at your *csh* command line input.

In any mode, a *Quit* signal will force FUDGIT to exit gracefully. This signal is generated by the key combination `^\`. This way of exiting is left for situations where nothing else works.

2.7.4 Input/Output

The Input/Output is done exclusively from the fitting mode. The commands `read` and `exec` read respectively from a file and a program and assign a given column to a vector. For example, the command

```
read file3 X:1 Y:3
```

will assign the first column of file `file3` to a vector named X and the third column to a vector named Y. As we have seen before, the size of vectors is stored in a constant named `data`. The only requirement on the data file is to have a constant number of columns separated by any number of blanks (space or tab). By default, anything following a '#' will be ignored. Each call to `read` or `exec` will set the constant `data` to the number of elements read in the specified vectors. The usage of `exec` is quite similar to `read`. For example, the following line will run program `shortrun 32` which generates output to stdout in different columns:

```
exec "shortrun 32" TIME:1 ENERGY:2
```

The first column is read in vector `TIME` and the second in a vector named `ENERGY`. Note that quotes are necessary to link `shortrun 32` as one argument. A range of input can be specified using a range specifier after the column number. A range specifier is built with the syntax [*lower-bound*: *upper-bound*]. Any bound can be replaced by the '*' wild card which means no bound. A range specifier can be put on any column while reading. Thus,

```
read fort.8 A:2[0:*] B:1[*:10]
```

will read all line elements from file `fort.8` such that the second column is ≥ 0 and that the first column is ≤ 10. The resulting elements will be put in vectors A and B respectively only if both conditions are satisfied. Note from the previous example that the column ordering is not important. As we have seen above, vector names must uniquely consist of capital letters ('_' is also legal) and digits in order to be recognized as such. A line range can also be specified using the syntax. {*lower-bound*: *upper-bound*}. For example,

```
read file2 VALUE:1{100:243}
```

will read any valid line between line 100 and line 243 of file *file2*. The line range can be given to any variable, but the last range given will be the only one active. Line range can be mixed with value range like in what follows:

```
read fort.8 A:2[0:*] B:1[*:10]{*:1024}
```

which will do the same thing as before except that reading is now restricted to the first 1024 lines of the file.

Vectors can be saved or appended to files with the `save and append` commands. The syntax is straightforward:

```
save vectors X Y Z newfile
```

or

```
append vectors TIME ENERGY file.2
```

will save or append the specified vectors to the files specified.

2.7.5 Variable Expansion

All the variables and constants (either scalars or strings) defined in C-calculator mode are available to the interactive fitting mode shell. The $ operator will fully expand a string variable or string constant, i.e, by using the $*var-name* or ${*var-name*} syntax.

Quite similarly, if the variable or constant is a scalar, then the value of the variable will be expanded on the command line (as a string) according to a pre-selected format. The default format is "%.3g" which is reasonable for including in filenames. This format can be changed using the `set format` command.

In the fitting mode, expansion will not take place if the '$' operator is following a '\' or if is located between single quotes.

Note that this variable expansion feature is available in all modes. To avoid the variable expansion in other modes, a '\' has to protect the '$'.

2.7.6 Macros and Aliases

The user can design his own environment by defining macros and aliases. An alias is a command (first word of the line) that gets expanded and is interpreted along with the other words of the line. A macro is a user-defined procedure that requires a certain number of arguments to be used in a series of commands. The number of argument is specified in order to avoid run-time errors. Arguments are referred to using the $*arg-number* syntax. Macros and aliases are only definable in and recognized by the fitting mode.

Aliases follow the C-shell syntax. Defining macros is a bit different and I refer the user to the reference manual found in the next chapter, where some examples are given along with the details. Keep in mind that macros and aliases are argument manipulation devices: arguments are all processed as strings whatever they represent (numbers, variable names, filename, etc.).

2.7.7 Flow Control

The fitting mode supports the three following flow control constructions:

```
foreach Var-name in Unix-commands
fmode-statements
end


if (conditions) then
fmode-statements
else if (conditions) then
fmode-statements
else
fmode-statements
endif


while (conditions)
fmode-statements
end
```

Their grammar is identical to the one in C-shell apart from the fact that `foreach` only expands UNIX commands, and that the *conditions* of `if` and `while` are C-calculator mode expressions. Thus the '$' operator is not required and the whole set of functions of the C mathematical library is available to the user.

2.7.8 Fourier Transforms

One dimensional Fourier transforms can be done directly from the fitting mode command line. Although fast Fourier transforms require a number of data points equal to a power of 2, the user can still write scripts

that will pad the vectors with zeros, up to the nearest power of 2. Note that this process involves an approximation in the resulting vectors. Fourier transforms require the real and the imaginary part. Real Fourier transforms can be done simply by supplying a null imaginary part. The user can use the full power of the C-calculator mode to implement her preferred windowing functions. Full details of indexing can be found in the reference manual under items `fft` and `invfft`.

FUDGIT also includes a smoothing command based on a loop of Fourier transforms using a cut-off frequency. For more detail, read the description of `smooth` command in the reference manual.

2.7.9 Cubic Spline Interpolation

FUDGIT allows the user to interpolate a value given a set of relations $y_i = f(x_i)$. This can be useful to get an estimate of the integral of a relation for which only a few points were obtained. Interpolation consists in two steps: an initialization process `spline` called with the two vectors characterizing the relation (e.g., `spline X Y`), and one or successive calls to C-calculator mode function `interp(x)` returning the interpolated value.

An example is presented to make it clearer.

```
# Read functional relation
# Say file "relfile" contains 24 data points
# ranging from x=2 to x=12
read relfile X:1 Y:2
# Initialize spline routine
spline X Y
# Build a curve having 120 data points from this set
set data 120
# Build new X vector ranging [0, 1]
let x=0; X=x++; tmp=data-1; X/=tmp
#  Map to X ranging [2, 12]
let X = 10*X + 2
# Build new Y vector including original set of 24 data points
let Y = interp(X)
```

The previous example shows a *natural cubic spline* for which the *second* derivative of the interpolating curve is such that it vanishes at the first and the last data points of the original set. The `spline` command accepts optional third and fourth arguments specifying the value of the *first* derivative of the interpolating curve at, respectively, the first and last data points of the original set.

See the reference manual for more details.

2.7.10 Basic Statistics and Integration

As it stands now, FUDGIT does not support fancy statistics. However, the user can write his own scripts, procedures and functions to deal with basic statistics. The `sum` built-in math function provides a handy way of calculating the mean, and the standard deviation (and variance). Thus, given a vector X,

```
let X2 = X*X                # define a vector containing the square of X
let mean = sum(X)/data      # get the average
let VAR = (X-mean)^2        # VAR contains the square of the difference
let var = sum(VAR)/(data-1) # the variance as such
let sigma = sqrt(var)       # the standard deviation
```

When used in conjunction with `spline` and `interp`, the `sum` command can also be used to perform basic numerical integration. Given a data set X representing sample points of a smooth function, basic numerical integration can be performed by applying a `spline-interp` procedure to build a denser vector $NEWX$. The interpolation should be made such that increasing `data` would not change the value of `sum(NEWX)/data`. Other basic integration techniques can also be computed directly using the C-calculator

programming language. See the reference manual for a more detailed description of each of the commands mentioned above.

2.7.11 Fitting Commands

We are now ready to perform a fit. We shall go over some examples, slowly introducing features of FUDGIT. In the following example, we shall fit $y = ax^b$ by using the log-log representation and fitting a straight line by a least-square method. Refer to the reference manual in the next chapter to understand the commands as they as introduced.

```
# We fit a straight line
set function straight
# using least square linear regression
set method ls_reg
# with 2 parameters called, say, B[1] and B[2]
set parameters B 2
# Read elements having positive X and Y values from file fort.32
read fort.32 X:1[0.00001:*] Y:2[0.00001:*] DY:4
# Take the ln()
let DY /= Y
let Y = ln(Y)
let X = ln(X)
# do the fit
fit X Y DY
# Save the result in file myfit
save parameters myfit
```

All these commands could be typed interactively (without the comments) or put into a file (with the comments) and then `load`ed. The file could also be given as an argument to FUDGIT by typing

```
% fudgit scriptfile.ft
```

from your C-shell. Note that the extension *.ft* is not necessary but is strongly recommanded.

In the following example, we shall fit two Gaussians using a nonlinear Marquardt-Levenberg method. Nonlinear fitting is generally done interactively since the parameter space is explored so as to minimize the value of χ^2. Some local minima can be reached in which case the values of the parameters will not be representative. Nonlinear fitting is an art!

```
set function gauss          # We fit two Gaussians
set method ml_fit           # using Marquardt-Levenberg
set parameters VAL 6        # We need 6 parameters called, say, VAL
read out2 T:1 E:2 ERROR:3   # Read elements from file out2
# Nonlinear fits is an art!
adjust 1 2 3 4 5 6  # Adjust all 6 parameters
# Initialize some parameters,
# since 'set parameters' created vector VAL = 0  by default
let VAL[1] = 1
let VAL[2] = -10
let VAL[3] = 3
let VAL[4] = 8
let VAL[5] = -2
let VAL[6] = 0
fit X Y DY                  # do the fit
# You might need to use less variables in your fit
```

```
adjust 1 5
# Initialize again
let VAL[1] = 2
.
.
.
fit X Y DY
# Are you satisfied of CHI^2?
# show parameters to see curvature matrix and correlations
show parameters
# save parameters in a file along with respective errors
save parameters myfit
```

Finally, we shall see an example involving a user-defined function. This example shows how multiple fits can be appended in a file. It also shows the use of macros.

```
# Make a series of fits from different directories
# Each directory bears a name representing a temperature value
# Tell GNUPLOT to not put a key
pmode set nokey
# The 2-d Ising critical temperature is tc
let tc = -1.0/(0.5*ln(sqrt(2)-1))
# Define the central macro
# Call this macro fitscript and it takes one argument
macro fitscript 1
    # Save the temperature value as a variable
    let temp=$1
    # Convert it since it was in units of tc
    let tempc = temp*tc
    # Compute the Onsager surface tension for this temperature
    let s = 2 *(1+(tempc/2)*ln(tanh(1/tempc)))
    # Read the data from this specific directory
    read $1/landm.0 T:1[20:*] U:2
    # Convert vector time
    let SQT = 1/sqrt(T)
    # Fit, the function and method are defined below.
    # Don't worry: we are defining a macro first.
    fit T U DU
    # Save the transformed and fitted vectors in the same directory
    # for subsequent plotting
    save vec SQT U UFIT $1/landm.0.newfit
    # Append the fitted parameters in file common to all fits
    # as well as the temperature value in the first column
    append var temp A[1] DA[1] A[2] DA[2] A[3] DA[3] newpar.0
    # Go in plotting mode to talk to GNUPLOT directly.
    pmode
    plot '$1/landm.0.newfit' using 1:2,\
    '$1/landm.0.newfit' using 1:3 with lines
    fmode
    # This is it!
stop
#
```

```
# The real work starts here
# We shall use a three parameter fit
set parameter A 3
# Marquardt-Levenberg
set method ml_fit
# Adjust them all
adjust 1 2 3
# Initialize
let A[1] = -2
let A[2] = 1
let A[3] = 2
# This is the function fitting vector U: Therefore, it must be called UFIT.
# One can use any temporary variable (e.g. TMP) to accelerate computation
# Note how to build names of parameter partial derivatives
set function user
    UFIT = A[1] + s* (A[2]^2 + A[3]^2* T)^(-0.5)
    DUFITD1 = 1                        # derivative wrt A[1]
    TMP = s*(A[2]^2 + A[3]^2*T)^(-1.5) # Temp var speeds up!!!
    DUFITD2 = -A[2]*TMP                # derivative wrt A[2]
    DUFITD3 = -A[3]* T * TMP           # derivative wrt A[3]
stop
# Do all directories and append to the parameter file
# 0.0001 0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.80 0.90 0.95 0.97 1.00
foreach Dir in echo 0.0001 0.[1-9]? 1.00
    echo Doing $Dir
    fitscript $Dir
end
```

Some more examples are given in a subsequent chapter. You should now be able to write your own fitting scripts. I suggest that you read the following items in the reference manual:

```
adjust, fit, set/show function, set/show iteration, set/show method,
set/show parameters, show fit, show/save parameters
```

and other related topics as found in the description of these commands.

2.8 Interactive Command Line Shell

When used interactively, the shell keeps a complete history of previous commands, independently of the mode.

The shell also supports command completion and filename completion. In the fitting mode, if the tab key is pressed while the cursor is in the first argument, FUDGIT will try to expand to a known command, alias or macro. Therefore, pressing tab twice on an empty line will list all the currently active commands, macros and aliases. If the cursor is not located in the first argument of the line, then FUDGIT will try to expand the argument to an existing filename or to an existing printable variable if a $ is found before the alphanumeric string to expand. Filename expansion supports the '~' HOME designator.

While in C-calculator mode, the expansion reduces to all what is found in the math lookup table, including variables, constants, vectors, functions, procedures and keywords.

In the plotting mode, expansion reduces to filename completion or to printable variable completion if the alphanumeric string to expand starts with a '$'.

Lines can be continued on the next one by using a '\' at the end of the line as in *csh*. Usually expanded characters can be put as is by *escaping* them with a leading '\'.

2.9 Dynamic Loading

Some operating systems[1] allow to load an object (a *file.o*) at run-time.[2] To do so, a dynamic loading (dl or dld) library, as well as some features of the loader `ld(1)` on IRIX are required. The present version of FUDGIT supports dynamic loading on IRIX, ULTRIX, and SUNOS only. However, FUDGIT needs to be linked (at compilation time) by the GNU `ld(1)` link editor when built on SUNOS.

Thus, the user can load any module of his own and momentarily `install` it in the internals of FUDGIT as a procedure (not returning a value) or a function (returning a value). Be sure external routines are of type *void* for procedures and of type *double* for functions, although a cast is internally done.

The `install` command allows the user to choose a different name for the internal installation, with the only restriction that the name be composed of lowercase letters only (as `proc` and `func` do). Once the external functions and/or procedures are installed, they can be used like any internal function or procedure.

Arguments are all passed by pointers to the external routine. Thus, the user have direct access to vectors, parameters, and strings passed as arguments. The case of variables is different however. Even if the value of an expression-variable is passed by pointer, the address is pointing to a temporary location storing the value of the expression. This is done so that calls like

```
let y = myfunc(X, sqrt(sin(x)), data)
```

be possible. A full example is presented in the section of the reference manual describing the `install` command. Also read the file *fudgit.h* located in the *tools* directory of this distribution. This file should be included in your modules.

Note that although vector indices run from 1 to `data` in the C-calculator mode, they run from 0 to `data-1` when manipulating the same vectors from your own C modules and from 1 to `data` when in FORTRAN. The pointers passed are always the address of the first element.

FORTRAN functions and procedures can be loaded provided an underscore is appended to the function or procedure name. This means that if you have a subroutine called *mysub* in a compiled object called *fortran_mod.o*, you must called `install` with

```
install fortran_mod.o mysub_:myname(X, n)
```

to install procedure `myname()` which accepts one vector and one expression as arguments. All functions and procedures are strongly typed in FUDGIT. This means that all arguments are checked for number and type consistency. Note that the colon (:) means to install `mysub` as a procedure called `myname`. Using an equal sign (=) would have installed it as a function. The user can install several functions and procedures from the same object: they would only need to be listed at the end of the preceding example line.

Future versions of FUDGIT should provide support for string arguments in FORTRAN modules.

The command `reinstall` is equivalent to `install` but it is used when a module has already been loaded once in the course of a FUDGIT session. `reinstall` will first unload the module so that symbols do not get defined twice in the program.

More details and a full example will be found under the `install` item of the next chapter.

[1] It will become a standard for the next generation of UNiX.

[2] While the program is running, and without recompiling it.

Chapter 3

Reference Manual

This chapter describes all the supported commands. As a convention, any name printed in *italics* is a generic name. Any name printed in `typewriter` style is a command name that has to be typed as such. Optional arguments are given a subscript$_{\text{optional}}$ at the end of the optional string. All commands are assumed to be *fitting mode* commands except when specified otherwise.

3.1 &

The '&' operator forces FUDGIT to use the built-in following fitting mode command and to ignore any existing macro or alias with the same name. This can be useful in constructions like:

```
macro cd 1
    &cd $1                  # The built-in cd
    pmode cd "$1"
stop
```

See also:

`macro, cd`

3.2 \

If anywhere in the middle of a line, a '\' will indicate FUDGIT to take the following character as is. If at the end of a line, a '\' indicates that the present line continues on the following one, and thus to ignore the following carriage return.

See also:

`line editing`

3.3 !

Any line beginning with the so-called bang operator '!' will execute the system command line with a Bourne shell. Aliased commands as found in your interactive C-shell do not hold any more. For example, commands like `!rm` will not be interactive (i.e. /bin/rm -i) even if you have such an alias in your *.cshrc* file. Be careful! A nice turnaround is to alias rm to '`! rm -i`' in your *.fudgitrc* file and to use the `rm` command directly from FUDGIT's shell.

When used in a macro name or an alias name, the '!' character has still another meaning. This tells the parser that characters following the '!' are optional. Therefore, if one types the following, interactively, (see NOTE)

```
set noexpand
alias da!te !date
set expand
```

then the parser will recognize **da**, **dat** and **date** as all synonymous to the system command **! date** run through a Bourne shell.

NOTE: In interactive mode, the history functions will try to interpret a history substitution if the '!' is not followed by a space. See the appendices. To avoid that the line be scanned for a history event designator, use the **set noexpand** command. In some cases, it might be simpler to use the **system** command.

Syntax:

!command

Example:

```
! mail
```

See also:

alias, ls, vi, foreach, system, set expand

3.4 ?

A question mark will indicate FUDGIT to try to get the possible options available to the command presently typed. This kind of help is context sensitive and works when an insufficient number of arguments is supplied. The question mark also serves as a wild character in string subtraction.

Syntax:

command **?**

Examples:

```
?
show ?
set function ?
```

See also:

help, strings

3.5 $

The '$' operator expands scalar variables or constants (double precision numbers from C-calculator mode lookup table) as well as string variables or constants. Existing scalar variables can thus be expanded as a string in order to serve as a file name or directory name, for example. The expansion is done according to the value given to the **set vformat** command which initially defaults to "%.3g". Using the scalar variable expansion operator in C-calculator mode is not recommanded since a lot of precision might be lost (actually it is a waste!). Scalar variable expansion is essentially provided to allow alternative procedures in certain cases, such as generating filenames from numbers. Math function **scan** can be considered as the complement of scalar variable expansion.

The '$' character also expands string variables. Expansion is done by replacing the *$String-Variable-Name* by the value of the string variable. This can be used to replace **scan** in cases where the string variable or constant represents a number. For example

```
foreach File in echo 2.2 4.4 6.7 8.32
    let x = $File
    .
    .
    .
end
```

In both cases, if the variable name has to be followed by alphanumeric characters, then the variable name can be delimited by braces as in standard csh.

Followed by an integer number, the '$' character serves to designate the arguments of a macro. Refer to the description of `macro`, concerning this point.

Syntax:

`$`*name*

or

`${`*name*`}`

See also:

`C, cmode, macro, echo, exit`

3.6 _dumplot

Command `_dumplot` is generally used in a macro to dump vectors in the plotting pipe. It is described in more detail under `special` item.

3.7 _killplot

Command `_killplot` is rarely used. It sends a KILL signal to the plotting program. It is described in more detail under `special` item.

3.8 adjust

The `adjust` command is used to specify the parameters to be adjusted in the "least square linear" and the "Marquardt-Levenberg nonlinear" fitting methods. Parameters not being adjusted will have their standard deviation set to zero.

Syntax:

`adjust` *index-list*

Example:

```
adjust 1 2 4
```

See also:

`set parameters, set method, set function, fit, show fit`

3.9 alias

The `alias` command is used to alias a multiple word command to a single word. Although macros and aliases are different objects, it is not allowed to define a macro and an alias with the same name since aliases are always expanded first. Recall that the bang operator ('!'), at the beginning of a line is recognized from a macro, an alias or a script file so that an alias like

```
alias date !date
```

is perfectly legal. However, this would have to be typed

```
alias date ! date
```

at the interactive command line, to avoid that the '!' be interpreted by the history functions.

When called without arguments, `alias` will list all the current aliases. For obvious reasons, it is not allowed to `alias` *unalias*. `alias` also supports the command abbreviation character '!'. To enter a '!' without having it interpreted by the history functions, just `set noexpand` for the time entering the command. When a '!' is part of the alias name this indicates that the alias command name can be abbreviated down to that point. Since the `&` operator is used to refer to the native commands, it is therefore forbidden to start an alias name by character '&'.

Syntax:

`alias` *command command-list*

Examples:

```
alias mv !mv
alias . quit
alias da!te !date
```

See also:

`!, &, macro, unalias, set expand`

3.10 append and save

The `append` command can be used to append various things to an existing file. If the file does not already exists, it will be automatically created.

The `save` command can be used to save various things to a file. If a file with the same name already exists, it will be overwritten without any warning.

3.10.1 append history

History can be saved or appended to a file. Any file saved this way can later be executed by the `load` command. Note that `append history` will silently fail if the file does not exist.

Syntax:

`append history` *filename*
`save history` *filename*

See also:

`load, line editing, fmode`

3.10.2 append macros

All the current macros and aliases can be saved or appended to a file. Any file saved this way can be subsequently `load`ed at any time. To avoid confusion between data files and script files we recommand that you use the *.ft* extension for your script files.

Syntax:

`append macros` *filename*
`save macros` *filename*

See also:

`alias, unalias, load, show, macro, unmacro`

3.10.3 append parameters

Parameters can be saved into a file at any time. The number output format will be the one chosen by the `set format` command. The column order will be a parameter followed by its standard deviation. All columns are separated by a tab. Therefore, if one has previously set parameters, i.e.

```
set parameters MYPAR 3
 .
 .
 .
save parameters myfile
```

then there will be 6 columns as follows:

```
MYPAR[1]    DMYPAR[1]       . . .      MYPAR[3]     DMYPAR[3]
```

in file *myfile*. Most of the time, the user will desire to save parameters along with some variables or constants. This can be done by giving the variable or constant (either string or scalar) names on the command line. For example,

```
let t = 0.23
set parameters A 2
.
.
.
save parameters t parfile
```

will create a file *parfile* containing the value of scalar variable `t`, followed by the 2 values of parameters `A`, alternated with the value of their standard deviations `DA`. Note that the given list of variables will be printed first.

Syntax:

`append parameters` *variable-list*$_{\text{optional}}$ *filename*
`save parameters` *variable-list*$_{\text{optional}}$ *filename*

See also:

`set format, set parameters, show parameters`

3.10.4 append variables

Any variable or number of variables can be saved to a file at any time. Vector elements referenced by an explicit index are considered as variables. String variables and constants are recognized as well.

Syntax:

`append variables` *variable-list filename*
`save variables` *variable-list filename*

Examples:

```
append variables x Y[3] a VECTOR[78] datafile1
save variables t PARAM[2] DPARAM[2] datafile2
```

See also:

`load, cmode, let, C, show, auto`

3.10.5 append vectors

Any vector or number of vectors can be saved to a file. All the values are written in columns separated by a tab. The number format will be the one chosen by the `set format` command.

Syntax:

`append vectors` *VECTOR-list filename*
`save vectors` *VECTOR-list filename*

Examples:

```
append vectors X Y ERROR1 TEST2 datafile1
save vectors TIME TEMP DT datafile2
```

See also:

`set format, set data, read, fit, fft, show, auto`

3.11 auto

The `auto` keyword is used to define automatic variables. The type of variable can be a scalar variable, a VECTOR or a String, depending on the upper-lower case letters in the variable name. The scope of auto variables is delimited by braces as in C. All auto variables are stored on the stack and are freed when the scope of the variable is left. Definition of variables can only be done right after a brace has been opened. Only scalar variables can be assigned as the are defined, while vectors are assigned to zero, and strings are empty. Contrarily to C, automatic scalar variables are set to zero if not assigned. `auto` is a C-calculator mode keyword.

Syntax:

`auto` *var-list*

Examples:

```
# Some dummy examples
set data 100
cmode
   x = y = 1               # These (x, y) are global
   X = y++                 # As well as vector X
   { auto x=2, X, Y        # All these variables are local...
```

```
        X=3; Y=sin(x)
         .
         .
         .
    }                       # ...and stop existing here
    x                       # This x still contains 1
# An example with a procedure
    proc test(x) {
        auto y=2
        z = x + y++         # This z is global
    }
fmode
```

See also:

`C, cmode, func, proc`

3.12 break

The `break` keyword is used as in C to break C-calculator mode `for` or `while` loops. `break` is a C-calculator mode command.

Syntax:

`break`

See also:

`C, continue, cmode, for, while`

3.13 C

The following gives a brief description of the supported C-calculator syntax and differences with standard C.

The following operators are recognized, in order of precedence:

`++, --`	`(post and pre) increment-decrement`
`-, !`	`unary minus and logical NOT`
`^`	`exponentiation, right associative`
`/, *, %`	`division, multiplication, modulo`
`+, -`	`addition, subtraction`
`>, >=, <, <=, ==, !=`	`relational operators`
`&&`	`logical AND`
`||`	`logical OR`
`=, +=, -=, /=, *=`	`assignments, right associative`

All operators are left associatives except those specified. They are all common to C except for the exponentiation operator.

The following keywords are reserved tokens: `auto, if, else, while, for, break, continue`, and `return`, plus two extra keywords `proc, func`. They roughly obey the same syntax as in C so that statements like:

`if (`*conditions*`)`
cmode-line-statement

or

```
if (conditions) cmode-line-statement
```

or

```
if (conditions) {
cmode-statements
}
```

The same thing is true for the else constructions `else` of which some examples follow:

```
if (conditions)
cmode-line-statement
else
cmode-line-statement
```

or

```
if (conditions) {
cmode-statements
} else {
cmode-statements
}
```

Here *cmode-line-statement* means any semicolon separated list of C-calculator mode statements typed on the same line. Since semicolons are separators and not terminators, empty statements are defined by empty braces { }.

The `return` keyword must have parentheses when returning a value from a function as in `return(x * sin(y))`. A single `return` will only be recognized from within a procedure.

To avoid potential confusion with variables, keywords cannot be abbreviated.

As opposed to C, there exists no integer in the C-calculator mode. All scalar variables and numbers are double precision. This means that logical true is 1.0 and false is 0.0. As in C, one must be careful with comparison operators. The C `switch` syntax is not supported (would require integers).

As an extension, string comparison is possible with the equality operators '==' and '!='. This will return true or false if the string variables (or constants) are identical or not. Assignments of string variables actually copies all characters of the string on the RHS to the string variable on the LHS. String additions and subtractions are also possible.

Function and procedure definitions are defined with prototypes, i.e., a list of variables representing the proper kind of variable. At run-time, the arguments of the function are checked for type compatibility and for their number.

All variables are global except automatic variables defined using the `auto` keyword.

See also:

`cmode, let, math, scan, strings, auto`

3.14 cd

The `cd` command changes the working directory. Called with no argument, `cd` will bring you to your $HOME directory. Note that `cd` changes the current working directory of FUDGIT only. Therefore, your plotting program will still be in the previous directory. To get around this difficulty, you only have to define a macro as follows, if your plotting program supports `cd`:

```
macro Cd 1
    &cd $1
    pmode cd "$1"
stop
alias cd Cd
```

Syntax:

`cd` *filename*$_{\text{optional}}$

Examples:

```
cd
cd /nazgul/users/fulano
```

See also:

`&, pwd, alias`

3.15 cmode

The `cmode` command allows you to go in the C-calculator mode. The only way to come back to the main fitting mode is by using the `fmode` command or to type ^D in interactive mode. Commands cannot be abbreviated in `cmode`. Parallel to the `cmode` command, the `let` command can be used to pass one single command, or command line to mathematical parser. To be consistent with `pmode` command, `cmode` also accepts arguments in which case it is equivalent to the `let` command. It is not an error to call `cmode` from the C-calculator mode. A warning message will be given though.

Syntax:

`cmode` *command-list*$_{\text{optional}}$

The C-calculator mode supports most of C syntax (see item C), and most of the C math library. Thus, the following functions are supported:

trigo:	hyperbolic:	expo:	special:	conversion:	random:
cos()	cosh()	ln()	besy0()	trunc()	srand()
cot()	coth()	log()	besy1()	floor()	rand()
csc()	csch()	exp()	besj0()	ceil()	
sec()	sech()	sqrt()	besj1()	rint()	
sin()	sinh()	cbrt()	besjn()	abs()	
tan()	tanh()		besyn()	int()	
acos()	acosh()		erf()	scan()	
asin()	asinh()		erfc()	min()	
atan()	atanh()		lgamma()	max()	
atan2()			interp()	sum()	
				vread()	

Any upper case variable (possibly including '_') possibly mixed with digits will be recognized as a vector, e.g., `TEMP_2, TEST, D`, etc. Any lower case name will be taken as a scalar variable, e.g., `x, t4`, etc. There are two predefined constants, `pi` $= \pi$ and `e`$= e$, which should not be unlocked and modified. As well, the built-in constant `data` contains the current size of the vectors and can be modified through the `set data` command, by the `read/exec` commands, or by `unlock`ing the constant and modifying it directly. The built-in constant `chi2` contains the value of χ^2 as obtained from the latest fit. And finally, the built-in scalar constant `param` contains the number of parameters as defined by `set parameters`.

A mix of upper case and lower case letters will serve to indicate a string variable. Strings values are indicated by double quotes as in C. Unlike C, FUDGIT considers strings as self-contained objects that can be added, subtracted, and checked for (in)equality. Thus, string objects (i.e. string variables, string constants and string values) can: serve as argument to `scan` function; be part of string assignment statements or of a truth statement involving (in)equality operator; be added (concatenated using the '+' operator) one with another; be subtracted (remove string termination using the '-' operator) one with another; and finally be argument of string functions.

A predefined string constant called `Tmp` contains the string "/tmp/fudgitPID" where PID is the process id number of the current process. This file, and any file belonging to you, whose name starts with the same string, will be erased automatically by the `exit` or `quit` commands. This string is typically used by the `gnuplot` macro in order to pass data to the GNUPLOT plotting program which cannot read data from standard input. Another predefined string constant is `ReadFile` which contains the last data filename that has been loaded. Finally, the string constant `Cwd` is made available in order to get the current working directory.

The following table contains all the built-in constants.

```
chi2            Value of chi^2 from the last fit;
data            Length of all vectors (< samples) as set by set data;
e               Neperian number;
param           Number of parameters as set by set parameters;
pi              Guess this one;

Cwd             Current working directory;
ReadFile        The last file (program) read by read (exec);
Tmp             A temporary filename "/tmp/fudgitPID";
```

Constants (either strings and scalars) can also be created by `locking` a variable. In the same manner, a constant can be modified directly if it has been `unlocked`.

The algebraic operations applicable to scalar variables can be applied to vectors. Vector algebra can be mixed with scalar variable algebra in which case the user has to take the implied loop into account. For example, although the following operation is not standard C programming: [1]

```
cmode
    x = 0
    X = x++
```

will define a vector X of size `data` (see `set data`) ranging from $X[1]$ to $X[data]$ and taking values from 0 to $data - 1$. Multiple commands can be given with the separator ';', for example, another version of the previous command could be written

```
cmode
    x=0;X=++x
```

in which case a vector X taking values from 1 to $data$ will be created. (Note that the latter uses a pre-increment whereas the former uses a post-increment operator on `x`: results are thus different). Vector elements can be referenced by elements using standard C grammar. Therefore, the same vector could be created by using a `while` construction as in:

```
fmode
set data 1000
let X=0;i=0
cmode
    while (i++ <= data)
       X[i] = i
fmode
```

or, using a `for` loop,

```
cmode
   for (x=0;x<=data;x++) {
```

[1] NOTE: In order to show that some commands can be typed from both C-calculator mode and the fitting fmode, the following examples shows the typing mode from the first line. However, one can always type the same C-calculator mode command from the fitting mode by using the `let` command (or `cmode` command).

```
        X[x] = x
    }
fmode
```

Noninteger variables will be truncated to the nearest lower integer to form a vector index.

```
cmode
    y= 2.01
    x=2.23; X[2]=Z[y]+5^x
```

Assigning a vector to a constant will assign all the elements to that constant.

```
fmode
let X = pi
let Z2 = 0
```

The C-calculator checks for undefined variables on the RHS of any assignment. From C-calculator mode, variables values can be seen by typing the variable name by itself or by using the `print` command, if the output is selected to be *stdout*. From the fitting mode, contents of constants and variables (either strings or scalars) is displayed using `show variables` command, or by using the '$' expansion operator. However, vectors can be only be seen from the fitting mode by using the `show vector` command.

Each unknown vector name given on the command line allocates a vector of `sample` size.

To be a calculator as such, the C-calculator prints the value of the expression given on the command line. Thus, the statement

```
cmode
    x + 2
```

will print the value of $x + 2.0$. The contents of many variables can be displayed at the same time by giving a coma separated list such as in

```
cmode
    x,"temperature", t
```

where the string *temperature* will be printed between the values of variables x and t. Note that the C-calculator mode recognizes strings by double quotes. Special characters such as '\n' are also legal in a string.

We conclude by giving some examples involving string variables:

```
fmode
let String = "new.file"
let x = (String == "new.file")
let y = ("file1" == "file2")
let Bing = "\a\a\a"
let Here = Cwd        # Store the value of the current working directory
let Input = Read()                        # Read from stdin
let Test = FileName(ReadFile) - ".data"
let Dir = DirName(InputFile)
let y = scan(Read(), "%lf")
let File = "STRING_23.4"
let number = scan("%*[_A-Z]%lf", File)
let Message = "A tab \t and a newline\n"
```

where the truth statement could be legally used as a condition for an `if`, a `while`, or a `for`.

See also:

`let, C, data, func, proc, print, fmode, math, while, for, return, auto, if, break, samples, quotes, strings`

3.16 comments

By default, anything following a '#' will be treated as a comment and ignored. This holds for data files as well as for command script files loaded with the `load` command. This default can be changed with the `set comment` command. Sometimes a comment character needs to be taken literally in a script file. The comment character will be accepted as data if it follows the '\' escape operator, i.e. '\#', or, in the fitting mode only, whenever the comment character is somewhere inside quotes or parentheses. The comment character is always accepted literally when typed on the interactive command line.

See also:

`set comment, read, load, show comment, exec`

3.17 continue

The `continue` keyword has the same usage it has in C for sending the control to the next iteration of a `for` or `while` loop. `continue` is a C-calculator mode command.

Syntax:

`continue`

See also:

`for, while, cmode, C`

3.18 data files

Files containing data are loaded by specifying the name of the data file to the `read` command. Data files should contain one data point per line. A data point can be a 256 dimensional object. By default, anything following character '#' will be treated as comment and ignored. In all cases, the numbers on each line of a data file must be separated by any number of blank spaces or tabs. These blanks divide each line into columns. Thus, FUDGIT can handle up to 256 columns per line. Warning will be given if a line has a different number of columns. Strings such as *NaN* or *Infinity* are recognized and refused. The default compilation gives a maximum line size of 1024 characters.

See also:

`read, exec, set comment`

3.19 echo

The `echo` command allows the user to print a string to the standard output. If no argument is given `echo` will only print a newline. This command can be used to display a message or, when coupled with the variable expansion operator '$', to see the value of a printable (either string or scalar) variable defined in the C-calculator.

Syntax:

`echo` *string-list*

Examples:

```
echo Starting the fit
echo $Mydir
```

See also:

`cmode, $`

3.20 else

The `else` keyword is used in `if` constructions, both in C-calculator anf fitting modes. Refer to the `if` entries for a complete description.

3.21 end

The `end` command is used to complete a `foreach` loop or a `while` loop. Keyword `end` is also used to tell `read` that we are finished writing data to stdin. This command should always be found on a line by itself (comments are allowed though).

See also:

`foreach, while, read, stop`

3.22 endif

The `endif` command is used to complete an `if` construction in fitting mode. Keyword `endif` must always be used on a line by itself (comments are allowed though). Refer to the `if` entries for a the complete description.

3.23 environment

FUDGIT is sensitive to the following environment variables:

- PAGER for the program called to format long listings.
- HOME for the directory to which `cd` defaults.
- SHELL for the shell called by `system` when this latter is called without arguments.

If not defined, the default pager is */usr/?/more* (path depends on system) and the default shell */bin/csh*.

See also:

`cd, system, show vectors, help`

3.24 exec

The command `exec` executes a program and reads data from it. It supports the same syntax `read` does except that the program name replaces the file name. A program is a program name or anything that can be typed in a shell. If the command line has more than one string, it must be glued with quotes. On a successful call, `exec` will set the string constant `ReadFile` to the name of the program which generated the data.

Syntax:

`exec` *commands assignment*$[$*range*$]_{\text{optional}}$ ...

Examples:

```
exec simulate X:1 Y:2[0:200]
exec "cat data | myfilter -g" X1:1[0:*] X2:2 X3:4
```

See also:

`read, comments`

3.25 exit

The commands `exit` and `quit` will exit FUDGIT. See details under item `quit`.

Syntax:

```
exit
```

See also:

```
quit, cmode
```

3.26 fft

The `fft` command will take the Fourier transform of the specified vectors and put the real part in a vector specified by the third argument. The imaginary part will be put in a vector specified by the fourth argument. Input vectors can be used for output. The resulting vectors will contain frequencies ranging from 0 to $N/2$ followed by $-(N/2-1)$ to -1 in units of $1/(N*\Delta)$ where Δ is the sampling rate. If a real vector is transformed $h(t) -> H(f)$, we should have $H(-f) = H^*(f)$. Therefore, with $H = R + iI$ and $H^* = R - iI$ be the transformed vectors, we should have $R(-f) = R(f)$ and $I(-f) = -I(f)$, where f is discrete and ranges as mentioned above. In terms of vector indices, these relations become $R[i] = R[N - i + 2]$ and $I[i] = -I[N - i + 2]$ for $1 < i < N/2$ in addition to the fact that $I[1] = I[(N/2) + 1] = 0$. Therefore, because the negative frequency part is the mirror image of the positive one, it is common to plot only the positive frequencies of the Fourier transform of a real vector. This can be done by reducing `data` to half its value.

Because of the use of a FFT algorithm, the number of data points must be an integer power of 2. If not, the user should pad the vector with zeros up to the next largest power of two. Each transform is normalized by the factor $\sqrt{(N)}$ so that `fft RE IM T_RE T_IM` followed by `invfft T_RE T_IMA RE2 IMA2` will not introduce a factor N in vectors `RE2` and `IMA2` (i.e., `RE` = `RE2` and `IM` = `IM2`). At his choice, the user can use the C-calculator functionality in order to implement windowing.

The power spectrum can be obtained from:

```
fft RE IMA T_RE T_IMA
let POW = T_RE^2 + T_IMA^2
```

where $POW[i]$ will contain the power value associated with frequency f, which goes from 0 to $N/2$ followed by $-(N/2-1)$ to -1 (in units of $1/(N*\Delta)$) as i goes from 1 to N.

Syntax:

`fft` *real-VECTOR ima-VECTOR real-VECTOR ima-VECTOR*

Examples:

```
# real vector X
let IM=0
# re-use IM vector for output
fft X IM Z IM
# complex vectors X+iY where i = sqrt(-1) transformed in V+iW
fft X Y V W
```

See also:

```
invfft, smooth, cmode, let, read, math, data
```

3.27 fit

The `fit` command is used to fit a function, chosen by `set function`, to a pair of vectors containing the independent and dependent variables. Depending on the type of fit, selected by the `set method` command, a third vector containing the standard deviation might be required. `fit` allocates a vector having the name of the dependent variable appended with the string `FIT`. This vector contains the computed values of the function for the given independent vector. Depending on the method, the built-in constant `chi2` will contain the value of the mean square deviation weighted by vector σ-*VECTOR* or the mean absolute deviation.

Syntax:

`fit` *independent-VECTOR dependent-VECTOR* σ-*VECTOR*

Example:

```
fit X Y DY
```

will create a vector `YFIT` containing the value of the fitted function for each of the values of the independent vector `X`. Note that the standard deviation is required for most fitting routines since it is used to weigh the value of local square deviation from the fit (in fact, this is the definition of χ^2). If σ-*VECTOR* is unavailable just use

```
let DY=1
```

using the previous example. This simply gives the same weight to all data points.

See also:

`set method, set function, show fit, show parameters, append`

3.28 fmode

The `fmode` command allows you to return to the fitting mode, when the program is in one of the C-calculator or plotting modes. The fitting mode, is the main mode of the program. The two other modes are the C-calculator mode, accessed by the `cmode` command, and the plotting mode, accessed by the `pmode` command. When used interactively, ^D returns to the fitting mode from either of the C-calculator mode or from the plotting mode. It is not an error to call `fmode` from the fitting mode. A warning message will be given though.

Syntax:

`fmode`

See also:

`cmode, pmode, let`

3.29 for

The `for` command is a C-calculator mode command. It behaves roughly like a standard C `for` construction. In interactive mode, any new input line will be prompted with a "n{...n\t" where 'n' stands for the nesting level and '\t' for a tab. Keyword `for` is a C-calculator mode command.

Syntax:

`for` (*init-expressions*; *cond-expressions*; *loop-expressions*)
cmode-line-statement

or

```
for (init-expressions; cond-expressions; loop-expressions) {
cmode-statements
}
```

Examples:

```
cmode
    for (i=1,j=2;i+j <= data; i+=2,j+=3) A[i] = X[j]
fmode
# Another example:
# A macro to remove point x in a vector. Syntax: delete "vector" "index"
macro delete 2
    cmode
        for(i=$2;i<data;i++) {
            $1[i] = $1[i+1]
        }
    fmode
    unlock data
    let data--
    lock data
stop
```

See also:

C, break, continue, cmode, if, set data, func, proc, if, lock, math

3.30 foreach

The foreach command loops through the strings obtained from a given UNIX command. Wild card characters are allowed since everything following the in keyword is passed to a Bourne shell for execution. Strings can be obtained from any program including the easiest cases *echo, ls* and *cat*. The variable name must be of string type, i.e., consisting of both upper case and lower case letters (and possibly _'s and digits).

Syntax:

```
foreach StringVarName in UNIX-command
body of the loop
end
```

Example:

```
#convert columns 2 and 3 of the following files in log-log format
foreach Fname in ls data*.7[0-9] datatest.42 data*.8[4-7]
    echo $Fname ...
    read $Fname X:2[0.001:*] Y:3[0.001:*]
    let X = log(X)
    let Y = log(Y)
    save vectors X Y $Fname.log
end
```

See also:

for, math function scan, while, macro

3.31 free

The command `free` is made available for memory management. It is used to free vectors, variables, functions, procedures, and numbers that were allocated in the C-calculator mode.

When called with the special argument "@all", `free` will erase all the user vectors, numbers and variables, as well as all active functions and procedures (not macros and aliases). Otherwise, `free` will free the specified vector(s) or variable(s). Constants (either scalar or string) cannot be removed without first unlocking them.

Syntax:

`free` *VECTOR- or variable-list*
`free @all`

Examples:

```
free @all
free X y TEMP
```

See also:

`unlock, C, cmode, show table, show memory, samples, let`

3.32 func

The `func` command defines a function. A function is distinct from a procedure from the fact that a function must return a value whereas a procedure must not. Arguments are given in the definition with any name prototype representative of the data type. As in C, the argument list must be comma separated when calling the function (after having defined it). An example follows. `func` is a C-calculator mode command.

The prototype list defines the type of variable to be used. Although all global variables are accessible from within the function, variables are always searched for from the prototype list first, then from the local list (`auto` variables), and finally from the global list. All scalar variables are passed by value: thus any scalar expression is legal as scalar argument. String arguments and vector arguments are passed by pointer: thus string and vector arguments must refer to a variable explicitly. The `show table` can be used to list all the installed objects at a given time.

Syntax:

`func` *functionname*`(`*proto-list*$_{\text{optional}}$`)` *cmode-line-statement*`; return(`*value*`)`

or

`func` *functionname*`(`*proto-list*$_{\text{optional}}$`) {`
cmode-statements
`return(`*value*`)`
`}`

Examples:

```
# The following example will print the factorial of all integers up to 120.
cmode
    func fac(x) {  # This 'x' is a prototype: it does not exist.
        if (x <= 0)  {
           return(1)
        } else {
           return(x * fac(--x))
        }
   }
   x=1  # This 'x' is a global scalar variable
```

```
   while (x<120) {
       fac(x++)
   }
fmode
# The following calculates the average of a vector
cmode
    func avg(X) {
        auto i,x

        for (x=0,i=1;i<=data;i++) {
            x += X[i]
        }
        return(x/data)
    }
fmode
```

See also:

`C, return, for, while, cmode, math, free, proc, show table, install`

3.33 help

The `help` command displays on-line help. To specify information on a particular topic use the syntax:

`help` *topic*

If *topic* is not specified, a short message is displayed about FUDGIT. Topic names can be abbreviated down to the shortest unambiguous string. In case of doubt, `help` will print out all possible completions. Thus, `help f` will print all help topics starting with the letter 'f'. After help for the requested topic has been given, help for a subtopic may be requested by typing the subtopic name, extending the help request. After that subtopic help has been displayed, the request may be extended again, or pressing return will return one level back to the previous topic. Eventually, the fitting mode prompt will return.

See also:

`help?`

3.34 history

The `history` command lists all the previous command lines, along with a number. History lines can be called using the !*string* construction or the !*number*. History is only available in interactive mode. See Appendix A for more details.

Syntax:

`history`

See also:

`append history, line editing`

3.35 if

There are two kinds of `if` constructions available in FUDGIT, one in the fitting mode and the other in the C-calculator mode.

3.35.1 C-calculator mode if

In C-calculator mode, `if` and `else` are reserved keywords. C-calculator mode `if` construction is similar to the one in standard C. Note that *cmode-statements* refers to any sequence of C-calculator mode commands and that *cmode-line-statement* refers to a semicolon separated list of C-calculator mode commands typed on the same line.

Syntax:

`if (`*conditions*`)` *cmode-line-statement*

or

`if (`*conditions*`)`
cmode-line-statement

or

`if (`*conditions*`) {`
cmode-statements
`}`

or, using the `else` constructions,

`if (`*conditions*`)`
cmode-line-statement
`else`
cmode-line-statement

or, for statements on more than one line,

`if (`*conditions*`) {`
cmode-statements
`} else if (`*conditions*`) {`
cmode-statements
`} else {`
cmode-statements
`}`

See also:

`C, cmode`

3.35.2 Fitting mode if

Fitting mode `if` has a syntax very similar to the one in C-shell. It requires the keywords `then` and `endif` and supports `else` constructions. The difference resides in the fact that the conditional statement has to follow C-calculator mode grammar and syntax and thus has a richer set of operators. The '$' expansion operator is therefore not needed in the conditional statement, as it is in C-shell conditional statements. All active variables, constants, and their string counterparts, are directly available to the conditional statement. Note that the *fmode-statements* can also contain C-calculator mode commands (even possibly including C-calculator mode `if`'s!).

Syntax:

`if (`*conditions*`) then`
fmode-statements
`endif`

or, using the `else` constructions,

```
if (conditions) then
fmode-statements
else if (conditions) then
fmode-statements
else
fmode-statements
endif
```

See also:

`while, foreach, macro`

3.36 in

The `in` keyword is required in `foreach` constructions in fitting mode. Refer to the latter for details.

3.37 install

The `install` command dynamically loads defined routines from an object file. The user decides on the internal name of the routine but the internal name must consist of lower case letters only. The object file is an object compiled by the C or FORTRAN compiler. On IRIX, the object must be compiled with the option `-G 0` given to either the C or FORTRAN compiler. The *rtn-name* is the name of one of the procedure(s) or function(s) the user wants to install from the object file. The routine will be installed as *name* and as a procedure (not returning value) or as a function (returning value) depending on the name separator being a `:` (colon) or a `=` (equal sign) respectively. (See example below).

NOTE: This option is only available on IRIX and SUNOS for the moment.

The external routine must expect pointers to double for all its arguments. Thus, all arguments are passed by pointers except that pointers to variables do not point to the variables as such but to a temporary copy of them. This allows us to have expressions like

`f = mycall(X, sin(x) + 1, data)`

for which the value `sin(x) + 1` must necessarily be a temporary copy. In this example, the prototype is a function `mycall(VEC, expr, expr)`. We shall consider an example in more detail below. All arguments are strongly typed as vector, parameter, expression or string. Prototyping is done using the uppercase-lowercase convention. Parameters are prototyped using the word `PARAM` or less (e.g. `PAR`).

On IRIX, the linker will create a binary file built from the module name and with the extension *file*.`ld`. This binary is the one that will be loaded in memory. Time stamps are included so that `ld(1)` will not be called if not necessary. These files are not erased at exit, since they are reusable and prevent the linker to be called if nothing changed between two sessions of FUDGIT.

Successive calls of `install` with the same module should not be done unless the same functions and procedures are reinstalled. If this is the case, the user should then reinstall the same modules (that could have been modified and recompiled in the mean time) using the `reinstall` command. If a module is reinstalled with different routines or function or procedure names, the previously defined functions or procedures might not be properly installed anymore and calling them might result in an undefined behavior.

The file *fudgit.h* describes the functions user-defined programs can linked with. Among other things, these functions allow the user to have elegant error handling and exit.

The `show table` command can be used to list all the installed objects at a given time.

A file having the same base name of the module but with the extension *libs* can be put in the same directory in order to include extra libraries while loading the module. On IRIX, these extra libraries must all contain objects compiled with the flag *-G 0* (see `cc(1)`). (For example, some IRIX systems have a -lm_G0 math library.) User-defined libraries can be specified along with system libraries. A typical example could be a line like:

```
/home/myname/myproject/libmyG0.a /usr/lib/libmG0.a
```

for linking with user's library */home/myname/myproject/libmyG0.a*. Equivalently, for non-IRIX systems, loading a FORTRAN object might require something like this:

```
/usr/lib/libF77.a /usr/lib/libm.a
```

Library names can be on multiple lines. However, the file cannot have more than 1024 bytes. A '#' found anywhere in this file will make the rest of the file to be ignored.

Note that when loading FORTRAN code, the user must append an underscore to the routine name so that `install` or `reinstall` can find it.

The IRIX version does not fully support incremental linking, i.e., to use, in an object to be installed, symbols that were defined in previously `install`ed objects. However, all the symbols contained in the original FUDGIT executable remain at all time available to all linked routines. Therefore, IRIX users should make sure that external objects are self-contained and only reference to external routines that are intrinsic to FUDGIT or come directly (and once) from linked libraries at installation time.

Syntax:

`install` *object-file rtn-name*`[:|=]`*name*`(arg-list)...`

Example:

```
hostname: cat mymodule.c
    #include <math.h>
    #include "fudgit.h"

    /* An example of a user-defined routine inversing the order of an even
     * vector. Typical call would be:
     *  myproc(A_VEC, data)
     * from C-calculator mode. NOTE that both VEC and expr are pointers.
     * To make things explicit, fudgit.h contains a few typedef's.
     */

    void myproc(X, dn)
    VEC X;
    expr dn;
    {
        int i, half_n;
        int n = (int)*dn;  /* note that dn is a pointer to a double */
        double tmp;

        if (n%2 == 1)   /* report error if odd number (Why not?)*/
            Ft_matherror("%s: Called with an odd number %d.", "myproc", n);

        /* You have full use of math and stdio libraries too!!! */
        fprintf(stderr,
        "BTW, Did you know that %lf is the sqrt(pi)?\n", sqrt(M_PI));

        half_n = n >>1;  /* half of n */
        for (i=0;i<half_n;i++) { /* Standard C: indices from 0 to data-1 */
            tmp = X[i];
            X[i] = X[n-i];
            X[n-i] = tmp;
        }
    }
```

```
    /*
     * Another example involving a function. The following calculates the
     * non-normalized correlation between vectors A and B as defined by
     * corr(A, B) = <A*B> - <A> * <B>
     *
     */

    double myfunc(A, B, dn)
    VEC A, B;
    expr dn;
    {
        int i, n = (int)*dn;  /* Again, dn is a pointer to a double */
        double sumA, sumB, sumAB;

        sumA = sumB = sumAB = 0.0;
        /* sum up the values of interest */
        for (i=0;i<n; i++) {  /* indices go from 0 to data-1 */
            sumA += A[i];
            sumB += B[i];
            sumAB += A[i] * B[i];
        }
        /* leave it simple */
        sumA /= *dn;
        sumB /= *dn;
        sumAB /= *dn;

        return (sumAB - sumA*sumB);
    }
hostname: cc -G 0 -O -c mymodule.c
hostname: cat loadex.ft
    # This is an example for loading
    # Install function myfunc as corr() and procedure myproc as inverse()
    # Prototypes are made from any name representing the proper type:
    install mymodule.o myproc:inverse(V, n) myfunc=corr(V, V, n)
    set data 24
    let x=1;X=x++
    let Y=sin(X)
    cmode
        # Inverse order of vector X
        inverse(X, data)
        # Calculate correlation between X and Y
        y=corr(X, Y, data)
        # Print its value
        "correlation:", y
    fmode
hostname: fudgit loadex.ft
install: myproc installed as procedure inverse.
install: myfunc installed as function corr.
BTW, Did you know that 1.772454 is the sqrt(pi)?
correlation: 9.20717026e-01
```

When linking FORTRAN functions or subroutines, the user must append an underscore after every function or subroutine name. All argument variables and vectors have to be defined `double precision` as

well as returning functions. Typical examples are included in the distribution in the *tools* directory.

See also:

`C, cmode, show table, func, proc`

3.38 invfft

Command `invfft` performs the inverse Fourier transform of the given vectors. It assumes that the frequencies are ordered from 0 to $N/2$ followed by negative frequencies ranging from $-(N-1)/2$ to -1 in units of $1/(N*\Delta)$ where Δ is the sampling interval. The results are normalized by a factor $1/\sqrt{(N)}$ so that a transform followed by an inverse transform should give the original vector. The resulting vectors are stored in the third and fourth arguments. Thus, `invfft X Y V W` inverse transforms X+iY into V+iW. Input vectors can be used as output vectors. See `fft` for more details.

Syntax:

`invfft` *real-VECTOR ima-VECTOR real-VECTOR ima-VECTOR*

See also:

`fft, smooth, cmode, let, read, math, data`

3.39 let

The `let` command opens the door to the C-calculator mode from the fitting mode, but leaves the program in fitting mode. All the `let` commands can always be typed directly from the C-calculator mode without having to prepend with the `let` keyword. The converse is also true; all the commands given in C-calculator mode could be typed from the fitting mode by prepending them with the `let` command. Although `let` is typed from the fitting mode, the remainder of the line is parsed according to C-calculator mode rules, and thus quotes are no longer swallowed. Variable expansion operator '$' is still recognized, but its use is not recommanded for C-calculator statements. See '$' for more details on this point.

Syntax:

`let` *C-calculator-mode-commands*

Examples:

```
# generate the zero order first kind bessel
# function between (0, 2*pi]
fmode
set data 2000
let x=1; X=x++
let tmp = 2*pi/data     # compute sequence only once
let X *= tmp
let Y = besj0(X)
```

See also:

`cmode, C, math`

3.40 line editing and history

The command shell supports line editing and history. The editing commands are based on the basic EMACS commands. A short summary follows but a more complete description can be found in Appendix B.

Line editing:

- ^B moves back a single character.
- ^F moves forward a single character.
- ^A moves to the beginning of the line.
- ^E moves to the end of the line.
- ^H and DEL delete the previous character.
- ^D deletes the current character.
- ^K deletes from current position to the end of line.
- ^L,^R redraws line in case it gets trashed.
- ^U deletes the entire line.
- ^W deletes the last word.

History:

- ^P moves back through history.
- ^N moves forward through history.
- !! previous command.
- !$ previous command last argument.
- !*string* last command starting with *string*.

Completion:

- tab complete command if first arg, filename otherwise.
- esc-? or double tab list possible completions.

Each line of input must be smaller than 1024 bytes which is more than sufficient for most applications. Lines can be continued on several lines provided carriage returns follow a '\' (as in standard shells).

See also:

`append history, $, history`

3.41 load

The `load` command executes each line of the specified input file as if it had been typed in interactively. Files created by the `save history` command can be `load`ed directly. Text files containing valid commands can be created and then executed by the `load` command. Files being `load`ed may themselves contain `load` commands. See `comment` for information about comments in command scripts. The `load` command is recursive so it can be nested. The only limitation is the I/O stack which has a default capacity of 32. This value can be easily changed at compilation time of the program.

The current working directory always returns to the value in effect before the loaded script was called. This is valid for nested `load` commands too.

In order to avoid confusion between data files and script files we strongly recommand you to stick to the conventional *.ft* extension for your script files.

Syntax:

`load` *filename.ft*

A `load` command is also performed implicitly on any filenames given as arguments to `fudgit`, when called from your UNIX session. These are loaded and executed in the order specified, and then FUDGIT exits.

See also:

`set comment, exec, startup, append history, append macros`

3.42 lock

Variables can be turned into constants using the `lock` command. Once a variable is `lock`ed, any assignment trying to change its value will result in a parsing error. This is valid for both scalar and string variables. It is not an error to try to lock a constant. A warning message will be given though. However, trying to lock an unexisting variable or something else than a constant or variable will result in an error.

Syntax:

`lock` *var-list*

See also:

`C, cmode, unlock`

3.43 ls

The command `ls` calls "/bin/ls -FC". If any arguments are given, those are passed to "/bin/ls -FC". Wild card characters are possible since expansion is done by a Bourne shell.

Syntax:

`ls` *ls-argument-list*

Examples:

```
ls p* test?
ls -l datafile
ls -l *.data
```

See also:

`system, alias`

3.44 macro

The `macro` command allows the user to define macros. Macros can be embedded, but another macro cannot be defined from within a macro, mainly because of their common way to refer to arguments. The name of the macro followed by the number of arguments required must be given. The maximum number of arguments a macro can have is 16. An exclamation mark in the macro name will indicate that the macro name can be abbreviated and that the characters following the exclamation point are optional. Macros are only recognized in the fitting mode. The total length of each macro is limited to 2048 bytes in size. Macros can be nested to a maximum of 32. Macros are only recognized from the fitting mode.

Syntax:

`macro` *macroname argument-number*
body of the macro
`stop`

Example:

```
# define a macro named fpl!ot (o, t, are optional)
# requiring 3 arguments . Uses the plotting program gnuplot.
# Syntax: fplot X Y YFIT
# plot X Y with data points and X YFIT with solid line
macro fpl!ot 3
    # save vectors in temp file (will be automatically removed on exit)
    save vec $1 $2 $3 $Tmp.fplot
    # plot second column with points and third with line
    pmode plot '$Tmp.fplot' us 1:2 wi point, \
       '$Tmp.fplot' us 1:3 wi line
stop
```

See also:

`append macros, show macros, load, startup, unmacro, alias, unalias`

3.45 math functions

The C-calculator mode math functions found in FUDGIT are very close to the corresponding functions found in the UNIX math library. Some other functions, not found in the math library, are also part of FUDGIT. Most of the numerically unstable functions (i.e. ln, log, exp,...) check for both an argument out of range and a value out of domain at each call. All math functions are double precision and can only be called from the C-calculator mode, or by using the `let` command from the fitting mode. These functions are also available in the conditional statements of the fitting mode `if` and `while`, since these statements are C-calculator mode statements, although part of fitting mode constructions.

3.45.1 math function abs

The `abs()` function returns the absolute value of its argument.

3.45.2 math function acos

The `acos()` function returns the arc cosine (inverse cosine) of its argument. `acos()` returns its argument in radians.

3.45.3 math function acosh

The `acosh()` function returns the positive (principal) hyperbolic arc cosine (inverse cosine) of its argument.

3.45.4 math function asin

The `asin()` function returns the arc sine (inverse sine) of its argument. `asin()` returns its argument in radians.

3.45.5 math function asinh

The `asinh()` function returns the hyperbolic arc sine (inverse sine) of its argument.

3.45.6 math function atan

The `atan()` function returns the arc tangent (inverse tangent) of its argument. `atan()` returns its argument in radians.

3.45.7 math function atan2

The `atan2(y, x)` function returns the arc tangent (inverse tangent) of the ratio of its arguments (y/x). `atan2()` returns its argument in radians. The signs of y and x are used to determine the quadrant.

3.45.8 math function atanh

The `atanh()` function returns the hyperbolic arc tangent (inverse tangent) of its argument.

3.45.9 math function besj0

The `besj0()` function returns the j0th Bessel function of its argument, i.e it returns the zeroth order Bessel function of the first kind. `besj0()` expects its argument to be in radians.

3.45.10 math function besj1

The `besj1()` function returns the j1st Bessel function of its argument, i.e it returns the first order Bessel function of the first kind. `besj1()` expects its argument to be in radians.

3.45.11 math function besjn

The `besjn(n, x)` function returns the jnst Bessel function of its argument, i.e it returns the n^{th} order Bessel function of the first kind. `besjn()` expects its second argument to be in radians.

3.45.12 math function besy0

The `besy0()` function returns the y0th Bessel function of its argument, i.e it returns the zeroth order Bessel function of the second kind. `besy0()` expects its argument to be in radians.

3.45.13 math function besy1

The `besy1()` function returns the y1st Bessel function of its argument, i.e it returns the first order Bessel function of the second kind. `besy1()` expects its argument to be in radians.

3.45.14 math function besyn

The `besyn(n, x)` function returns the ynst Bessel function of its argument, i.e it returns the n^{th} order Bessel function of the second kind. `besyn()` expects its second argument to be in radians.

3.45.15 math function cbrt

The `cbrt()` function returns the cubic root of its argument.

3.45.16 math function ceil

The `ceil()` function returns the smallest integer that is not less than its argument.

3.45.17 math function cos

The `cos()` function returns the cosine of its argument. `cos()` expects its argument to be in radians.

3.45.18 math function cosh

The `cosh()` function returns the hyperbolic cosine of its argument.

3.45.19 math function cot

The `cot()` function returns the cotangent of its argument. `cot()` expects its argument to be in radians.

3.45.20 math function coth

The `coth()` function returns the hyperbolic cotangent of its argument.

3.45.21 math function csc

The `csc()` function returns the cosecant of its argument. `csc()` expects its argument to be in radians.

3.45.22 math function csch

The `csch()` function returns the hyperbolic cosecant of its argument.

3.45.23 math function erf

The `erf()` function returns the error function of its argument. The error function is defined as

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

3.45.24 math function erfc

The `erfc()` function returns `1 - erf()` where `erf()` is the error function of its argument. It is provided because of the extreme loss of relative accuracy if `erf(x)` is called for large x and the result subtracted from 1.0 (e.g., for $x = 10$, 12 places are lost).

3.45.25 math function exp

The `exp()` function returns the exponential function of its argument (e raised to the power of its argument). Overflow is checked on all `exp()` operations.

3.45.26 math function floor

The `floor()` function returns the largest integer not greater than its argument.

3.45.27 math function hypot

The `hypot(x, y)` function returns sqrt(x*x+y*y) computed in such a way that underflow will not happen, and overflow occurs only if the final result deserves it.

3.45.28 math function int

The `int()` function returns the integer part of its argument, truncated toward zero. The returned value is still a double. This function is equivalent to trunc() is is kept for compatibility.

3.45.29 math function interp

The `interp()` function returns an interpolated value of the function at the value of its argument. The functional relation is previously initialized using the fitting mode command `spline`. The interpolation is obtained from cubic splines. *Natural* (i.e., the second derivative of the interpolating function at either or both the first and last point of the original data equal zero) cubic spline or specific first derivatives at the extreme points of the original data set are specified while initializing the process using `spline` command.

See also:

`spline`

3.45.30 math function lgamma

The `lgamma()` function returns the natural logarithm of the gamma function of its argument. For an integer `n, lgamma(n+1) = ln(fac(n))` where fac is a factorial function.

3.45.31 math function ln

The `ln()` function returns the natural logarithm (base e) of its argument. Illegal argument is checked for.

3.45.32 math function log

The `log()` function returns the logarithm (base 10) of its argument.

3.45.33 math function max

The built-in function `max(x, y)` returns the maximum value of x and y. `max(x, max(y, z))` obviously returns the largest value of x, y, and z.

3.45.34 math function min

The built-in function `min(x, y)` returns the minimum value of x and y. `min(x, min(y, z))` obviously returns the smallest value of x, y, and z.

3.45.35 math function rand

The `rand()` function returns a random number between [0,1). Depending on the machine on which it is compiled, it might use the extended 48 bits random number generator or less.

3.45.36 math function rint

The `rint()` function returns the value of its argument rounded to the nearest integer.

3.45.37 math function scan

This math function is a bit different from others in the fact that it handles strings and returns a number. In fact, the `scan(`*String, Format*`)` function returns a double precision number as extracted from string *String* and according to string format *Format*. The format is built with the same rules `sscanf` uses. See man pages on `scanf(3)`. Note that the format must contain one active "%lf". An example might be of some help here, especially to show how to use `scan` in conjunction with C-calculator mode defined strings. `scan` is particularly helpful to extract numbers from filenames. Recall that strings are defined by double quotes as in standard C.

At this point, it might be useful for you to know the "%[ ]" scanf construction. Let's go through some examples: `"%*[a-zA-Z]"` means to ignore the longest string matched so that it is composed of any letter; `"%*[^0-9]"` means to ignore the longest string matched so that it is NOT composed of any digit; `"%*[^_.]"` means to ignore the longest string matched so that it is not composed of characters '_' or '.'.

Examples:

```
# define a string called Testname
let Testname = "dummy25.dat"
# let y be the Neperian log of the number contained in that string
let y = ln(scan(Testname, "%*[^0-9]%lf.dat"))
# The following reads a number from stdin
let input = scan(Read(), "%lf")
```

See also:

`$, strings, C, cmode, quotes`

3.45.38 math function sec

The `sec()` function returns the secant of its argument. `sec()` expects its argument to be in radians.

3.45.39 math function sech

The `sech()` function returns the hyperbolic secant of its argument.

3.45.40 math function sin

The `sin()` function returns the sine of its argument. `sin()` expects its argument to be in radians.

3.45.41 math function sinh

The `sinh()` function returns the hyperbolic sine of its argument. `sinh()` expects its argument to be in radians.

3.45.42 math function sqrt

The `sqrt()` function returns the square root of its argument.

3.45.43 math function srand

The `srand()` function sets the seed of the random number generator. Its argument will always be truncated to an integer towards zero. `srand()` returns the truncated value.

3.45.44 math function sum

The sum function returns the sum of the elements of the vector passed as an argument. Recall that vector are passed by pointers so that `y = sum(X^2)` is not legal. Instead, on must explicitly calculate

```
# Given vector X, the following calculates the sum of X^2
let X2 = X^2
let y = sum(X2)
```

in order to evaluate the sum. The sum function can be used to calculate basic statistics (mean, standard deviation, correlation, ...) and to do basic integration together with a spline-interp algorithm if the points are distant and the function smooth enough.

See also:

`interp, spline`

3.45.45 math function tan

The `tan()` function returns the tangent of its argument. `tan()` expects its argument to be in radians.

3.45.46 math function tanh

The `tanh()` function returns the hyperbolic tangent of its argument. `tanh()` expects its argument to be in radians.

3.45.47 math function trunc

The `trunc()` function returns the value of the argument when truncated towards zero.

3.46 pause

The **pause** command displays any text associated with the command and then waits a specified amount of time or until a carriage return is pressed if the given time value is a negative integer. The **pause** command is especially useful in conjunction with `load`ed files.

Syntax:

pause *value* $string_{\text{optional}}$

Examples:

```
pause -1
pause 3
pause -1  Hit return to continue
pause 10  This fits equation 4 to file $ReadFile.
```

See also:

```
echo, load
```

3.47 plot

There exists no plot command as such. However two macros are predefined. One is `gnu!plot` to use with GNUPLOT and `sgi!plot` to use with SGIPLOT. As they currently are, only two vectors can be passed to these macros. They serve like examples for building your own macros as well. See `show macros` to see the contents of the predefined macros of your site.

See also:

```
set plotting, special, macro, show macros
```

3.48 pmode

The `pmode` command talks directly to the plotting program chosen with the `set plotting` command. Any command usually typed to the plotting routine is now valid. Furthermore, all the current variables, constants and their string counterparts can be expanded in the plotting mode. The fitting macros and aliases are not recognized in this mode. The command `fmode` permits the user to return from the plotting mode as does ^D when typed interactively. If `pmode` is called with trailing arguments, the remainder of the line will be passed to the plotting program while remaining in fitting mode. It is not an error to call `pmode` from the plotting mode. An warning message will be given though.

If the plotting program is defined as a null string (`set plotting ""`) then all command lines given in `pmode` will be ignored and warning messages will be given accordingly.

Syntax:

`pmode` $command_{\text{optional}}$

Examples:

```
pmode
pmode set nokey
pmode plot "fudgfile" with lines
```

See also:

```
set plotting, set prompt-pm, special
```

3.49 print

The `print` command is a C-calculator mode command that writes the value of a valid mathematical expression to a file selected by `set output`. The default is *stdout*. If there is more than one variable, a coma separated list must be given in which case each expression value will printed on the same line and separated by a tab. As with other number output commands, the output format is the one selected by the `set format` command. The default is "The `print` command differs from `show variables` as follows:

- `print` accepts any expression for indexing vector elements;

- `print` requires a comma separated list;
- `print` can be part of a function or procedure;
- `print` can print strings provided they are in double quotes. This includes characters '`\n`', '`\t`', '`\a`', ...;
- `print` does not append a newline.
- `print` can print any mathemetical expression.
- `print` is a C-calculator mode command.

A simpler way to print variables to *stdout* from the C-calculator mode is to use the feature that any variable or coma separated list of variables given on the command line will be displayed, separated by tabs and appended with a newline character. Thus the construction @ifhelp

```
set output stdout
cmode
    print x, y, "\n"
```

is equivalent to

```
cmode
    x, y
```

typed in C-calculator mode (it becomes `let x,y` in fitting mode). The only difference between `print` and the automatic printing feature of C-calculator mode is that (1) `set output` only affects `print` command, and that (2) `print` does not automatically append a new line character.

Syntax:

`print` *coma-separated-var-list*

Examples:

```
cmode
    print x+2
    print String, x, y, z
    print "Warning \a\a\a", "x = ", x, "\n"
```

See also:

`cmode, func, C, show table, show variable, math functions, quotes, set format, set output`

3.50 proc

The `proc` command is a C-calculator command used to define procedures. Procedures differs from functions in the fact that they do not return any value. The procedure arguments are passed and referred to the same way they are in functions. Keyword `proc` is a C-calculator mode command. The `show table` command can be used to list all the installed objects at a given time.

Syntax:

`proc` *procedurename*(*proto-list*$_{\text{optional}}$) *cmode-line-statement*

or

`proc` *procedurename*(*proto-list*$_{\text{optional}}$) {
cmode-statements
}

Examples:

```
# The following example will print the Fibonacci numbers lower than 1000
cmode
    proc fib(x) {
        a = 0
        b = 1
        while (b < x) {
            print b
            c = b
            b += a
            a = c
        }
        print "\n"
    }
    # The following 'for' loop is equivalent to the preceding fib()
    proc fib2(x) {
        auto a,b,c        # This proc creates no global variable

        for(a=0,b=1;b<x;c=b,b+=a,a=c) {
            print b
        }
        print "\n"
    }
    fib(1000)  # A procedure as called from C-calculator mode.
fmode
let fib2(1000)  # A procedure as called from fitting mode.
# A short example involving a vector
set data 10
let proc init(X, x) X=x
let b=3
let init(Y, 2/4 + b) # Shows that scalar can also be expressions.
```

See also:

`return, cmode, C, func, auto, math, show table, install`

3.51 pwd

The `pwd` command prints the name of the working directory on the screen.

Syntax:

```
pwd
```

See also:

```
cd, ls
```

3.52 quit

The commands `exit` and `quit` are equivalent and both will exit FUDGIT. On exit, all temporary files */tmp/fudgitPID** (note the wild card) will be erased. Here PID is the current process number. Moreover, if a plotting process is active, it will be sent a KILL signal. It is therefore a good habit to use the `$Tmp` string variable to build your temporary files.

Syntax:

`quit`

See also:

`cmode, exit`

3.53 quotes

In the fitting mode, single and double quotes serve to indicate that all the characters between quotes should be taken as only one word, even if there are some blanks (tab or space) among them. The difference between single and double quotes is that within the former variable expansion (using '$') does not take place whereas it does in the latter. Quotes are not recognized between parentheses.

In C-calculator mode, double quotes serve to indicate a string and parsing is done accordingly. As in C, double quotes can be included in a string using the '\' operator. Note that C special characters as '\n' for a newline, '\a' for a bell, '\t' for a tab, and so on, are recognized in a string. Single quotes have no special meanings. The only way to pass a '$' without expanding the following name is to escape the '$' with a '\'.

Thus, a null string is given by '' or "" in the fitting mode and by "" only in C-calculator mode.

In pmode, both single and double quotes are freely passed to the plotting program. This is valid when trailing commands are are passed to `pmode`, although FUDGIT implicitly stays in the fitting mode. Once again, expansion of a '$' followed by a string can be avoided using the escape character, i.e., by typing '\$'.

See also:

`exec, set plotting, math function scan, print`

3.54 read

The `read` command is used to read data points from a file or from standard input. Each column is assigned to a given vector. Vectors not already allocated will automatically be. Range of values can be specified on any variable using the [*low*:*high*] syntax. A '*' replacing a value will be taken as unexistent. Range of lines can be specified on any variable using the {*low*:*high*} syntax. The last line range given will be the only one in effect. If the file name specified is '–' data will be read from the current standard input until the keyword `end` is found on a line by itself. The `read` –and the `load` commands are recursive functions so they can be nested insofar as you can understand what is going on. An assignment consists in a vector name and a column number separated by a colon. After a file has been successfully read, `read` will put the name of the data file in string constant `ReadFile`.

Syntax:

`read` *filename assignment*$[range]_{\text{optional}}\{linerange\}_{\text{optional}}$ ...

Examples:

```
read file1 X:1[0:*] Y:2
read file2 TIME:2{100:400}
read - T:1 VALUE:2
1    2.3
2    4.7
.    .
.    .
.    .
end
```

The first form will read positive values of the first column in vector X and corresponding values of the second in vector Y. The second will read the second column of file *file2* from line 100 to line 400. The third will read T and $VALUE$ from stdin. The assignment does not need to be in increasing order of column. Also note that the first column is 1.

See also:

`exec, data`

3.55 reinstall

The `reinstall` command is used to perform the dynamical loading of a module that was already loaded. Typically, this is done after a module has been modified and recompiled. Refer to `install` for more detail.

3.56 return

The C-calculator `return` keyword is used as in standard C to return from a function or a procedure. Contrary to C, `return` requires parentheses when returning a value from a function. It is an error to return a value from a procedure or to not return anything from a function. `return` is a C-calculator mode command.

Syntax:

`return(`*expression*`)`
`return`

See also:

`C, cmode, func, proc, auto`

3.57 save

Look under **append** command description.

3.58 set

The `set` command sets a lot of options, as follows.

3.58.1 set comment

The `set comment` command selects the character which will cause the rest of the line to be ignored. The default value is '#'. Note that the effect of a comment character will be void if: (1) found somewhere between single quotes in fitting mode or (2) escaped with a '\'.

Syntax:

`set comment` *character*

Example:

`set comment ?`

See also:

`show comment, comments`

3.58.2 set data

The `set data` command changes the effective size of vectors. All the vector arithmetic checks for index boundaries. The `data` constant is the higher bound of the check and necessarily the size of all vectors. Changing the `data` value does not change the values nor the capacity of vectors. It only changes the upper bound on the value the index can take. The `data` constant is also changed by the commands `read` and `exec`, which set it to the number of valid data points read. Because the upper bound can never be higher than the

effective capacity of vectors, a `data` value higher than the current `samples` value will be refused. See `set samples`. Typically, `set data` is used when one wants the C-calculator to generate (and plot) vectors. The `read` and `exec` commands take care of adjusting it. `data` constant can also be changed from the C-calculator mode if the constant is `unlock`ed. However, no check is made to ensure the given value is not higher than `sample` size, in which case a segmentation fault will crash the whole program. It is always safer to use `set data`.

Syntax:

`set data` *number*

Example:

```
set data 300
```

See also:

`lock, unlock, read, exec, cmode`

3.58.3 set debug

The `set debug` command puts the reading of `load`ed files in verbose mode, so that debugging is more easily done. All the commands, expanded macros and/or string variables are echoed as they are executed. There are some different debug levels at the present time:

- 0 clear all the debugging states.
- 1 echo the expanded lines as they are read. The command is parsed and comments are stripped out. This is most useful for debugging script files. History substitutions are shown.
- 2 display all command lines as they are read from the script.
- 3 display the line numbers of the ignored lines as they are read from datafiles.
- 4 echo command lines as they are passed to the math parser.
- 5 turn the math parser debugger on. To use this, the program must have been compiled with the YYDEBUG preprocessor variable on.
- 6 trace the flow of fitting mode `if` constructions.

Debugging values are not exclusive so that more than one level can be turned on. Levels are subject to change.

Syntax:

`set debug` *value-list*

Example:

```
set debug 0 1 3
```

See also:

`load`

3.58.4 set error

FUDGIT allows the user to select among different possible error checks to be made on each single mathematical operations. The `set error` command will set computational error checks as follows:

- 0: clear all computational error check bits.
- 1: check for 'infinity' values.
- 2: check for 'not a number' values.
- 3: check for 'out of domain' math function errors.
- 4: check for 'out of range' math function errors.

Error checks are not exclusive and more than one can be specified on the command line. The default status has all error check levels activated (1 2 3 4).

It is sometimes desirable to disable one of the checks. For example, the operation $y = 1/sinh(x)$ will give a 'out of range' error for large x (> 709 on most machines), although y is in fact 0. If one uses `set error 0 1 2 3`, then no error will be reported and y will be set to zero accordingly.

Syntax:

`set debug` *value-list*

Example:

```
set error 0 2 3
```

See also:

`C, cmode`

3.58.5 set expand

In interactive mode, history expansion and substitution will occur only if the `expand` variable is set. It is disabled using `set noexpand`. The default is on.

Syntax:

```
set expand
```

See also:

`set noexpand, history, line editing`

3.58.6 set format

The command `set format` will set the printf format for variables. Use only if you are sure of what you are doing. It defaults to "% 10.8e". See `man printf(3)` if in doubt.

Syntax:

`set format` *string*

Examples:

```
set format %6.2lf
set format "% .8g"
```

See also:

`show, append`

3.58.7 set function

The `set function` command is perhaps the most crucial command in data fitting. It is used to select a built-in fitting function or to enter a user-defined function. The following fitting functions are available:

NAME	DESCRIPTION	PARAMETERS REQUIRED
straight	Straight line	(2 parameters)
sine	Sine series	(N parameters)
cosine	Cosine series	(N parameters)
legendre	Legendre series	(N parameters)
polynomial	Power series	(N parameters)
gauss	Gaussian series	(3N parameters)
expo	Exponential series	(2N parameters)
user	User-defined function	(N parameters)

Assume a variable vector X and a parameter vector A then, the nonlinear gauss fitting function is a series of gaussians where

$$f(X,A) = \sum_{i=1,4,7,\ldots,N} A[i] \times e^{-\left(\frac{(X-A[i+1])}{A[i+2]}\right)^2}.$$

The nonlinear expo function is a series of exponentials where

$$f(X,A) = \sum_{1=1,3,\ldots,N} A[i] \times e^{X*A[i+1]}.$$

For a user-defined function, the `set function user` will prompt for more input. The following input is related to the variable to fit. For purposes of clarity, let's say that we have to fit vectors `X Y DY`. This requires a fit function `YFIT` (the name is made from the dependent variable appended with `FIT`) and all the partial derivatives `DYFITD1, DYFITD2, ..., DYFITDN` taken with respect to the parameters $n = 1, \ldots N$. All these functions are defined one per line as in the case of a macro until a `stop` is entered. Temporary variables are permitted. `set function user` actually defines a C-calculator mode macro that will be executed before each iteration of the fit. Therefore the complete C-calculator mode grammar is fully supported here. Temporary vectors can thus be used to speed up the calculation.

The C-calculator macro can be a simple call to a predefined procedure. When defined so, the parsing does not have to be done at each iteration, and a slightly faster process should result.

Example:

```
# read column 1, 2 and 3 of file "file"
read file T:1 R:2 DR:3
# make a three parameter fit
set parameter K 3
# this is a linear fit; use singular value decomposition
set method svd_fit
# enter my function
set function user
    RFIT = K[1] + K[2]*T^0.5 + K[3]*T^1.5
    DRFITD1 = 1
    DRFITD2 = T^0.5
    DRFITD3 = T^1.5
stop
fit T R DR
```

The vector `RFIT` will contain the fitted function. The difference between the fit and real data can be obtained right away by defining a vector

```
let RDIFF = R - RFIT
```

that can be plotted with respect to `T`.

The same thing is done for nonlinear fit with the exception that the partial derivatives of the function with respect to the parameters will contain reference to some parameter(s). (This is precisely the meaning of nonlinear here).

There is virtually no restriction on the number of parameters (memory is the sole limitation: `set parameter` command allocates a matrix of `parameters` X `samples`). The only conditions are that a linear regression must have 2 parameters defined (this is obvious) and the built-in nonlinear functions must be modulo 3 for the series of gaussians and modulo 2 for the series of exponentials.

See also:

`fit, set method, adjust, proc, auto`

3.58.8 set input

The `set input` command selects the file for the input of the C-calculator mode `Read` and `vread` command. The string *stdin* is valid as a filename. If the selected file does not exist or cannot be read, an error message will be given and the value will go back to the default value, which is *stdin*.

Syntax:

`set input` *filename*

See also:

`Read, vread`

3.58.9 set iteration

The `set iteration` command permits the user to change the iteration number for the Marquardt-Levenberg nonlinear fitting method. See `set function`. The default value is 10. However, the fitting process will stop if there is no difference in χ^2 for two consecutive iterations. However, a negative value will force to iterate up to the absolute value of that number, without checking for convergence.

Syntax:

`set iteration` *value*

Example:

```
set iteration 3
```

See also:

`fit, set method, set function`

3.58.10 set method

The `set method` command allows the user to select the fitting method to be used when calling the `fit` command. The following methods are available:

```
NAME         DESCRIPTION
----         -----------
ls_reg       least square linear regression (2 parameters)
lad_reg      least absolute deviation linear regression (2 parameters)
ls_fit       general least square linear fit using QR decomposition
svd_fit      general least square linear fit using singular value
              decomposition
ml_fit       general least square nonlinear fit using
              Marquardt-Levenberg method
```

Among them, only `ml_fit` and ls_fit depends on `iteration` and `adjust`.

For all methods except `lad_reg`, the value of χ^2 will be put in the scalar constant `chi2`. In the case of `lad_reg`, will contain the average absolute deviation.

Syntax:

`set method` *method*

Example:

```
set method svd
```

See also:

`fit, set iteration, set function`

3.58.11 set noexpand

The `set noexpand` command disallows history expansion on the interactive command line.

Syntax:

```
set noexpand
```

See also:

```
set expand
```

3.58.12 set output

The `set output` command selects the file for the output of the C-calculator mode `print` command. The strings *stdout* and *stderr* are both valid as a filename. If the selected file already exists, it will be overwritten with no warning. The default value is *stdout*.

Syntax:

```
set output filename
```

See also:

```
print
```

3.58.13 set pager

The `set pager` command allows the user to select a pager. A pager is the program that is called when the structure to be displayed has more than 24 elements. The default pager is (1) the environment variable PAGER if it exists or (2) */usr/?/more* (path depends on system) if not. If `pager` is defined to a null string (""), then no pager will be used.

Syntax:

```
set pager string
```

Example:

```
set pager "more -c"
```

See also:

```
show, show pager
```

3.58.14 set parameters

The command `set parameters` will fix the parameter name and size. Since the set of parameters is a kind of vector, parameter name cannot contain lower case letters. Parameters are initialized to zero. A built-in scalar constant called `param` contains the number of parameters at all time.

Syntax:

```
set parameters parameter-name size
```

Example:

```
# set the vector D of size 3 to be determined by the fit.
set parameters D 3
```

See also:

```
show parameters, show setup
```

3.58.15 set plotting

The `set plotting` command changes the default plotting program used by the plotting mode. The default is GNUPLOT but this can be changed to any plotting program that can be driven from stdin. A maximum of 16 arguments can be passed when the program is first called. Changing the plotting program will send a KILL signal to the existing plotting program (if any). If the plotting program is set to a null string (""), FUDGIT will ignore all the plotting commands and warning messages will be given. Setting the plotting program to a file that cannot be found or executed will result in an error at the first `pmode` call.

Syntax:

```
set plotting command
```

Examples:

```
set plotting "/usr/local/bin/sgiplot -p"
set plotting /usr/local/bin/gnuplot
```

See also:

```
show plotting
```

3.58.16 set prompts

All three FUDGIT prompts can be changed by the `set` command. The name of the prompts are:

- `prompt-cm` for the C-calculator mode prompt (default: "cmode> ";
- `prompt-fm` for the fitting mode prompt (default: "fudgit> ";
- `prompt-pm` for the plotting mode prompt (default: "pmode> ".

A null string "" (i.e., two consecutive quotes) can be given to any of these.

Syntax:

```
set prompt-cm string
set prompt-fm string
set prompt-pm string
```

See also:

```
show prompts
```

3.58.17 set samples

The command `set samples` changes the current capacity of the fitting program. Typically, `samples` is set at the beginning of a session since all the existing vectors and variables are erased on this call. The default setting is 4000 points.

Syntax:

```
set samples value
```

Example:

```
set samples 6000
```

See also:

```
set data, cmode, let, lock
```

3.58.18 set vformat

The command `set vformat` will set the sprintf format used for the expansion of scalar variables by the expansion operator '$'. Use only if you are sure of what you are doing. It defaults to "%.3lg". See `man printf(3)` if in doubt.

Syntax:

```
set vformat string
```

Examples:

```
set vformat %6.2lf
set vformat "%.4lg"
```

See also:

```
$, cmode, C
```

3.59 shell

The `shell` command starts a shell according to your SHELL environment variable. It is equivalent to `system` command. Refer to the latter for details.

3.60 show

The `show` command is used to see the chosen options or to look at any defined vectors, parameters or variables.

See also:

```
set, echo
```

3.60.1 show comment

The `show comment` command echoes the current comment escape character.

Syntax:

```
show comment
```

See also:

```
set comment, comments
```

3.60.2 show data

The `show data` command displays the current value of `data` constant. Left for compatibility.

Syntax:

```
show data
```

See also:

```
set data, lock, unlock, set samples
```

3.60.3 show debug

The `show debug` command displays the current value of the `debug` variable. The value is displayed in octal since the `set debug` n command turns on the n^{th} bit of this number.

Syntax:

```
show debug
```

See also:

```
set debug
```

3.60.4 show error

The `show error` command displays the current value of the `error` computational check variable. The value is displayed in octal since the `set error` n command turns on the n^{th} bit of this number.

Syntax:

```
show error
```

See also:

```
set error
```

3.60.5 show input

The `show input` command shows the filename selected for the input of the C-calculator mode `Read` and `vread` command. The default value is *stdin*

Syntax:

```
show input
```

See also:

```
set input, Read, vread
```

3.60.6 show iteration

The `show iteration` command displays the current value of `iteration` variable.

Syntax:

```
show iteration
```

See also:

```
set iteration, set method
```

3.60.7 show fit

The `show fit` command displays the different quantities relevant to the current fitting method. Typical examples are χ^2, the covariance matrix, the curvature matrix, correlation factor, etc...

Syntax:

```
show fit
```

See also:

```
fit, set parameters, set function, set method
```

3.60.8 show format

The `show format` command displays the current value of `format` variable. The `format` string is used when displaying any number on the screen. Refer to printf(3) of the UNIX manual.

Syntax:

```
show format
```

See also:

```
set format, show
```

3.60.9 show function

The command `show function` displays the current function type. If the function type is `user`, then the user-defined function will be displayed.

Syntax:

```
show function
```

See also:

```
set function, show setup, fit, math
```

3.60.10 show macros

If called with an argument, the `show macros` command will display the specified macro. Otherwise, all currently defined macros will be displayed. The selected `pager` is called if the command is given in interactive mode (at the command line prompt).

Syntax:

`show macros` *macroname*$_{\text{optional}}$

See also:

```
set pager, save macros, alias
```

3.60.11 show memory

The `show memory` function will display the current state of memory consumption of the program. All sizes are given in bytes. It uses a direct call to mallinfo(3). The arena is the size of memory requested by the process to the kernel. It is then split in different blocks shared among the internal matrices and user's vectors, macros, functions, procedures, variables and history.

Syntax:

```
show memory
```

See also:

```
free, show table
```

3.60.12 show method

The `show method` command displays the current value of the fitting `method`. It contains *none* by default.

Syntax:

```
show method
```

See also:

```
set method, fit, set function
```

3.60.13 show output

The `show output` command shows the filename selected for the output of the C-calculator mode `print` command. The default value is *stdout*

Syntax:

```
show output
```

See also:

```
set output, print
```

3.60.14 show pager

The `show pager` command displays the current value of the `pager` program.

Syntax:

```
show pager
```

See also:

```
set pager, environment, show
```

3.60.15 show parameters

The command `show parameters` will display the parameter values on the screen. If the number of parameters is larger than 24, then the selected `pager` will be called if the command is given in interactive mode (at the command line prompt). As with `append` and `save parameters`, `show parameter` can accept optional variable or constant (either string or scalar) list of names, in which case the value of the given variables will be displayed along with the parameter values.

Syntax:

`show parameters` *variable-list*$_{\text{optional}}$

See also:

```
set pager, set parameters, save parameters, show fit
```

3.60.16 show plotting

The `show plotting` command displays the current value of the `plotting` program.

Syntax:

```
show plotting
```

See also:

```
set plotting, startup, pmode
```

3.60.17 show prompts

The `show prompts` command displays the current values of the different mode `prompts`.

Syntax:

```
show prompt-cm
show prompt-fm
show prompt-pm
```

See also:

```
set prompt, startup
```

3.60.18 show samples

The `show samples` command displays the current value of the `samples` variable. Recall that although `data` is responsible for the visible part of all vectors, vectors all have a fixed allocated length of `samples` long. Any change to `samples` through `set samples` frees all the existing vectors.

Syntax:

```
show samples
```

See also:

```
set samples, set data, cmode
```

3.60.19 show setup

The command `show setup` will show some values of the program, such as the last data filename read, the number of data points, current capacity, current comment character, current iteration number, current plotting program, etc. Left for compatibility.

Syntax:

```
show setup
```

See also:

```
set comments
```

3.60.20 show table

The command `show table` displays the current lookup table of the C-calculator mode parser. It shows all current variables, numbers, vectors and functions included in the internal table. It also shows the state of the internal machine (C interpreter), stack and frame used in the C-calculator. This is used mainly for debugging or to prevent stack or machine code overflow.

Syntax:

```
show table
```

See also:

```
free, show memory, cmode
```

3.60.21 show variables

Any constants or variables can be displayed on the screen. The `show variable` command differs from `print` as follows:

- `show variables` only accepts integers for indexing vector elements;
- `show variables` requires a blank separated list;
- `show variables` cannot be part of a function or procedure.

As it has been mentioned previously, this is due to the different types of parsing between the C-calculator and fitting modes. As with all other number displaying commands, the printing format is always the one selected by the `set format` command.

Syntax:

`show variables` *variable-list*

Example:

```
show variables x X[2] Y[2] DY[2] time
```

See also:

```
print, save variables, show table, show vectors, cmode
```

3.60.22 show vectors

Any vector or number of vectors can be seen on the screen. If the size of vectors is larger than 24, the selected `pager` will be called if the command is given in interactive mode (at the command line prompt).

Syntax:

`show vectors` *VECTOR-list*

Example:

```
show vectors X Y DY
```

See also:

`set pager, append vectors, read, cmode, let`

3.60.23 show vformat

The command `show vformat` will display the printf format used for the expansion of scalar variables by the expansion operator '$'. Refer to the printf(3) description in the UNIX manual for more details.

Syntax:

```
show vformat
```

See also:

`$, cmode, C, set vformat`

3.61 smooth

The `smooth` command uses a gaussian windowing function (low-pass filter) on a Fourier transform loop in order to smooth the given vector. The windowing function is $\exp(-(f/(\sigma \times f_{max}))^2)$ where f_{max} is equal to half of the smallest power of 2 larger than the number of data points `data`. Variable f is the frequency that ranges from 0 to f_{max}. More likely, the smoothing factor is a non null positive real number from the (0, 1] interval. A smoothing factor $\sigma >= 1$ leaves the vector unchanged.

The number of data points `data` needs not to be a power of 2.

To be used with discernment!

Syntax:

`smooth` σ *in-VECTOR out-VECTOR*

See also:

`fft, invfft, cmode, C`

3.62 special

The following special commands are left for debugging or macro purposes. They start with an underscore to avoid mistakes and remind of their special character.

`_killplot` will kill the current plotting program.

Syntax:

```
_killplot
```

`_dumplot` will send the following vectors in the plotting program pipe. This is only useful if the current plotting program accept data from its stdin. `_dumplot` can accept up to 16 arguments.

Syntax:

`_dumplot` *VECTOR-list*

Example:

```
_dumplot X Y DY
```

See also:

`macro, show macros, plot`

3.63 spline

The `spline` function initializes the internal table for the calculation of interpolated values using cubic spline method. Interpolated values are obtained from calls to the C-calculator math function `interp()`. The value of the first derivative at the first and last data points can be specified by optional arguments. If not specified, or if one of the optional arguments is an asterisk `*`, then a *natural* cubic spline is assumed in which case the interpolated curve is such that the second derivative at the extreme points (or one of them) is null. The asterisk is more likely to be used in cases where the user would like to specify the first derivative at the last point only. The independent vector must be such that its value increases monotonically.

Syntax:

`spline` *indep-VECTOR dep-VECTOR* $y1_{\text{optional}}$ yn_{optional}

Example:

```
# Read vectors having a functional relation Y = F(X) from file "datafile"
read datafile X:1 Y:2
# Initialize the spline (as being natural)
spline X Y
# Save extreme values
let from = X[1]; to = X[data]
# Say there were data=10 points and you want 100
set data 100
# Rebuild X vector
# First build X ranging [0, 1]
let x=0; X=x++; tmp=data-1; X/=tmp
# Then from 'from' to 'to': from + (to - from)*X
let tmp=(to-from); X = from + X*tmp
# Rebuild Y vector possibly containing original values as a subset
let Y = interp(X)
# Note that any value can be asked for
let interp(2.34*pi)
```

See also:

`math interp`

3.64 startup

If a file *.fudgitrc* exists in your home directory, it will be automatically loaded at startup time of the program. This is useful if one wants to include his own macros or have his own preferences loaded to FUDGIT. This file is loaded for both interactive use (`fudgit`) and batch use (`fudgit` *script1 script2...*).

Examples:

```
set plotting /usr/local/bin/sgiplot
set prompt-pm ""
set comment ?
set samples 10000
```

A file called *.hist_fudgit* is will be created in your home directory in order to keep history between calls of FUDGIT. The number of events is determined at compilation time and defaults to 52.

See also:

`environment, alias, set plotting, set prompt`

3.65 stop

The command `stop` is used to terminate a macro or a fitting function defined by the user. However, it can also be used in a script file in order to stop execution at a certain point. In this case, an warning message will report that `stop` is being used outside a macro or function and the file from which the command was found will be considered as at the end of file (EOF).

See also:

`macro, set function`

3.66 string functions

FUDGIT has a set of functions returning string objects. These are made available to deal with filename construction, or to read from standard input. To be consistent with string type, string functions are named with both lower case and upper case letters.

Strings can be added or subtracted in the C-calculator mode. String addition *s1* + *s2* simply concatenates strings *s2* to string *s1*. String subtraction *s1* - *s2* removes *s2* from the end of *s1*. Note that the wild card '?' is supported in string subtractions.

3.66.1 string function DirName

The string function `DirName` returns the directory name as extracted from the filename given as an argument.

Syntax:

`Dirname(`*String*`)`

See also:

`string functions FileName, Scan, Read`

3.66.2 string function FileName

The string function `FileName` strips the leading directory names of the filename given as an argument. Note that the UNIX command:

`basename` *File Extension*

is equivalent to the FUDGIT command:

`FileName(`*File*`) -` *Extension*

so that filename constructions can be made in `foreach` loop for example.

Syntax:

`FileName(`*String*`)`

Examples:

```
foreach File in ls /usr/machin/data/*.32
    read $File X:1 Y:2{2:23}
    # Some commands
    .
    .
    # let File be the filename only, less the ".32" extension
    let File = FileName(File) - ".32"
    # And let Dir be the directory name
    let Dir = DirName(File)
end
```

See also:

`foreach, string functions, cmode, $`

3.66.3 string function Read

The `Read` function read a line from the file chosen by the `set input` function, strips the newline character and returns the resulting string. If the input is `stdin`, the user will be prompted by a *?* and the program will stop until a non-null string is entered. This is most likely to be used in macros requiring some input during run time. The `Read()` function can be used to read numbers with the help of `scan()`. See the example below.

`Read` can also be used to build vectors by taking one every n points. This can be done by two imbedded `for` loops.

Note: The newline character is not passed to the string.

Examples:

```
# Read a string from stdin (the default)
set input stdin
let String = Read()
# How to get a value out of a string: equivalent to vread()
let value = scan(Read(), "\%lf") \eq
# How to skip lines in a file
# Read say file project/numbers.data
set input project/numbers.data
cmode
for (i=1; i<=top; i++) {
    Line = Read()                          # Read one line
    X[i] = scan(Line, "%lf");              # get first column
    Y[i] = scan(Line, "%*lf %*lf %lf");      # get third column
    for (j=1; j<n; j++) {
        Line = Read()                      # Read n-1 lines
    }
}
fmode
set input stdin
```

See also:

`set input, math function scan, string functions, $`

3.66.4 string function Scan

`Scan(`*String, Format*`)` function returns a string as extracted from string *String* and according to string format *Format*. The format is built with the same rules `sscanf` uses. See man pages on `scanf(3)`. Note that the format must contain one active "%s" or "%[]" construction. An example might be of some help here, especially to show how to use `Scan` in conjunction with C-calculator mode defined strings. `Scan` is particularly helpful to extract parts of filenames. Recall that strings are defined by double quotes as in standard C.

Knowing about the "%[ ]" scanf(3) construction might be useful at this point. Consider the following few examples: `"%[a-zA-Z]"` means to read the longest string matched so that it is composed of any letter; `"%[^0-9]"` means to read the longest string matched so that it is NOT composed of any digit; `"%[^_.]"` means to read the longest string matched so that it is not composed of characters '_' or '.'.

Examples:

```
# define a string called Testname
let Testname = "dummy25.dat"
# Read until a point is encountered
let Base = Scan(Testname, "%[^.]"))
```

See also:

`$, scan, string functions Read, DirName, FileName, C, cmode, quotes`

3.67 system

When called with arguments, the `system` command is equivalent to the '!' bang operator, so the remainder of the line will be given to a Bourne shell for execution. If `system` has no argument, a shell (depending on environment variable SHELL) will be started.

Syntax:

`system` *shell-commands*$_{\text{optional}}$

See also:

`environment, !, shell`

3.68 then

The `then` keyword is required in the fitting mode `if` constructions. Refer to the latter for details.

3.69 unalias

The `unalias` command unaliases any alias previously assigned by the `alias` command.

Syntax:

`unalias` *alias_name*

Examples:

```
unalias date
unalias gnuplot
```

See also:

`&, alias, macro, unmacro, append, show`

3.70 unlock

The `unlock` command changes a constant into a variable and thus allows the user to change its value. This is particularly useful in functions and procedures needing to change the value of the `data` constant. Unlocking `data` gives the user complete freedom on the effective size of vectors. No check is done on `data` assignments, and therefore assigning a value to `data` that is superior to `samples` will result in a program crash. For this reason, it is always safer to change `data` using the `set data` command. It is not an error to unlock a variable. A warning message will be given though. However, trying to unlock something else than a constant or variable will result in an error.

See also:

`lock, set data, set samples, cmode`

3.71 unmacro

The `unmacro` command is the counterpart of `macro`. It is used to undefine macros. As does `free`, `unmacro` accepts the "@all" string in which case all the macros will be erased and freed from memory.

Syntax:

`unmacro` *macro-list*
`unmacro @all`

Examples:

```
unmacro myplot
unmacro @all
```

See also:

`alias, unalias, append, show`

3.72 version

The `version` command displays the version number and the welcoming message of FUDGIT.

3.73 vi

The command `vi` calls the editor. It is equivalent to `!vi filename`. Note that wild cards are also recognized and expanded.

Syntax:

`vi` *argument-list*

Examples:

```
vi file
vi test.*
```

See also:

`!, system, alias, shell`

3.74 while

The `while` command allows the user to construct controlled loops on a series of operations. However, FUDGIT supports two kinds of `while` constructions, one in the fitting mode and the other in the C-calculator mode.

3.74.1 C-calculator while

The C-calculator mode while construction has a syntax similar to that of standard C. In interactive mode, any new input line will be prompted with a "n{...n\t" where 'n' stands for the nesting level and '\t' for a tab. Recall that *cmode-line-statement* means a string of semicolon separated C-calculator mode commands.

Syntax:

`while (`*conditions*`)` *cmode-line-statement*

or

`while (`*conditions*`)`
cmode-line-statement

or

3.74.2 Fitting mode while

The fitting mode `while` is very similar to the C-shell `while`. As for the `if` construction, the difference remains in a broader range of operators available to the conditional statement and the fact that the variable expansion operator '$' is not required.

As for the `foreach` construction, a `end` keyword is required to indicate the end of the loop. Note that *fmode-statements* can also contain C-calculator mode commands (including cmode `while` loops!). Recall that the conditional statement is a C-calculator mode expression.

Syntax:

`while(`*conditions*`)`
fmode-statements
`end`

See also:

`foreach, if, cmode`

Chapter 4

More Examples

Here follows a few more examples that can help writing your own scripts. The following scripts also show how commands can be abbreviated.

4.1 Example 1

```
# Example 1
# read a file and transform the first column to 1/sqrt(x) and plot
set plotting /usr/local/bin/gnuplot
pmode set term X11
macro plot 2
    save vec $1 $2 $Tmp
    pmode plot "$Tmp" with lines
stop
read landm.0 X:1 Y:2
# this line is ignored
# a line can be blanked too

let X = 1/sqrt(X)
plot X Y
# EOF
```

4.2 Example 2

```
# Example 2
# fit a straight line using singular value decomposition
set function poly
set parameters A 2
set method svd
read landm.0 X:1 Y:2
let X=1/sqrt(X)
let DY = 1
fit X Y DY
save vec X Y YFIT DY
```

4.3 Example 3

```
# Example 3
# fit a straight line using the nonlinear method (why not?)
set fun us
    YFIT = A[1] + A[2]*X
    DYFITD1 = 1
    DYFITD2 = X
stop
set met ml
set par A 2
adjust 1 2
read landm.0 X:1 Y:2
let X=1/sqrt(X)
# We have to use a weight of 1 for chi^2 if unknown...
let DY=1
let A[1] = -2
let A[2] = 1
fit X Y DY
sav par myfile
set plo /usr/local/bin/sgiplot -p
sgip X YFIT
# EOF
```

4.4 Example 4

```
# Example 4
# Make the Fourier transform of a sine distribution.
# Shows how padding can be done.
# Windowing can be done by multiplying your vector by a chosen
# distribution weight factor.
# Uses GNUPLOT.
macro dofft 2
    fft $1 $2 $1_FT $2_FT
    let POW$1 = $1_FT^2 + $2_FT^2
stop
macro doinvfft 2
    invfft $1 $2 $1_IFT $2_IFT
    let POW$1 = $1_IFT^2 + $2_IFT^2
stop
pmode clear; set data style line
# Use 800 points padded to 1024
set data 1024
let n=0
let N=n++
# The null imaginary part
let IM=0
let x=0
let X=x++
let X*=(2*pi/data)
# Initialize A to zero on 1024 points
let A=0
```

```
# But only fill 800 with the function
set data 800
let A=cos(2*X) + sin(2*X) + cos(20*X) + sin(20*X)
set data 1024
# Plot the function
gnu N A
pause -1 The function: Hit return
# Fourier transform
dofft A IM
# Plot first half of transform since real
unlock data
let data/=2
gnu N POWA
pause -1 The Fourier transform: Hit return
let data*=2
lock data
# Do the inverse Fourier transform
doinvfft A_FT IM_FT
# Plot the real part which should be the original vector.
gnu N A_FT_IFT
echo The function transformed back
# EOF
```

4.5 Example 5

```
# Example 5
# A bunch of utilities to use with GNUPLOT.
# All plot commands are written and then loaded by gnuplot
# Written by Ross Thomson.
#
# Initialize or Reset plot number
alias reset!Plot let plot_num = 0

# A macro to do a single plot in gnuplot
macro ipl!ot  2
   resetPlot
   plot $1 $2
stop

# Plot for gnuplot with repeated plots
macro pl!ot 2
    save vector $1 $2 $Tmp.$plot_num
    if (plot_num) then
        pmode replot "$Tmp.$plot_num"
    else
        pmode plot "$Tmp.$plot_num"
    endif
    let plot_num++
stop

# Plot for gnuplot with repeated plots and titles
macro tpl!ot 3
```

```
    save vector $1 $2 $Tmp.$plot_num
    if (plot_num) then
        pmode replot "$Tmp.$plot_num" title "$3"
    else
        pmode plot "$Tmp.$plot_num" title "$3"
    endif
    let plot_num++
stop

# Initialize variables before a batch plot
macro tbip!lot 0
    !rm -f $Tmp.gnuscript
    set output $Tmp.gnuscript
    resetPlot
stop

# Plot for gnuplot batch plots (handled by gnuplot) and titles
# Plotting instructions are only put in file and then given to gnuplot
macro tbpl!ot 3
    save vector $1 $2 $Tmp.$plot_num
    if (plot_num) then
        let print ", '$Tmp.$plot_num' title '$3'\\"
    else
        let print "plot '$Tmp.$plot_num' title '$3'\\"
    endif
    let plot_num++
stop

# To be called after a tbplot
macro bat!chplot 0
  let print "\n"
  set output stdout
  pmode load "$Tmp.gnuscript"
  pmode pause 0  "Batch plotted"
stop

# Some plotting macros
# Save the current graph in a square printable postcript file
# Usage: sqpost filename.ps
macro sqpost 1
    pmode
       set size 0.7,0.92
       set term post port 'Helvetica-Bold'
       set output '$1'
       replot
       set term X11
       replot
    fmode
stop

# Save the current graph in a rectangular printable postcript file
macro post 1
```

```
    pmode
       set size 0.7,1.0
       set term post port 'Helvetica-Bold'
       set output '$1'
       replot
       set term X11
       replot
    fmode
stop
```

GNU Readline and History Library

The followings are the user's guides as extracted from the manuals of GNU READLINE 1.1 which can be found integrally in the readline directory, along with the copyrights. They are reproduced here for the sake of completeness of FUDGIT user's guide.

Appendix A

Using History Interactively

This chapter describes how to use the GNU History Library interactively, from the user's standpoint. For information on using the GNU History Library in your own programs, see the section entitled *Programming with GNU History* in the complete manual.

A.1 History Interaction

The History library provides a history expansion feature that is similar to the history expansion in Csh. The following text describes the syntax that you use to manipulate the history information.

History expansion takes place in two parts. The first is to determine which line from the previous history should be used during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the previous history is called the **event**, and the portions of that line that are acted upon are called **words**. The line is broken into words in the same fashion that the Bash shell does, so that several English (or Unix) words surrounded by quotes are considered as one word.

A.1.1 Event Designators

An event designator is a reference to a command line entry in the history list.

`!` Start a history substitution, except when followed by a space, tab, or the end of the line... `=` or `(`.

`!!` Refer to the previous command. This is a synonym for `!-1`.

`!n` Refer to command line *n*.

`!-n` Refer to the command line *n* lines back.

`!string` Refer to the most recent command starting with *string*.

`!?string[?]` Refer to the most recent command containing *string*.

A.1.2 Word Designators

A `:` separates the event specification from the word designator. It can be omitted if the word designator begins with a `^`, `$`, `*` or `%`. Words are numbered from the beginning of the line, with the first word being denoted by a 0 (zero).

`0` **(zero)** The zero'th word. For many applications, this is the command word.

`n` The *n*'th word.

`^` The first argument. that is, word 1.

`$` The last argument.

`%` The word matched by the most recent `?string?` search.

`x-y` A range of words; `-`*y* Abbreviates `0-`*y*.

`*` All of the words, excepting the zero'th. This is a synonym for `1-$`. It is not an error to use `*` if there is just one word in the event. The empty string is returned in that case.

A.1.3 Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a `:`.

`#` The entire command line typed so far. This means the current command, not the previous command, so it really isn't a word designator, and doesn't belong in this section.

`h` Remove a trailing pathname component, leaving only the head.

`r` Remove a trailing suffix of the form '`.`'*suffix*, leaving the basename.

`e` Remove all but the suffix.

`t` Remove all leading pathname components, leaving the tail.

`p` Print the new command but do not execute it.

Appendix B

Command Line Editing

This text describes GNU's command line editing interface.

B.1 Introduction to Line Editing

In this tex a the following notation is used to describe keystrokes.

The text `C-k` is read as 'Control-K' and describes the character produced when the Control key is depressed and the `k` key is struck.

The text `M-k` is read as 'Meta-K' and describes the character produced when the meta key (if you have one) is depressed, and the `k` key is struck. If you do not have a meta key, the identical keystroke can be generated by typing `ESC` *first*, and then typing `k`. Either process is known as **metafying** the `k` key.

The text `M-C-k` is read as 'Meta-Control-k' and describes the character produced by **metafying** `C-k`.

In addition, several keys have their own names. Specifically, `DEL`, `ESC`, `LFD`, `SPC`, `RET`, and `TAB` all stand for themselves when seen in this text, or in an init file (see section *Readline Init File*, for more info).

B.2 Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press `RETURN`. You do not have to be at the end of the line to press `RETURN`; the entire line is accepted regardless of the location of the cursor within the line.

B.2.1 Readline Bare Essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use `DEL` to back up, and delete the mistyped character.

Sometimes you may miss typing a character that you wanted to type, and not notice your error until you have typed several other characters. In that case, you can type `C-b` to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with `C-f`.

When you add text in the middle of a line, you will notice that characters to the right of the cursor get 'pushed over' to make room for the text that you have inserted. Likewise, when you delete text behind the cursor, characters to the right of the cursor get 'pulled back' to fill in the blank space created by the removal of the text. A list of the basic bare essentials for editing the text of an input line follows.

`C-b` Move back one character.

`C-f` Move forward one character.

`DEL` Delete the character to the left of the cursor.

`C-d` Delete the character underneath the cursor.

Printing characters Insert itself into the line at the cursor.

`C-_` Undo the last thing that you did. You can undo all the way back to an empty line.

B.2.2 Readline Movement Commands

The above table describes the most basic possible keystrokes that you need in order to do editing of the input line. For your convenience, many other commands have been added in addition to `C-b`, `C-f`, `C-d`, and `DEL`. Here are some commands for moving more rapidly about the line.

`C-a` Move to the start of the line.

`C-e` Move to the end of the line.

`M-f` Move forward a word.

`M-b` Move backward a word.

`C-l` Clear the screen, reprinting the current line at the top.

Notice how `C-f` moves forward a character, while `M-f` moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

B.2.3 Readline Killing Commands

The act of **cutting** text means to delete the text from the line, and to save away the deleted text for later use, just as if you had cut the text out of the line with a pair of scissors.

Killing text means to delete the text from the line, but to save it away for later use, usually by **yanking** it back into the line. If the description for a command says that it 'kills' text, then you can be sure that you can get the text back in a different (or the same) place later.

Here is the list of commands for killing text.

`C-k` Kill the text from the current cursor position to the end of the line.

`M-d` Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`M-DEL` Kill from the cursor the start ofthe previous word, or if between words, to the start of the previous word.

`C-w` Kill from the cursor to the previous whitespace. This is different than `M-DEL` because the word boundaries differ.

And, here is how to **yank** the text back into the line. Yanking is

`C-y` Yank the most recently killed text back into the buffer at the cursor.

`M-y` Rotate the kill-ring, and yank the new top. You can only do this if the prior command is `C-y` or `M-y`.

When you use a kill command, the text is saved in a **kill-ring**. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it in one clean sweep. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

B.2.4 Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might type `M-- C-k`.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first 'digit' you type is a minus sign (-), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the `C-d` command an argument of 10, you could type `M-1 0 C-d`.

B.3 Readline Init File

Although the Readline library comes with a set of Emacs-like keybindings, it is possible that you would like to use a different set of keybindings. You can customize programs that use Readline by putting commands in an **init** file in your home directory. The name of this file is ' */.inputrc*'.

When a program which uses the Readline library starts up, the ' */.inputrc*' file is read, and the keybindings are set.

B.3.1 Readline Init Syntax

You can start up with a vi-like editing mode by placing

```
set editing-mode vi
```

in your ' */.inputrc*' file.

You can have Readline use a single line for display, scrolling the input between the two edges of the screen by placing

```
set horizontal-scroll-mode On
```

in your ' */.inputrc*' file.

The syntax for controlling keybindings in the ' */.inputrc*' file is simple. First you have to know the *name* of the command that you want to change. The following pages contain tables of the command name, the default keybinding, and a short description of what the command does.

Once you know the name of the command, simply place the name of the key you wish to bind the command to, a colon, and then the name of the command on a line in the ' */.inputrc*' file. Here is an example:

```
# This is a comment line.
Meta-Rubout:  backward-kill-word
Control-u:  universal-argument
```

Commands For Moving

`beginning-of-line (C-a)` Move to the start of the current line.

`end-of-line (C-e)` Move to the end of the line.

`forward-char (C-f)` Move forward a character.

`backward-char (C-b)` Move back a character.

`forward-word (M-f)` Move forward to the end of the next word.

`backward-word (M-b)` Move back to the start of this, or the previous, word.

`clear-screen (C-l)` Clear the screen leaving the current line at the top of the screen.

Commands For Manipulating The History

`accept-line (Newline, Return)` Accept the line regardless of where the cursor is. If this line is non-empty, add it too the history list. If this line was a history line, then restore the history line to its original state.

`previous-history (C-p)` Move 'up' through the history list.

`next-history (C-n)` Move 'down' through the history list.

`beginning-of-history (M-<)` Move to the first line in the history.

`end-of-history (M->)` Move to the end of the input history, i.e., the line you are entering!

`reverse-search-history (C-r)` Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.

`forward-search-history (C-s)` Search forward starting at the current line and moving 'down' through the the history as necessary.

Commands For Changing Text

`delete-char (C-d)` Delete the character under the cursor. If the cursor is at the beginning of the line, and there are no characters in the line, and the last character typed was not C-d, then return EOF.

`backward-delete-char (Rubout)` Delete the character behind the cursor. A numeric arg says to kill the characters instead of deleting them.

`quoted-insert (C-q, C-v)` Add the next character that you type to the line verbatim. This is how to insert things like C-q for example.

`tab-insert (M-TAB)` Insert a tab character.

`self-insert (a, b, A, 1, !, ...)` Insert yourself.

`transpose-chars (C-t)` Drag the character before point forward over the character at point. Point moves forward as well. If point is at the end of the line, then transpose the two characters before point. Negative args don't work.

`transpose-words (M-t)` Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

`upcase-word (M-u)` Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move point.

`downcase-word (M-l)` Lowercase the current (or following) word. With a negative argument, do the previous word, but do not move point.

`capitalize-word (M-c)` Uppercase the current (or following) word. With a negative argument, do the previous word, but do not move point.

Killing And Yanking

`kill-line (C-k)` Kill the text from the current cursor position to the end of the line.

`backward-kill-line ()` Kill backward to the beginning of the line. This is normally unbound.

`kill-word (M-d)` Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`backward-kill-word (M-DEL)` Kill the word behind the cursor.

`unix-line-discard (C-u)` Do what C-u used to do in Unix line input. We save the killed text on the kill-ring, though.

`unix-word-rubout (C-w)` Do what C-w used to do in Unix line input. The killed text is saved on the kill-ring. This is different than backward-kill-word because the word boundaries differ.

`yank (C-y)` Yank the top of the kill ring into the buffer at point.

`yank-pop (M-y)` Rotate the kill-ring, and yank the new top. You can only do this if the prior command is yank or yank-pop.

Specifying Numeric Arguments

`digit-argument (M-0, M-1, ... M--)` Add this digit to the argument already accumulating, or start a new argument. M– starts a negative argument.

`universal-argument ()` Do what C-u does in emacs. By default, this is not bound.

Letting Readline Type For You

`complete (TAB)` Attempt to do completion on the text before point. This is implementation defined. Generally, if you are typing a filename argument, you can do filename completion; if you are typing a command, you can do command completion, if you are typing in a symbol to GDB, you can do symbol name completion, if you are typing in a variable to Bash, you can do variable name completion...

`possible-completions (M-?)` List the possible completions of the text before point.

Some Miscellaneous Commands

`abort (C-g)` Ding! Stops things.

`do-uppercase-version (M-a, M-b, ...)` Run the command that is bound to your uppercase brother.

`prefix-meta (ESC)` Make the next character that you type be metafied. This is for people without a meta key. `ESC-f` is equivalent to `M-f`.

`undo (C-_)` Incremental undo, separately remembered for each line.

`revert-line (M-r)` Undo all changes made to this line. This is like typing the 'undo' command enough times to get back to the beginning.

B.3.2 Readline Vi Mode

While the Readline library does not have a full set of Vi editing functions, it does contain enough to allow simple editing of the line.

In order to switch interactively between Emacs and Vi editing modes, use the command M-C-j (toggle-editing-mode).

When you enter a line in Vi mode, you are already placed in 'insertion' mode, as if you had typed an 'i'. Pressing `ESC` switches you into 'edit' mode, where you can edit the text of the line with the standard Vi movement keys, move to previous history lines with 'k', and following lines with 'j', and so forth.