DISCUS
Package

Reference Guide

developed by

Reinhard Neder
Email: reinhard.neder@fau.de

Thomas Proffen
Email: tproffen@ornl.gov

http://discus.sourceforge.net/

Document created: November 9, 2017

Contents

Welcome

Congratulations for downloading the DISCUS package. The DISCUS package consists of the following programs:

- **DISCUS SUITE**: The DISCUS SUITE is a common program that includes DISCUS DIFFEV and KUPLOT. It allows to switch seamlessly between the individual sections and will eventually replace the individual parts. It also offers an optimization for large scale computing facilities.
- **DISCUS**: This is the main diffuse scattering and defect structure simulation program. This is the program that gave the package its name.
- **DIFFEV**: This is a more recent addition. This program allows the refinement of a set of high level parameters (*e.g.* input parameters of a DISCUS macro) based on a differential evolutionary algorithm.
- **MIXSCAT**: This is newest addition to the DISCUS package. This program allows on to extract differential PDFs from neutron- and x-ray data.
- **KUPLOT**: This is plotting program of the package. It is nearly as old as DISCUS itself.

Each of the programs has its own users guide which include disclaimers and the appropriate references to cite in case you are publishing work derived from using these programs.
This manual contains a description of the command language common to all programs as well as detailed installation instructions. DISCUS itself has been developing over the last nearly 20 years and the command language was developed before anyone knew about things like python. The commands are loosely related to FORTRAN77 syntax, so the older generation should have no trouble with it.

Please visit the DISCUS homepage frequently at `http://tproffen.github.io/DiffuseCode/` and also consider subscribing to the DISCUS mailing lists.

0.1 Getting started

0.1.1 Windows

Once the programs have been installed you should see an icon for each of the programs on your personal desktop. Double click the icon to start the program of your choice.

At program start, each of the programs sets its starting folder to

```
C:\Users\your_name
```

where `your_name` will be your user name. The string `Users` may be slightly different depending on the language settings. To work with data and macros that are stored in a separate folder, you need to change to this folder.
Within the DIFFUSE program window, type the command `cd` including a blank space after the `cd`. At this point do not hit the enter key. Open the desired folder with the Windows-Explorer. Left click on the small folder symbol in the top line that indicates the path to your folder. This should highlight the full path to the folder.
For DISCUS, DIFFEV, and MIXSCAT you can drag this folder symbol into the DIFFUSE program window. The command line should now read something like

```
cd C:\Users\Neder\Documents\DISCUS_examples
```

For KUPLOT and DISCUS SUITE, select the highlighted path to the folder with CTRL-c. Activate the KUPLOT or DISCUS SUITE window and press the middle button on the mouse. This should place the full path into the program window.
Once you activate the window and hit the ENTER key the program will work in this folder.

0.1.2 Linux

Once you have installed the Diffuse program package, the binaries will usually reside in the directory: `/usr/local/bin/`. To use any of the programs, open a terminal window, switch to the desired directory and type the program name.

Chapter 1

FORTRAN style interpreter

The programs include a FORTRAN style interpreter that allows the user to program complex modifications. The interpreter provides variables, linked to data sets and free variables, loops, logical construction, basic arithmetic and built in functions. Commands related to the FORTRAN interpreter are `=`, `break`, `do`, `else`, `elseif`, `enddo`, `endif`, `eval`, `if` and `variable`. The command `eval` allows to examine the contents of a variable or to evaluate an expression, e.g. `eval r[1]*0.5`.

1.1 Variables

The programs in the *DISCUS* suite use variables to store data. The programs distinguish three types of variables:

- Variables with fixed name for general use
- Variables with user defined names for general use
- Variables with fixed name that carry program information

Variables with fixed name for general use
There are two types of variables: variables of type integer or real, denoted by their fixed names "i" and "r", immediately followed by a left square bracket [, one or more indices (separated by a comma), and finally a right square bracket].
Examples for build in variables are given next.

Example : i[1], r[0], i[i[1]], x[2]

The build-in variables `i[n]` and `r[n]` are general purpose variables. The index `n` can take integer valus from zero to a maximum value defined at program compilation, usually 500. In addition, each of the program has a number of build-in variables related to its function. DISCUS for example has a number of variables related to the structure that the user is using, like unit cell dimensions, atom positions etc. For a list of these variables, please refer to the respective program users guides.
Variables with user defined names for general use
The second set of variables are single valued variable with user defined names. These allow

the user to use more obvious variable names. The programs allow to define real valued, integer valued and character variables.
An example of the use of defined variables is shown here:

```
1  variable real,alpha,90.0
2  variable real,beta
3  variable real,diff
4  beta = 94.0
5  diff = alpha - beta
6  eval diff
```

In line one we define a real variable `alpha` with an initial value of 90. Next, two more variables are defined and the difference between `alpha` and `beta` is calculated (line 5) and the result displayed on the screen (line 6).
Recently another type of variables was added to the command language: character variables. These can be used in conjunction with format statements (see command reference under *expressions*). An example setting the plot title in KUPLOT to the current date is listed below.

```
1  variable character,datum
2  #
3  datum="%c",fdate(0)
4  tit2 "Plotted on %c",datum
```

Note the slightly different way character intrinsic functions are used. A complete list of character functions can be found in Table 1.3.
Variables with predefined name

Starting with version 5.7.0 system wide variables with fixed name were introduced. Currently these are mostly related to the refinement, see the DIFFEV manual for details.
One further sysmten variable is pi.
Variables with fixed name that carry program information
These variables, which are common to the whole program suite are used by the programs to return information to the user.
Currently just one such variable `res[n]` exists. It is used in a wide variety of commands and functions to return a successful operation or to contain result values.
As an example see:

```
1  fexist dummy.file
2  eval res[0]
2  eval res[1]
```

Here the `res` variable is used to flag the existence of the file called "dummy.file". `res[0]` usually carries the number of results returned, in this special case one. Here `res[1]` would be equal to one if the file exists and zero if it does not exist.

1.2 Arithmetic expressions

The language allows the use of arithmetic expressions using the same notation as in FORTRAN. Valid operators are `+, -, *, /` and `**`. Expressions can be grouped by round brackets (and). The usual hierarchy for the operators holds. Values of expressions can be assigned to any modifiable variable. If you know FORTRAN (or another programming language) you will have no problems with these examples.

```
i[0] = 1
r[3] = 3.1415
r[i[1]] = 2.0*(i[5]-5.0/6.5)
```

1.3 Logical expressions

Logical expressions are formed similar to FORTRAN. They may contain numerical comparisons using the syntax: *<arithmetic expression><operator><arithmetic expression>*. The allowed operators are `.lt., .le., .gt., .ge., .eq.` and `.ne.` for operations less than, less equal, greater than, greater equal, equal and not equal, respectively. Logical expressions can be combined by the logical operators `.not., .and., .or., .xor.` and `.eqv.`. The following example shows an expression that is true for values of `i[1]` within the interval of 3 and 11, false otherwise.

```
i[1].ge.3 .and. i[1].le.11
```

Logical operations may be nested and grouped using round brackets (and). For more examples see section 1.6.

1.4 Intrinsic functions

Several intrinsic functions are defined. Each function is referenced, as in FORTRAN, by its name followed by a pair of parentheses (and) that include the list of arguments. The (does not have to immediately follow the function name. Trigonometric and arithmetic functions are listed in table 1.1. Table 1.2 contains various random number generating functions. Some programs such as DISCUS have a set of specific intrinsic functions allowing one to for example calculate bond length. Again each individual users guide will contain a table of these specific functions.
Finally the available functions returning a character string are listed in Table 1.3.

1.5 Loops

Loops can be programmed using the `do` command. Three different types of loops are implemented. The first type executes a predefined number of times. The syntax for this type of loop is

do <variable> = <start>,<end> [,<increment>]
. . . commands to be executed . . .
enddo

Loops may contain constants or arithmetic expressions for <start>, <end>, and <increment>. <increment> defaults to 1. The internal type of the variables is real. The loop counter is evaluated from *(<end> - <start>)/<increment> + 1*. If this is negative, the loop is not executed at all. The parameters for the counter variable, start end and increment variables are evaluated

Type	Name	Description
real	sin(r) cos(r) tan(r)	Sine, cosine and tangent of <r> in radian
real	sind(r) cosd(r) tand(r)	Sine, cosine and tangent of <r> in degrees
real	asin(r) acos(r) atan(r)	Arc sin, cosine, tangent of <r>, result in radian
real	asind(r) acosd(r) atand(r)	Arc sin, cosine, tangent of <r>, result in degrees
real	sqrt(r)	Square root of <r>
real	exp(r)	Exponential of <r>, base e
real	ln(r)	Logarithm of <r>
real	sinh(r) cosh(r) tanh(r)	Hyperbolic sine, cosine and tangent of <r>
real	abs(r)	Absolute value of <r>
integer	mod(r1, r2)	Modulo <r1> of <r2>
integer	int(r)	Convert <r> to integer
integer	nint(r)	Convert <r> to nearest integer
real	frac(r)	Returns fractional part of <r>
real	min(r1, r2)	Smaller number of <r1> and <r2>
real	max(r1, r2)	Larger number of <r1> and <r2>

Table 1.1: Trigonometric and arithmetic functions

Type	Name	Description
real	ran(r)	Uniformly distributed pseudo random number between 0.0 inclusively and 1.0 exclusively. Argument <r> is a dummy
real	gran(r1, typ)	Gaussian distributed random number with mean 0 and a width given by <r1>. If <typ> is "s" <r1> is taken as sigma, if <typ> is "f" <r1> is taken as FWHM.
real	gbox(r1, r2, r3)	Returns pseudo random number with distribution given by a box centered at 0 with a width of <r2> and two half Gaussian distributions with individual sigmas of <r1> and <r3> to the left and right, respectively.
real	gskew(r1,typ)	Skewed Gaussian distributed random number with mean
real	logn(r1,r2)	Returns a lognormal distributed number with mean <r1> and sigma <r2> of the underlying Gaussian distribution.
integer	pois(r1)	Returns a poisson distributed random number with mean <r1>.

Table 1.2: Random number functions

only at the beginning of the do - loop and stored in internal variables. It is possible to change the values of <variable>, <start> and/or <end> within the loop without any effect on the performance of the loop. This practice is not encouraged, could, however, be an unexpected source of errors.
The second type of loop is executed while <logical expression> is true. Thus it might not be executed at all. The syntax for this type of loop is

do while <logical expression>
. . . commands to be executed . . .

Name	Description
date(0)	Returns current date as `CCYYMMDDhhmmss.sss`.
fdate(0)	Returns current date as charater string `Day Mon dd hh:mm.ss yyyy`.
fmodt(0)	Returns modification date of the last file opened as charater string `Day Mon dd hh:mm.ss yyyy`.
cdate(0)	Returns current date as `Day Mon DD hh:mm:ss CCYY`.
getcwd(0)	Returns the current working directory.
getenv('VAR')	Returns the value of the environment variable `VAR`.

Table 1.3: String functions

enddo

The last type of loop is executed until <logical expression> is true. This loop, however, is always executed once and has the following syntax

do
. . . commands to be executed . . .
enddo until <logical expression>

In the body of commands any valid program commands can be used. This includes calls to the sublevels, further do loops or macros, even if these macros contain do loops themselves. The maximum level of nesting is limited by the parameter *MAXLEV* in the file *doloop_mod.f90*. If necessary adjust this parameter to allow for deeper nesting. All commands from the first `do` command to the corresponding 'enddo' are read and stored in an internal array. This array can take at most *MAXCOM* (defined in file *doloop_mod.f90* as well) commands at every level of nesting. If lengthy macro files are included in the do loop, this parameter might have to be adjusted.
If a do loop (or an if block) needs to be terminated, the `break` command will perform this function. The parameter on the `break'` command line gives the number of nested levels of `do` and `if` blocks to be terminated. The interpreter will continue execution with the first command following the corresponding `enddo` or `endif` command. An example is given below, note, that the line numbers are only given for better orientation and are no actual part of the listed commands.

```
1  do i[2]=1,5
2     do i[1]=1,5
3        if ((i[1]+i[2]) .eq 6) then
4           break 2
5        endif
6     enddo
7  enddo
```

In this example, the execution of the inner do-loop will stop as soon as the sum of the two increment variables `i[1]` and `i[2]` is equal to 6. The program continues with the `enddo` line of the outer do - loop. Notice that two levels need to be interrupted, the if block and the innermost do loop. If the parameter had been equal to one, only the if block would have been interrupted, while the innermost do loop would have continued without break.

1.6 Conditional statements

Commands can be executed conditionally by using the `if` command. Analogous to FORTRAN, the if-control structure takes the following form:

if (<logical expression>) then
. . . commands to be executed . . .
elseif (<logical expression>) then
. . . commands to be executed . . .
else
. . . commands to be executed . . .
endif

The logical expressions are explained in section 1.3. Enclosed within an if block any valid program command can be used. This includes calls to the sublevels further if blocks, do loops or macros, even if these macros contain if blocks or do loops themselves. The `elseif` and `else` section is optional. The maximum level of nesting is limited by the parameter *MAXLEV* in the file *doloop_mod.f90*. If necessary adjust this parameter to allow for deeper nesting. All commands from the first `if` command to the corresponding `endif` are read and stored in an internal array. This array can take at most *MAXCOM* (defined in file *doloop_mod.f90* as well) commands at every level of nesting. If lengthy macro files are included in the do loop, this parameter might have to be adjusted.
If an if block (or a do loop) needs to be terminated, the `break` command will perform this function. The parameter on the `break` command line gives the number of nested levels of `do` and `if` blocks to be terminated. The interpreter will continue execution with the first command following the corresponding `enddo` or `endif` command. See the example in section 1.5 for further explanations.

```
 1  #
 2  # Read crystal file
 3  #
 4  read
 5  cell cell.cll,10,10,10
 6  #
 7  # Remove atoms with probability 0.3
 8  #
 9  do i[0]=1,n[1]
10     if(ran(0).lt.0.3) then
11        remove i[0]
12     endif
13  enddo
```

The example listed above illustrates the use of loops and conditional statements within *DISCUS*. Again, the line number are given for easy reference and not part of the actual input. The first three lines are just comments. In lines 4 and 5 an asymmetric unit is read from the file *cell.cll* and expanded to a crystal size of 10x10x10 unit cells. In line 9 starts a do-loop over all atoms within the crystal. The variable n[1] contains this information (see DISCUS users guide). Since the function `ran` produces a uniformly distributed pseudo random number in the range 0.0 to 1.0, the if statement in line 10 is true in about 30% of its calls, at least for sufficiently large crystal sizes. Thus approximately 30% of the atoms are removed (line 11), and the corresponding amount of vacancies (VOID) created within the crystal.

1.7 Filenames

Usually, file names are understood as typed, including capital letters. Unix operating systems distinguish between upper and lower case typing ! However, sometimes it is required to be able to alter a file name e.g. within a loop. Thus, the command language allows the user to construct file names by writing additional (integer) numerical input into the filename. The syntax for this is:

"string%dstring",<integer expression>

The file format MUST be enclosed in quotation marks. The position of each integer must be characterized by a `%d`. The sequence of strings and '%d's can be mixed at will. The corresponding integer expressions must follow after the closing quotation mark. If the command line requires further parameters (like `addfile` for example) they must be given after the format-parameters. The interpretation of the '%d's follows the C syntax. Up to 10 numbers can be written into a filename. All of the following examples will result in the file name *a1.1*:

```
i[5]=1
outfile a1.1
outfile "a%d.%d",1,1
outfile "a%d.%d",4-3,i[5]
```

The second example shows how filenames are changed within a loop. Here the output (e.g. Fourier transform) will be written to the files *data1.calc* to *data11.calc*.

```
do i[1]=1,11
  ..
  outfile "data%d.calc",i[1]
  ..
enddo
```

As personal style you might find it best to label the files *data01.calc* to *data11.calc* i.e. with leading zeros. This is readily achived with the `%D` format specifier.

1.8 Macros

Any list of valid program commands can be written to an ASCII file and executed indirectly by the command `@<filename>`. The commands are executed as typed. Macro files can be written by any editor on your system or be generated by the 'learn' command. 'learn' starts to remember all the commands that follow and saves them into the file given on the `learn` command. The learn sequence is terminated by the `lend` command. The default extension of the macro file is *.mac*. Macro files can be nested and even reference themselves directly or indirectly. This referencing of macro files is, however, just a nesting of the corresponding text of each macro, not a call to a function. All variable retain their values. If an error occurs while executing a macro, the program immediately stops execution of all macros and returns to the interactive prompt. If the macro switched to a sublevel, and the error occurred inside of this sublevel, the program will remain within this sublevel the interactive prompt corresponding to this sublevel is returned. The command `stop` allows the user to interrupt the execution of

a macro, enter commands and continue the macro using the command `cont`. Note, that the macro needs to continued in the same sublevel it was interrupted.
On the command line of the macro command `@`, optional parameters can be supplied. Within the macro these have to be referenced as $1, $2 etc. Upon execution of the macro the formal parameters $n are replaced by the character string of the actual values from the command line. Parameter $0 contains the number of parameters specified on the command line. As any other command parameters, these parameters must be separated by comma. If a formal parameter is referenced inside a macro without a corresponding parameter on the command line, an error message is given. An example is given below:

```
# Adds two numbers supplied as command line parameters.
# The value is stored in variable defined by parameter three
#
$3 = $1 + $2
eval $3
```

If this macro is called with the following line, `@add 1,2,i[4]`, the result is stored in variable i[4] which now has the integer value 3.
If the program is started with command line parameters, e.g. `discus 1.mac 2.mac`, the program will execute the given macros in the specified order, in our example first `1.mac` then `2.mac`. You cannot, however, provide parameters to these macros.
Alternatively a single macro can be executed by starting the program with the comand line option `–macro macro_file_name parmeter(s)`
If a macro is not found in the current working directory, a system macro director is searched. This system macro directory is located at `path_to_binary/sysmac/discus/`. Commonly used macro files might be installed in this directory. If a macro file is not found, an error message is diplayed.

1.9 Working with files

The command language offers the user several commands to write variables to a file or read values from a file. First a file needs to be opened using the command `fopen`. An optional parameter `append` allows one to append data to an existing file. Once the task is finished, the file must be closed via `fclose`. In the standard configuration, the program can open five files at the same time. The first parameter of all file input/output related commands is the unit number which can range from 1 to 5. The commands `fget` and `fput` are used to read and write data, respectively. The following example illustrates the usage of these commands:

```
 1  fopen 1,sin.dat
 2  fput 1,'Cool sinus function'
 3  #
 4  do i[1]=1,50
 5     r[1]=i[1]*0.1
 6     r[2]=sin(r[1])
 7     fput 1,r[1],r[2]
 8  enddo
 9  #
10  fclose 1
```

In line 1 we open the file *sin.dat* and write a title (line 2). If the file already exists it will be overwritten. Note that the text must be given in *single* quotes. Text and variables may be mixed

in a single line. Next we have a loop calculating $y = \sin(x)$ and writing the resulting x and y values to the open file (line 7). Finally the file is closed (line 10). To read values from a file use simply the command `fget` and the read numbers will be stored in the specified variables. In contrast to writing to a file, mixing of text and number is not allowed when reading data. However, complete lines will be skipped when the command `fget` is entered without any parameters.

1.10 Remote control

It is possible to remote control the programs in the DISCUS package using so called sockets. One program acts as server and receives and executes commands. In order to enable the server feature, the program needs to be started using the `-remote` command line switch as in the example below:

```
dhcp165057:prog> ./discus -remote

          ************************************************************
          *               D I S C U S   Version 3.6.1                *
          *                                                          *
          *         Created : Fri Jan 15 13:27:43 JST 2010           *
          *----------------------------------------------------------*
          * (c) R.B. Neder  (reinhard.neder@krist.uni-erlangen.de)   *
          *     Th. Proffen (tproffen@lanl.gov)                      *
          ************************************************************
          *                                                          *
          * For information on current changes type: help News       *
          *                                                          *
          ************************************************************

 Command line editing enabled ..

 User macros in   : /Users/thomasproffen/mac/discus/
 System macros in : /Users/thomasproffen/mac/discus/
 Start directory  : /Users/thomasproffen/Code/Diffuse/discus/prog

 ------ > Running in SERVER mode
 Running in local mode (127.0.0.1) ..
 Listening to port 3330 ..
 Allowing connections from 127.0.0.1 ..
```

Note that the host IP address and port number are given at the end of the startup output. This information is needed to connect to this running version of in our case DISCUS. To allow connections from computers other than 127.0.0.1 (localhost) or using a different port, use the command line options `-access` and `-port`.
In order to connect to the DISCUS server, we use the command `socket` as in this example:

```
 discus > socket open,127.0.0.1,3330
 Connecting to 127.0.0.1:3330 ..
 Server : ready
 Connected ..
 discus > socket send,echo This is cool
 Server :  This is cool
 Server : ready
 discus >
```

Of course any program or script that can use sockets is able to connect to the DISCUS package programs in this way. For more details refer to the `socket` command reference later in this guide.

Chapter 2

Installation

In this section we will describe the installation process for the DISCUS program package. The current version of the software can be downloaded from the DISCUS homepage at `https://github.com/tproffen/DiffuseCode/`. Refer to the section corresponding to your operating system for installation information.
At the github release site you will find installation guides for DISCUS for Unix, MacOS and Cygwin. The Windows installation is performed via a self extracting installer.

Windows

The Windows version of the DISCUS package is distributed as a self-extracting installer. This makes the installation very easy. Simply download the file *Diffuse-X.X.X-win64-YYMMDD.exe* or if necessary *Diffuse-X.X.X-win32-YYMMDD.exe*. Here YYMMDD specifies the date of the distribution and. X.X.X stands for the version number. Make sure you download the most recent one. Run the installer by double clicking on the corresponding file icon. You will first receive a Windows security alert, because the installer is not digitally signed by us. Once we get around to figure out how, this warning will go away. For now, just click on *Run*. This will start the installation process itself and the installation dialog will show up. Follow the instructions on the screen and that is all, you are ready to use any of the programs that are part of the DISCUS package. Look in the *START - Programs* menu for links to the programs as well as the documentation.
As you may noticed, we have changed the way the installer is build. For that reason, you need to **uninstall any DISCUS version prior to 2010 before proceeding with the installation.** The current installer works fine on *Windows XP, Windows Vista Windows7* and *Windows 10*.

DISCUS and CYGWIN

The DISCUS package for Windows is developed using the *CYGWIN* package (`http://www.cygwin.com`) which provides a UNIX like environment for Windows. We recommend installing the *CYGWIN* 32bit or 64 bit package to be used with the DISCUS package, although this is not required. One side effect of the use of *CYGWIN* is that one needs to specify UNIX style paths. Also you might see that for example drive C: is refered to as `/cygdrive/c/`.

Another side effect is the file format for all ASCII or text files like macros, cell files, diffraction pattern output etc. Unfortunately UNIX and Windows use a different encoding to signal the end of a line for such file types. Since we use *cygwin*, the file format is UNIX style. As a consequence, the Windows *Editor* usually found under *All programs* in the *Accessories* section cannot handle such file types. Please use a more advanced text editor like *Notepad+* or *WordPad* instead to edit these files. If you installed *CYGWIN*, you can use the programs *unix2dos* and *dos2unix* to convert file formats.

Mac OSX

The Mac OSX version of the DISCUS package is distributed as a binary installer as well. Simply download the the file *Diffuse-mac-YYMMDD.exe*. Again YYMMDD specifies the date of the distribution. Make sure you download the most recent one. Once the download is finished, run the installer by double clicking on the corresponding file icon. You will see an installation screen. Follow the instructions. The programs will be installed in `/Applications/discus`.

Unix / Linux

For Unix or Linux operating systems, the DISCUS package is distributed as source code and needs to be compiled before the programs can be used. You might also check the DISCUS homepage for available binary distributions for Linux which might be available in the near future. Refer to the separate file `INSTALL_DISCUS.pdf` or `AAA_INSTALL_DISCUS.pdf` for details.
To build the programs from the source, you will need a FORTRAN and a C compiler. We use `gcc` and `gfortran` which are freely available at `http://directory.fsf.org/project/gcc/`. While `gcc` is installed on most Linux systems, `gfortran` might need to be installed separately.
First copy or download the file *DIFFUSE_CODE_YYYY_MMDD.tar.gz* or from the github release pages the link to *vX.X.X.tar.gz*. Next unpack the archive using the command

```
tar -xvf DIFFUSE_CODE_YYYY_MMDD.tar.gz
```

This will create a directory *DiffuseCode*, containing the distribution. Within this directory there are separate directories for each of the different programs as well as a directory *lib_f90* which contains command language related routines common to all programs. Build a a new Directory called *DiffuseBuild* next to the source code directory. Go to this build directory and run ccmake to install the program:
Finally some environment variables need to be defined. Each program looks for a variable corresponding to its name. For example DISCUS will use a variable `DISCUS` and so on. The definition of the variables can be done e.g. in the *.login* or *.cshrc* file using the command *setenv DISCUS /path/to/discus* for the *csh* of *set DISCUS=/path/to/discus; export DISCUS* if you are using the bourne shell. If this path is also included in your search path you can start the program simply by entering *discus*. Similarly, the other programs are started by entering their respective names.

Chapter 3

COMMON commands

3.1 News

Here you find a list of recent changes, additions, bug corrections

3.2 2017_Sep

Throughout the program the internal calculation of random numbers was changed to the FORTRAN 90 intrinsic function.

3.3 2017_July

Predefined variables REF_* are now read/write. See ==> variable
Introduced new intrinsic character functions: index and length See help under functions for details.

3.4 2016_October

New system variables have been introduced. They can be used like any other user defined variable. System variables are in capital letters.

3.5 2014November

A new random number "gskew" has been added, which returns a Gaussian distributed random number. The underlying distribution can be set to be is left or right skewed.

3.6 Options

program [-debug] [-remote] [-port=p] [-access=ip] [macro.mac]
program -macro <macro.mac>[<par1> [<par2> ...]]

All programs allow the following command line parameters. The flag "-debug" starts the program in debug mode. This is the same as using the command "set debug,on". The switch "-remote" starts the program in remote control mode. In this case the commands are send to the program through a socket. In this case the switch "-port' allows one to specify the port, the program will be listening on. Port numbers should be larger than 1024. The switch "-access' allows one to specify from which host connections will be accepted. The default is 'localhost'.
See file "remote.f" for an example how to remote control the applications from another program.
Note that the program will not accept input from the keyboard when in remote control mode.
All other command line arguments are interpreted as macro files and will be executed at startup. These macros may not rely on parameters to be given on the command line.
If a macro is to be executed that takes 1 or more parameters, use the "-macro" option. Note that this option is mutually exclusive to all other options. The first command line argument after the '-macro' option it the macro name, all further optional command line arguments are taken as macro parameters. These have to be seperated by one or more spaces. Parameters that need to contain spaces must be enclosed in single or double quotation marks. -macro test.mac 1 2 3 This is the same as @test.mac 1,2,3
-macro test1.mac '1 + 2 + 3' This is the same as @test1.mac 1 + 2 + 3

3.7

#<comment>
Any line beginning with a "#" is regarded as comment.

3.8 @

@<filename> [<argument> ...]
Any list of valid commands can be written to an ASCII file and and indirectly by the command:

```
prompt > @<name>
```

The commands may start in with leading blanks to help readability of the macro file. The commands are executed as typed. Macro files may call other macro files. This is not a call in the sense of calling a function. All variables are identical at all levels of macro file nesting.
Macro files can be written by any editor on your system or be generated by the ==> 'learn' command. 'learn' starts to remember all the commands that follow and saves them into the file given on the 'learn' command. The learn sequence is terminated by the 'lend' command. The default extension is ".mac"
Optionally arguments can be listed on the command line. These arguments will replace the formal parameters inside the macro. The formal parameters must be given as "$1", "$2" ... The string <argument> will replace the string "$1". "$1" is the first argument on the command line, "$2" the second and so on. If there are not enough command line arguments, an error message is displayed. The parameter "$0" contains the number of parameters listed on the line that called the macro. If no parameters were given this value will be zero.
The prompt setting ==> 'set prompt,"redirect"' has an important side effect on macro treatment. With the "redirect" setting, macros are stored internally, once they have been read from disk,

and will be reused from memory. This helps to reduce unnecessary I/O, especially when you have nested macros inside loops. As a side effect, if a macro is modified on the disk, a further "@macro.mac" will not read the modified version but will continue to use the internaly stored version.
For all other settings, the internal macro storage is cleared when you get back to the normal interactive mode. This allows you to run a macro, then modify the version stored on the disk and execute the modified/corrected version.

3.9 =

<variable> = <expression>

The expression on the right of the equal sign is evaluated and its result stored in variable <variable>.

3.10 input

Input editing functions

If the program was compiled with -DREADLINE, the following basic editing functions are available at the program prompt:

```
^A                : moves to the beginning of the line
^B                : moves back a single character
^E                : moves to the end of the line
^F                : moves forward a single character
^K                : kills from current position to the end of line
^P or arrow up    : moves back through history
^N or arrow down  : moves forward through history
^H and DEL        : delete the previous character
^D                : deletes the current character
^L/^R             : redraw line in case it gets trashed
^U                : kills the entire line
^W                : kills last word
```

Furthermore you can move within the line using the arrow keys.
NOTE:

If you redirect the input for executed PROG using 'prog < infile' you MUST use the command 'set prompt,off' or 'set prompt,redirect' in the first line to avoid that the program 'hangs' at the end of the file. (-> set prompt)

3.11 break

break <levels>

The 'break' command stops the execution of the current block structure and advances to the next command following the block structure. With <levels> equal to 1 only the current block structure is interrupted, with any higher number the <levels> innermost block structures are interrupted. The 'break' command can be used only inside a block structure.

3.12 cd

cd [<directory>]

This command allows one to change the current working directory (may not be available everywhere). If the command is called with no parameters, the current working directory is shown.
For the Windows versions, two different styles help to copy the folder name into the program window.
For KUPLOT and the DISCUS_SUITE type cd and a space. Do not hit the enter/return key at this moment. Within a Windos Explorer click on the folder icon and copy the string CTRL-c. Activate the KUPLOT or DISCUS_SUITE program and click the middle mouse button. This should paste the full path to the folder into the KUPLOT or DISCUS_SUITE window.
For DISCUS, DIFFEV, and MIXSCAT type "cd " with the space as well. Now left click and drag the selected folder icon into the program window. This should copy the full pathinto the program window. Activate the window and hit the ENTER key.

3.13 continue

continue ["prog"]

This command is effective only while PROG is in the interrupted macro mode or inside interrupted do-loop or if-statements, which serves as a debug mode for lengthy macros or block structures Make sure you have returned to the same sub menu before you continue!
Without parameters PROG resumes the execution of a macro or block structure in the line following the 'stop' command. If you had started another macro while debugging a macro, and this new macro contained a 'stop' command as well, the 'continue' command will run the remaining lines in the new macro and then stop again at the position of the 'stop' command in the outer macro.
By providing the 'prog' parameter, PROG immediately interrupts all macros and returns to the normal prompt. If you are in one of the sub sections "discus", "diffev", "kuplot", you can continue either with this subsection or go back to the main suite if you enter the program name as "suite".

3.14 do

Loops can be programmed with the 'do' command. The command may take the following forms:

```
do <variable> = <start>,<end> [,<increment>]
  <commands to be repeated>
enddo
```

Here loops may contain constants or arithmetic expressions for <start>, <end>, and <increment>. The internal type of the variables is real. The loop counter is evaluated from (<end> - <start>) / <increment> =1 . If this is negative, the loop is not executed at all.

```
do while (<logical expression>)
  <commands to be repeated>
enddo
```

These loops are executed while <logical expression> is true. Thus, they may not be executed at all.

```
do
  <commands to be repeated>
enddo until (<logical expression>)
```

These loops, however, are always executed once, and repeated until <logical expression> is true. If an error occurs during execution of the loop, the loop is interrupted.

3.15 echo

```
echo [<string>]
echo ["string%dstring",<integer expression>]
echo ["string%Dstring",<integer expression>]
echo ["string%fstring",<float expression>]
echo ["string%Fstring",<float expression>]
echo ["string%cstring",<character expression>]
```

The string <string> is echoed to the default output device as typed. This command serves as a marker inside long macro files. It gives the user a chance to include easy to find messages in order to follow lengthy or nested structures.
The alternative command format allows to echo formatted strings to the screen. Each "%d" is replaced by the value of the corresponding parameter. The sequence of "%d" corresponds to the sequence of the integer parameters, "%f" stands for parameters of the type real.
The value of a numerical expression between the "%" and the "d" determines the width of the integer field that is printed. In the case of a floating variable two expressions separated by a decimal point specify the width and the number of decimal digits that are printed.
The capital forms "%D" and "%F" will fill leading spaces with zeros.
A character format descriptor "%c" or "%Nc", with N an integer number, describes a string of characters.

```
Examples
echo ">%3d<",44            produces :  > 44<
echo ">%1+2d<",44          produces :  > 44<
echo ">%3D<",44            produces :  >044<
echo ">%5.1f<",44.1        produces :  > 44.1<
echo ">%2**2+1.1f<",44.1   produces :  > 44.1<
echo ">%c<",'bla'          produces :  >bla<
echo ">%5c<",'bla'         produces :  >  bla<
```

3.16 eval

eval <expr> [, <expr> ...]

Evaluates the expression(s) and displays the result(s). The result is not stored, this command is for interactive display only.

3.17 exit

exit

Terminates the program and gets you back to your shell.

3.18 expressions

Arithmetic expressions can be evaluated in a FORTRAN style. Character expressions are used to assign a string of characters to a variable or filename.

Arithmetic expressions:

Five basic operators are defined:

```
"+"  Addition
"-"  Subtraction
"*"  Multiplication
"/"  Division
"**" Exponentiation
```

The usual hierarchy of operators holds. The parts of the expression can be grouped with parentheses "(",")" in order to circumvent the standard hierarchy Several intrinsic functions have been defined, see "functions" for a full listing.
Examples of valid expressions are:

```
1
1+3*(sin(3.14*r[1]))
x[1]*0.155
asind(0.5)
```

Character expressions

A character expression is signaled by a pair of " ". The content may be a just a simple string of characters or additional format specifiers that are replaced by the value of a variable.

```
variable character,string
variable character,line
string = "abcdefgh"
line   = "%4c",string(2:5)
line   = "%c %c",string(1:2),string(7:8)
line   = "Number: %3d",4
line   = "Number: %3.1f",4.1
line   = string          ! Both commands work,
line   = "%c",string     ! this is the prefered style
line   = "%c",fdate(0)   ! See ==> functions for a list of
```

! character functions Within an ==> 'if' construction you may also specify a character expression in the form:

```
if( '"%2c",string(3:4)' .eq. 'cd' ) then
```

An expression (M:N) refers to the substring from the M's to the N's character.

Format specifiers

In filenames ==>"filename" or character expressions format specifiers are used to write the value of numerical or character variables into the corresponding string. These format specifiers may be:

```
%d      writes a decimal/integer number, the number of digits
        depends on the numerical value of the number
%D      writes a decimal/integer number, the number of digits
        depends on the numerical value of the number
%3d     writes a decimal/integer number that fills 3 digits
%3D     writes a decimal/integer number that fills 3 digits,
        leading blanks are filled with zeros
        Any width larger than the number of digits required is allowed
%Md     writes a string M digits wide. M may be omitted.
        M may be an integer expression.
        d or D are allowed, D give leading zeros
%f      A floating/real number is written flushed left into a
        character string of 8 digits
%F      A floating/real number is written flushed left into a
        character string of 8 digits
%12.3f  A floating/real number is written flushed right into a
        character string of 12 digits. 3 digits are used for the
        fractional part.
%12.3F  A floating/real number is written flushed right into a
        character string of 12 digits. 3 digits are used for the
        fractional part. Leading blanks are filled by zeros.
%M.Nf   writes a string M digits wide. N may be omitted.
        M and N may be integer expressions.
        f or F are allowed, F give leading zeros
%c      A character string is written, the width depends on the input
        variable
%5c     A character string of 5 characters is written.
```

Examples

```
variable character,string
variable character,line
string = "abcdefgh"
line   = "%c",string     ==> "abcdefgh"
line   = "%4c",string(1:4)     ==> "abcd"
line   = "Hallo %c",string(2:4)    ==> "Hallo bcd"
line   = "Number %5d",1234         ==> "Number  1234"
line   = "Number %5D",1234         ==> "Number 01234"
line   = "Float  %8.3f",3.1415     ==> "Float     3.141"
line   = "Float  %8.3F",3.1415     ==> "Float  0003.141"
```

3.19 filenames

Usually, file names are understood as typed, including capital letters. Unix operating systems distinguish between upper and lower case typing !
Additionally (integer) numerical input can be written into the filename. The syntax for this is:

```
"string%dstring",<integer expression>
"string%fstring",<real expression>
"string%cstring",<character expression>
```

The file format MUST be enclosed in quotation marks. The position of each integer must be characterized by a "%d". The sequence of strings and "%d"'s can be mixed at will. The corresponding integer expressions must follow after the closing quotation mark. If the command line requires further parameters (like "addfile" for example) they must be given after the format-parameters. The interpretation of the "%d"'s follows the C syntax. Up to 10 numbers can be written into a filename.
Refer to the help entry "expressions" for further help.
Examples:

```
1)
i[5]=1
outfile a1.1
outfile "a%d.%d",1,1
outfile "a%d.%d",4-3,i[5]
outfile "a%1.1f",1.1
```

All the above examples will result in the file name "a1.1".

```
2)
do i[1]=1,11
...
outfile "data%d.calc",i[1]
...
enddo
```

The output is written to the files "data1.calc" through "data11.calc"

3.20 fclose

fclose {<**number**>|**"all"**}
This command closes a file that was opened with 'fopen <number>' or closes all open files. If this command is not used before exiting the program, data might be lost !

3.21 fend

fend <**number**>,{**'continue'**|**'error'**}
This command determines the reaction to an unexpected end of file while reading data from input file <number>. If the parameter is set to "continue", the program will set the variable res[0] to -1 and continue the macro. If you repeat the ==> 'fget' command, the program will again set res[0] to -1 and will not result in an error. In order to catch and EOF, you have to evaluate the value of res[0] each time the 'fget' command is executed.
If the parameter is set to "error", the program will stop reading data from the input file and terminate the macro with an error message. The value of res[0] remains undefined.
The default condition at program start is "error"

3.22 fexist

fexist <**file**>

This command checks the existence of the specified file <file>. The result is written on the screen and returned via the res[] variables. If the file exists, res[1] is 1, otherwise it is 0. The variable res[0] returns the number of parameters, here 1.

3.23 fformat

fformat <**nc**>,<**format**>

This command allows one to specify a FORTRAN style format string <format> to be used for column <nc>. The default is free format, which can be selected using the character * as format string. If the command is called with no parameters, the current settings are displayed on the screen. Note that an unsuitable format might result in a conversion error and *** being written to the file !

```
Example: fform 1,F7.3
```

3.24 fget

fget <**number**>,<**p1**>,<**p2**>,..

This command allows the user to read data from a file that had been opened with 'fopen <number>. If no parameters are given, a line is read, yet its content is ignored and the line gets skipped. Otherwise the read numbers will overwrite the contents of the specified variables. The values in the input file must be separated by a blank or a comma. This means that to read a set of words in "This is a sentence", you will have to read this into 4 character variables. Note that a 'fget' command that does not run into an unexpected end of file sets the value of res[0] to zero!
Example: variavle character, str_a fget 1, r[2],i[2] fget 1, r[2],str_a,i[2]

3.25 fopen

fopen <**number**>,<**file**> **[,{"append"** | **"overwrite"}]**

This command allows the user to open a file for reading and writing using the commands 'fget' and 'fput'. The first argument is the number of the io_stream. You can open several files at once, the exact value depends on the value of the variable MAC_MAX_IO in file "macro.inc". The second argument is the file name. The default is that existing files will be overwritten if 'fput' is used. Alternatively one can append data to a file by specifying the optional parameter "append".

3.26 fput

fput <**number**>,<**p1**>,<**p2**>, ..

This command allows one to write data to the file that had been opened by 'fopen <number>'. The parameters <pi> can either be variables and expressions or simple text enclosed in single

quotes. If no parameters are given, an empty line is written. In order to mix character variables or character functions and numbers, the first parameter must be a format descriptor in double "".

```
Examples:  fput 1, i[1],sqrt(1.0+i[1]*0.01)
           fput 1, 'Current value of i[1] : ',i[1]
           fput 1, "%c %d",'Current value of i[1] : ',i[1]
           fput 1, "%c",fdate(0)
```

3.27 fsub

fsub <number>,[<left>,<right>]

The command allows you to limit the string from which 'fget' reads the data from file <number>. Data will only be read columns <left> to <right>. If both parameters are missing, the full input string is read. If the parameter <right> is set to "-1", the string is read from <left> all the way to the end of the input string, independent of its length.
The default values at program start are 1,-1 for all input channels.

```
Examples:
  Input line: "A text string 20.0  30.0"
  fsub 14,24
  fget r[1],r[2]
```

3.28 functions

The following intrinsic numerical functions exist:

```
asin(<arg>)                  !
acos(<arg>)                  !
atan(<arg>)                  !
atan(<arg1>,<arg2>)          ! Arguments are sine and cosine of angle
asind(<arg>)                 ! Result in degrees
acosd(<arg>)                 ! Result in degrees
atand(<arg>)                 ! Result in degrees
atand(<arg1>,<arg2>)         ! Arguments are sine and cosine of angle
sin(<arg>)                   !
cos(<arg>)                   !
tan(<arg>)                   !
sind(<arg>)                  ! Argument in degrees
cosd(<arg>)                  ! Argument in degrees
tand(<arg>)                  ! Argument in degrees
sinh(<arg>)                  ! Hyperbolic functions
cosh(<arg>)                  !
tanh(<arg>)                  !
sqrt(<arg>)                  ! Square root of <arg>
exp(<arg>)                   ! exponential (base e)
ln(<arg>)                    ! natural logarithm of <arg>
abs(<arg>)                   ! Absolute value of <arg>
mod(<arg1>,<arg2>)           ! Modulo <arg1> of <arg2>, real arguments
max(<arg1>,<arg2>)           ! Maximum of <arg1> and <arg2>
min(<arg1>,<arg2>)           ! Minimum of <arg1> and <arg2>
int(<arg>)                   ! Convert argument to integer
nint(<arg>)                  ! Convert argument to nearest integer
```

```
frac(<arg>)                     ! Returns fractional part of <arg>
ran(<arg>)                      ! Returns uniformly distributed pseudo
                                  random value 0<= r < 1.
gran(<arg>{,"s"|"f"})           ! Returns gaussian distributed pseudo
                                  random value with mean 0.0 and
                                  sigma <arg> or FWHM <arg> if the
                                  second argument is equal to "f".
gskew(<arg>,<skew>{,"s"|"f"})   ! Returns gaussian distributed pseudo
                                  random value with mean 0.0 and
                                  sigma <arg> or FWHM <arg> if the
                                  second argument is equal to "f".
                                  If skew is 0 the distribution is symmetric
                                  For skew <=  1.0 it is right skewed
                                  For skew >= -1.0 it is left  skewed
logn(<arg1>,<arg2>{,"s"|"f"})   ! Returns lognormal distributed pseudo
                                  random value. <arg1> is the location
                                  of the most likely value.
                                  <arg2> is the width of the distribution.
                                  More accurately, <arg2> is the width of
                                  the underlying distribution ln(logn).
                                  It is either sigma <arg2> or FWHM <arg2>
                                  if the third argument is equal to "f".
pois(<arg>)                     ! Returns Poisson  distributed pseudo
                                  random value with mean <arg>.
```

The arguments to any of these functions are any arithmetic expression.
System functions:

```
date(0)                         ! Returns the current date as character
                                  string in the format:
                                  CCYYMMDDhhmmss.sss
                                  CCYY  : year    (century, year)
                                  MM    : month   (1,2,... 12)
                                  DD    : day     (1,2,... 31)
                                  hh    : hour    (1,2,... 24)
                                  mm    : minute  (1,2,... 60)
                                  ss.sss: second.milliseconds
                                  (g77: milliseconds are 000)
fdate(0)                        ! Returns the current date as character
                                  string in the format:
                                  Day Mon DD hh:mm:ss CCYY
                                  Day   : weekday (Mon, Tue,... Sun)
                                  Mon   : month   (Jan, Mar,... Dec)
                                  DD    : day     (1,2,... 31)
                                  hh    : hour    (1,2,... 24)
                                  mm    : minute  (1,2,... 60)
                                  ss    : second  (1,2,... 60)
                                  CCYY  : year    (century, year)
fmodt(0)                        ! Returns modification data and time
                                ! of the file opened last. The format
                                ! is the same as for fdate(0)
getcwd(0)                       ! Returns the current directory as
                                  character string
getenv('name')                  ! Returns the value of the environment
                                  variable <name>
index(<line>, <substring> [,"BACK"]) ! Returns the location of the
                                  substring within the line.
                                  If the optional keyword "BACK" is present,
                                  index returns the last occurence.
                                  Both, the line and the substrings may be
                                  given as simple strings in single quotations
                                  or as a variable with with a preceeding
                                  format specifier, Examples:
```

```
                                line ='Discus'
                                index('Discus','s')
                                index('Discus','s', BACK)
                                index("%",'Discus',"%",'s')
                                index("%",line    ,"%",'s')
                                index("%c",getcwd(0), '/')
length(<line>)                  !Returns the length of string <line>
                                The line may be
                                given as simple strings in single quotations
                                or as a variable with with a preceeding
                                format specifier, Examples:
                                length('Discus')
                                length("%c",line)
                                index("%c",getcwd(0), '/')
```

3.29 help

help [<**command**> [, <**subcommand**>]]

The 'help' command is used to display on-line help messages. They are short notes on the command <command>. The command may be abbreviated. If the abbreviation is not unique, only the first help topic that matches the command is listed.
The first line of the help text gives the syntax of the command that is explained in the following lines. For a few commands the syntax line is repeated for different set of possible parameters. After the text is displayed, you are in the HELP sublevel of PROG and there are the following commands possible:

```
<command> : Display help for <command> of current help level.
".."      : Go up one help level.
"?"       : Prints list of help entries of the current level.
<RETURN>  : Exit help sublevel.
```

3.30 if

The if-control structure takes the following form:

```
if ( <logical expression> ) then
  <conditional commands>
[elseif ( <logical expression>) then
  <conditional commands>]
[else
  <conditional commands>]
endif
```

The logical expressions may contain numerical comparisons with syntax:

```
<arithmetic expression> <operator> <arithmetic comparison>
```

The following operators are allowed:

```
.lt.    ! less than
.le.    ! less or equal
.gt.    ! greater than
.ge.    ! greater or equal
.eq.    ! equal
.ne.    ! not equal
```

The logical expressions may also contain string comparisons with syntax:

```
'<string1>' <operator> '<string2>'
```

Both strings MUST be enclosed by single apostrophes '. The operators are the same as those for the numeric expressions, lexical comparisons are used to evaluate the comparisons "less" and "greater". Within the single apostrophes you can place a character replacement operation. Thus a valid example would be: variable character, line line='text' if('"%c",line' .eq. 'text') then Logical expressions can be combined by logical operators:

```
.not.   ! negation of the following expression
.and.   ! logical and
.eqv.   ! logical equivalent
.xor.   ! logical exclusive or
.or.    ! logical or
```

Logical operations may be nested and grouped by brackets "(" and ")".

3.31 learn

learn [<name>]

Starts a learn sequence. All following commands are saved as typed in file <name>. defaults to "<prog>.mac". ==> lend finishes the learn sequence.

3.32 lend

lend

Finishes the learn sequence started by ==> learn.

3.33 seed

seed [<value>]

Reinitializes the pseudo random generator. The seed passed to the random generator is -abs(nint(value)). If the <value> is omitted, the random generator will be passed the number of hundredth of seconds passed since midnight, essentially initializing the sequence at a unknown fairly random point.

3.34 set

set <command>,..

This command allows to alter various program independent setting. Allowed values for <command> are:

prompt

set prompt, {**"on"**|**"off"**|**"redirect"**}**,[**{**"on"**|**"off"**|**"file"**}**, ["save"]]**
set prompt, "old"

First parameter sets the status of the PROG prompt. The default is "on", i.e. PROG prompts for the next command by writing "discus > " (in case you run DISCUS ..). You can turn this prompt off. This is useful, if you are running a long macro and do not want to get all the prompts written into the output. By using this option you can considerably shorten the output written by PROG into a redirected log file. If you are using PROG on a UNIX platform, you can start the program with redirected input by the command:

```
"prog < inputfile"
```

By default, PROG will write the prompt "discus >" into the output file, expecting a RETURN from the keyboard. Very long lines in the output file will result. To avoid this situation insert the line "set output,redirect" as first line in the inputfile to force discus to echo the lines from file inputfile.
The prompt setting "redirect" has an important side effect on macro treatment. With the "redirect" setting, macros are stored internally, once they have been read from disk, and will be reused from memory. This helps to reduce unnecessary I/O, especially when you have nested macros inside loops. For all other settings, the internal macro storage is cleared when you get back to the normal interactive mode. This allows you to run a macro, then modify the version stored on the disk and execute the modified/corrected version.
This second parameter allows the user to assign where the text output of the program should go: "on" prints on the screen, "off" will result in no output and "file" will save the output to a file progname.log (e.g. discus.log in DISCUS). Note that the output of the commands 'echo' and 'eval' will always appear on the screen. The last parameter allows on to save the current prompt and output settings.
The parameter "old" allow the user to and restore the setting of the prompt and output. This can e.g. be used to turn the prompt off in a macro and then restore the original setting after the macro is executed.

error

set error , {**"cont"** | **"exit"** | **"live"** }

Sets the error status.

```
"cont"  PROG returns the normal prompt after the display of the error
        message. You can continue the input of commands.
        The execution of a macro file is stopped, the program continues
        with the regular prompt of the menu/submenu where the error occurred.
```

```
"exit"  PROG terminates after the display of the error message.
        This option is useful if you run PROG in the batch mode of your
        operating system. Instead of continuing with a faulty calculation
        PROG stops and you can immediately check the error.
"live"  PROG remains alive after an error is encountered. The variable
        "res[0]" is set to -1. The error number is written into "res[1]",
        the error type to "res[2]". Further error codes are written into
        "res[3]".
        With this error status, the program remains alive within a loop
        as well, which it does not do with the error status "cont".
        The program also continue to execute a macro!
        It is most helpful to catch errors from the 'system' command
        and to allow a flexible response.
```

debug

set debug, {**"on"** | **"off"**}

This command allows the user to enable various DEBUG outputs ...

3.35 show

show {**"res"** | **"variables"**}

The show command displays settings onto your screen. The individual programs discus, kuplot, diffev, mixscat all have specific parameters to the show command as well, see the help at the main program level for details.

```
"res"
```

Many commands produce results that are stored in the result variable res[<i>]. Thise are displayed via "show res". The entry res[0] gives the number of entries in the result variable.

```
"variables"
```

Lists the variables that have been defined, their type and their current values. The command is identical to the "variable show" command.

3.36 sleep

sleep <**seconds**>

This command causes the program to sleep for <seconds> seconds

3.37 socket

socket "open",<**host**>**,**<**port**>
socket "close"
socket "exit"
socket "send",<**string**>

socket <**string**>
socket "transfer",<**slave_var**>,<**expression**>
socket "transfer",<**slave_var**>,<**character expression**>,<**local_var**>

These commands connect to a program that has been started with the "-remote" option, sees ==> "Options" for further help. The commands will be send to the slave program where they will be executed as typed.

open

socket "open",<**host**>,<**port**>

Opens the connection to the program running on host <host> and listening on port <port>. The server program must have been started beforehand with the "-remote" option

close

socket close

Temporarily closes the connection to the slave program, which keeps running. You can open to the same program with a new socket open,<pname>.

exit

socket exit

Terminates the slave program and closes the connection. To communicate with the slave program again, you must first start this with the "-remote" option and then open the connection with socket "open",<pname>.

send

socket send,<**string**>
socket <**string**>

The string is send to the slave program and executed as typed. Both forms are equivalent.

transfer

socket transfer,<**slave_var**>,<**expression**>
socket transfer,<**slave_var**>,<**character expression**>,<**local_var**>

This command transverse the value of <expression> to the slave variable <slave_var>. You must define this slave variable prior to a transfer.

Example

```
system /usr/local/bin/kuplot -remote   ! starts kuplot slave
i[0] = 4.0                             ! set local variable
socket open,localhost,3331             ! open connection
socket func r[0]**2,-2,2,0.1           ! define a function to plot
socket transfer,i[1],i[0]              ! transfer local variable
```

```
socket tit1 "%d",i[1]                ! set plot title
socket plot                          ! display plot
socket mark 1,1                      ! change marker interval
socket plot                          ! plot again
socket send, mark 0.5,0.5            ! change marker interval
socket send, plot                    ! plot again
socket exit                          ! terminate kuplot, exit
```

3.38 stop

stop

This command is active only while reading from a macro file or in interactive mode inside a block structure (do-loops and/or if's).
The current macro file is interrupted and you can type commands as in the normal input mode. You can use the whole range of PROG commands, including the '@' macro command. The 'stop' commands provides a convenient mode to debug a macro by setting a break point at which you can check the value of variables or set new values, run an additional macro etc.
To continue execution of the macro or to continue with the normal PROG mode, use the ==> 'continue' command.
If included in a block structure statement (do-loops and/or if's) in both, macro and interactive mode, the program continues reading all statements that belong to the block structure. During execution of the structure, PROG interrupts this execution if it encounters a 'stop' command. You can issue any PROG command except further do or if commands.
To continue execution of the structure or to continue with the normal PROG mode, use the ==> 'continue' command.

3.39 system

system <**com**>
system "string%dstring",<**integer expression**>
system "string%fstring",<**float expression**>

Executes the single shell command <com>. If the command string is enclosed in "", you can place integer and real format specifiers "%d" "%f" which are then substituted by the corresponding values.
Example i[0]=10 system "ls %d.*"
This would list all files called 10.*

3.40 wait

wait [{**"return"** | **"input" [,**<**prompt**>**]** }**]**

This command waits for user input. Without a parameter or with "return", the program waits for a simple <RETURN>. If the first parameter is "input", the program expects the user to enter one or more real numbers or expressions. The optional <prompt> can be used to ask the user to input the expressions. This is especially helpful if the prompt has been turned of by ==> set

prompt,off. The number of expressions entered by the user is stored in the variable res[0] and the results of the expressions in res[i].
This command allows to write interactive macros, demo macros and tutorials.

3.41 variable

```
variable {"integer"|"real"},<name> [,<initial_value>]
variable {"character"},<name>
variable show
```

The programs that are part of the DIFFUSE suite offer predefined variables i[*] and r[*]. These are an integer and a real array, respectively, into whose element you may store appropriate values. In order to enhance readability of a macro, you can define your own variable names by the use of this command. The variable may either be an integer or a real variable. There is no predefined syntax for the variable names. Optionally you can initialize the variable to <initial_value>, default is zero. These user defined variables may be used just as the system integer and real variables i[*] and r[*]. Character variables may be used to hold a string of text. The variable names may only consist of alphanumerical characters including the underscore "_". Presently they are single value variables.
If the first command parameter is "show" the program displays a list of user defined variables and their current values. Refer to the help entry "expressions" for further help.
Examples:

```
variable  int,alpha,90
variable  int,beta
variable  real,diff
beta = 94
diff = alpha - beta
eval diff
variable  character,string
variable  character,line
string = "abcdefg"
line   = "%3c",string(2:4)
var show
```

Variable names that are part of intrinsic functions, keywords like "do", "elseif", "eq", and program specific variables like "r" and "i" are not allowed. Thus a variable called "a" is illegal, since it is part of the intrinsic function "asin".
Internally the program sorts the variable names by length and in inverse alphabetical order. This sorting has no serious consequence for the user other than finding the variable in the printed list when using the 'variable show' command.
A few predefined variables exist, their names are capitalized: PI 3.1415... REF_GENERATION Current generation number as set by DIFFEV REF_MEMBER Current population size as set by DIFFEV REF_CHILDREN Current child population size as set by DIFFEV REF_DIMENSION Current Number of parameters a set by DIFFEV If the value is changed, you can store more values in ref_para[], but this does NOT change the actual dimension pop_dimx that DIFFEV uses.

3.42 errors

The program has been written such that it should handle almost any typing error when giving commands and hopefully all errors that result from calculation with erroneous data. When an error is found an error message is displayed that should get you back on track. See the manual for a complete list of error messages. In this part we refer to the program you are using as DISCUS for convenience.
The error messages concerning the use of the command language are grouped in the following categories:

```
COMM   Command language errors
FORT   Fortran interpreter errors
I/O    Errors regarding input/output
MACR   Errors related to macros
MATH   General mathematical errors
```

Each error message is displayed together with the corresponding category <cccc> and the error number <numb> in the form:

```
****CCCC****message                    **** numb ****
```

In the default mode DISCUS returns the standard prompt and you can continue the execution from this point. You can set the error status to "exit" by the ==>'set' command. In this case DISCUS terminates if an error is detected. This option is useful to terminate a faulty sequence of commands when running DISCUS in the batch mode of your operating system.

comm

Command language errors These messages describe illegal usage of the command language, such as unknown commands, improper numbers of parameters.

Error -1: DISCUS directory not defined

The environment variable DISCUS_DIR was not defined. Check the chapter on installation for your platform for the appropriate definition.

Error -2: Command parameter has zero length

On the command line you probably have a typing error like two comma following each other without significant values in between, or the first non blank character after the command is a comma.

Error -3: Could not allocate arrays

The program has to allocate arrays, but received a error message. Does your computer have enough available memory space?

Error -5: Error in operating system command

The operating system/shell returned an error message. Check the appropriate system manuals for details.

Error -6: Missing or wrong parameters for command

Either the command needs more parameters than were provided, or the parameters are incorrect. Check the number and type of parameters. Is the sequence of numerical and character parameters correct?

Error -8: Unknown command

The command interpreter read an unknown command. Check the spelling of the command or check, whether this command is allowed at the current sublevel.

Error -11: Error in subroutine

More or less a system error message, ignore this message.

Error -17: Too many parameters

More parameters have been provided than are required by the command. Check the number, and type of parameters supplied, or the occurrence of additional ',’.

fort

Fortran interpreter errors These messages describe erroneous mathematical calculations and improper usage of control structures (do,if, ...).

Error -1: Nonnumerical Parameters in expression

The interpreter found a nonnumerical string where a number is expected. If an intrinsic function or a variable was intended, check for spelling or missing parentheses.

Error -2: Unknown Variable

The expression contains a reference to an unknown variable. Check the spelling of the variable. Chapter 3.7.1 of the manual and the help entry "variables" contains a list of allowed variables. Check whether the variable is a read-only variable and was used on the left side of an expression. Some of the variables associated with microdomains are read-only depending of the circumstances!

Error -3: Unknown intrinsic function

The expression contains a reference to an unknown intrinsic function. Check the spelling of the function. Chapter 3.7.4 of the general part in the manual and the help entry "functions’ contain a complete list of the allowed intrinsic functions.

Error -4: Division by zero'

An attempt was made to divide by zero. Check the value of the argument and correct the algorithm that calculates the argument.

Error -5: Square root of negative number

An attempt was made to calculate the square root of a negative argument. Check the value of the argument and correct the algorithm that calculates the argument.

Error -6: Missing or wrong Parameters for command

Either the function or variable referenced needs more parameters than were provided, or the parameters are incorrect. Check the number and type of parameters. Is the sequence of numerical and character parameters correct?

Error -7: Argument for asin,acos greater 1

An attempt was made to calculate asin or acos with an argument greater than 1. Check the value of the argument and correct the algorithm that calculates the argument.

Error -8: Index outside array limits

The index supplied for the variable is outside the limits of this variable. Check the general part for the dimensions of the variables.

Error -9: Number of brackets is not matching

The number of opening and closing brackets "[" and "]" does not match or is illegally nested with parentheses "(", ")" or other operators. Check the string used in the expression and correct it following the FORTRAN rules.

Error -10: Index for array element is missing

You have used a string like "i[]", where the opening and closing brackets do not contain any expression. Check the string used in the expression and correct it following the FORTRAN rules.

Error -11: Number of parentheses is not matching

The number of opening and closing parentheses "(" and ")" does not match or is illegally nested with brackets "[", "]" or other operators. Check the string used in the expression and correct it following the FORTRAN rules.

Error -12: Expression between () is missing

You have used a string like "()", where the opening and closing parentheses "(" and ")" do not contain any expression. Check the string used in the expression and correct it following the FORTRAN rules.

Error -13: Wrong number of indices for array

The number of indices given for the entered parameter is wrong. Check the help entry 'variables' for the proper number of indices.

Error -14: Index of DO-loop counter is missing

Here the index for the loop counter of a do-loop is missing. Check the online help for the correct syntax of such loops.

Error -15: Too many commands

The program stores all commands within a control block in an array. The maximum number of commands that can be stored in this array is given by the parameter MAXCOM in file "doloop.inc". The macro or run used more commands than currently allowed by this parameter. Rewrite the macro or list of commands such that less commands are sufficient, or change the value of the parameter and recompile the program.

Error -16: Too deeply leveled (do,if) construction

The program stores all commands within a control block in an array. The maximum number of levels for this array is given by the parameter MAXLEV in file "doloop.inc". The macro or run used more levels than currently allowed by this parameter. Rewrite the macro or list of commands such that less levels are sufficient, or change the value of the parameter and recompile the program.

Error -18: Unresolvable condition

An error occurred while trying to calculate the value of an arithmetic or logical expression. Check that there is no illegal operation /(division by zero .../ no typing errors, all parentheses are properly matched.

Error -19: Illegal nesting of control commands

Do loops and/or if constructions have been nested with overlapping segments, missing enddo or endif statements or similar causes. Check for spelling errors on the control statements, and that each control statement is properly terminated by a corresponding enddo or endif statement that is not enclosed within another control block.

Error -20: Illegal argument for ln(x) function

The argument for the ln must be positive, larger than zero. Check the value of the argument or the value of the expression that serves as argument

Error -21: Missing ' while comparing stings

An expression of the form ('string' .eq. 'line') was used, where one of the quotation marks has been omitted. Check the respective line.

Error -22: Maximum number of real variables defined

DISCUS can define a fixed number of user variable names. The maximum number allowed for your installation id displayed by the command variable show If you would like more user definable variable names, change the value of the parameter VAR_REAL_MAX in "config.inc"

Error -23: Maximum number of int. variables defined

DISCUS can define a fixed number of user variable names. The maximum number allowed for your installation id displayed by the command variable show If you would like more user definable variable names, change the value of the parameter VAR_INTE_MAX in "config.inc"

Error -24: Variable is not defined

You tried to use a name within an expression that was not recognized as a user defined variable name. Check the spelling of the line. Was an intrinsic function to be used, or was the variable not defined? See the ==> 'variable' entry in the help menu regarding the definition of variables. You will also get this error message if you tried to define a variable using the command: variable real,dummy=3 The equal sign "=" may not be used as part of a variable name. If you intend to provide an initializing value, use the command as: variable real,dummy,3

Error -25: Variable name contains illegal characters

You tried to define a variable name that contains characters other than letters, numbers or the underscore "_". The variable names are restricted to alphanumerical characters and the underscore "_".

Error -26: Variable name contains illegal characters

Variable names may consist only of letters (lower and upper case), numbers and the underscore "_". Check the spelling of the variable you tried to define with respect to these rules.

Error -27: Function with wrong number of arguments

You called an intrinsic function with the wrong number of arguments. Check the listing of intrinsic functions for the valid number of arguments and the compare to the input line you had typed.

Error -28: Too deeply leveled break command

Illegal use of the break command. The parameter on the break command signals how many block structure levels are to be exited. Check the value of this parameter with regard to the nesting of do-loops and if-blocks.

Error -29: Character substring out of bounds', & !-29 ! fortran

In a statement like variable character, line line = 'abcde' echo "%c",line(1,5) The first index is less than one, or the second index is larger than the number of characters in the strin, or the second index is less than the first.

Error -30: Right quotation mark missing in format'

A statement like echo " text " is missing the right quotation mark.

Error -31: Incomplete (do,if) statement

Some part of a ==> 'do' or ==> 'if' statement is missing. Check the line for missing part or typing errors.

Error -32: Variable name is already defined

The variable that you want to define is already in use as another data type.

Error -33: Variable in use; cannot initialize value

A variable name can be redifined as identical data type, in order to be able to use a macro with a variable definition inside a loop. You may, however, not provide an initialisation value, as this would override the current value.

Error -34: String has length zero', & !-34 ! fortran

A statement like echo "" or line = '' occured in which the single or doluble quotation marks enclose a zero length string.

i/o

Errors related to input / output An error occurred while attempting to read/write from a file

Error -1: File does not exist

DISCUS could not find the file. Check the spelling and the path.

Error -2: Error opening file

DISCUS could not open a file. The file might be in use by another process.

Error -3: Error reading file

An error occurred while DISCUS was reading a file. Check whether the contents of the file is correct.

Error -4: File already exists

An attempt was made to overwrite an existing file. Rename or delete the file in question.

Error -5: No such entry in online help

You have tried to obtain help for a string that does not have a matching entry in the help file. Check the spelling of the string. Are you at the right sublevel? Use the '?' command to get a listing of available help entries.

Error -6: Unexpected end of file

DISCUS has encountered the end of a file, but is still expecting data. Check the file(s) involved, to see whether the data are complete or whether erroneous data are present.

Error -7: Learning sequence already in progress

You have tried to start a learning sequence by ==>'learn' without closing the active learning sequence. Close the current learning sequence by ==> 'lend' before starting to record a new macro.

Error -8: Nothing learned - no macro written

You did not type any commands since the ==>'learn' command. No commands are written to the macro file. You need to give at least one command before closing a learn sequence.

Error -9: Error reading user input

An error occurred while reading the last input. Does the string contain any characters where a number is expected, or any control or escape sequences.

Error -10: IO stream already open

The command 'fopen' was issued while there was already a file open. Close the currently open file with 'fclose'.

Error -11: No IO stream open to close

The command 'fclose' was issued, but there is no open file.

Error -12: Error writing to file

An error occurred when reading a file with 'fget'. Check the file for nonnumerical values and check that the number of columns is equal or larger than the number of arguments of 'fget'.

Error -13: I/O stream number outside valid range

The I/O stream number must be larger than 0 and less than the value defined in macro.inc, which usually is 10.

Error -14: Filename has zero length

You tried to open a file with ==> 'fopen', whose file name is of length zero. Check the statement for the missing filename, or an additional comma.

Error -15: No socket connected

Apparently you tried to use a ==> 'socket' command prior to opening a connection to the remote computer. Use a 'socket open' command first.

Error -16: Could not resolve hostname for socket

The hostname to which you want to connect could not be resolved into an IP address. Check the host name for typing errors and the Internet access of your computer.

Error -17: Could not grab socket

Internal socket debug message.

Error -18: Could not open socket connection

Internal socket debug message.

Error -19: Problem sending to socket

Internal socket debug message.

Error -20: Problem receiving from socket

Internal socket debug message.

Error -21: Received null string from socket receive

The remote computer send back an answer of zero length. Apparently the connection was terminated prematurely.
Try to connect again.

Error -22: Socket accept problem

Internal socket debug message.

Error -23: Scocket: Rejected connection

Internal socket debug message.

Error -24: Socket bind problem

Internal socket debug message.

Error -25: Socket listen problem

Internal socket debug message.

Error -26: Second parameter must be >= first Param.

You tried to read a substring with a line like echo "%c",line(1:5) but the second parameter , here a "5" was less than the first, here a "1". The second parameter must be equal to or larger than the first in order to specify a valid substring.

macro

Errors related to macro These messages describe situations that result from missing macrofiles, missing macro parameters ...

Error -1: Too many macro parameters given

The number of parameters given on the macro command line is higher than allowed in your installation. The maximum number of parameters allowed is defined by the parameter MAC_MAX_PARA in the file macro.inc. Check the macro command line for any additional "," or rewrite the macro to use less parameters. If necessary adjust the value of the parameter MAC_MAX_PARA and recompile the program.

Error -12: Macro not found

The file given on the @<name> command does not exist. Check the spelling of <name> and the path.

Error -13: Macro filename is missing on the command line

The command '@' to execute a macro was called without any macro file name. The file name must start immediately after the "@". Check the '@' command for completeness and blanks after the "@".

Error -35: Too deeply leveled macros

The maximum level at which macros may be nested is defined in the file macro.inc in the parameter MAC_MAX_LEVEL. Check the nesting of macro file for the level of nesting or possible recursive nesting without proper termination. Rewrite the macros to use less nesting, or change the value of the parameter and recompile the program.

Error -36: Unexpected EOF in macro file

When DISCUS finds a '@' command inside a macro, it stores the current macro name, the line number inside the current macro and closes the current macro file. After completion of the new macro, the previous macro is read again up to the position stored. The error message is displayed when an end of file is found before the position is reached. Check whether the macro file was damaged, or accidentally deleted during execution of the nested macro.

Error -41: Not enough macro parameters given

DISCUS read a parameter number inside a macro file that is higher than the number of parameters given on the command line of the macro. Check the parameters inside the macro for correct numbering and spelling. Check the number of parameters supplied on the command and check whether any "," is missing between parameters.