NEMO

Release 4.3

teuben

Oct 19, 2021

CONTENTS:

1 Introduction **3**

2 User Interface (*) **5**
- 2.1 Keywords . . . 5
- 2.2 Advanced User Interfaces . . . 8
- 2.3 Help . . . 9
- 2.4 Program keywords . . . 10
- 2.5 System keywords . . . 10
- 2.6 YAPP . . . 11

3 Filestructure (*) **13**
- 3.1 Binary Structured Files . . . 13
- 3.2 Pipes . . . 15
- 3.3 History of Data Reduction . . . 15
- 3.4 Table format . . . 16
- 3.5 Dynamically Loadable Functions . . . 16

4 Graphics and Image Display (*) **19**
- 4.1 The YAPP graphics interface . . . 19
- 4.2 pyplot=: python matplotlib . . . 20
- 4.3 General Graphics Display . . . 20
- 4.4 Image Display Interface . . . 20

5 Examples (*) **21**
- 5.1 N-body experiments . . . 21
- 5.2 Images . . . 27
- 5.3 Tables . . . 27
- 5.4 Potential . . . 27
- 5.5 Orbits . . . 27
- 5.6 Exchanging data . . . 30
- 5.7 Potential(tmp) . . . 30
- 5.8 Images . . . 31

6 Using NEMO **33**

7 In order to use NEMO, you will need to modify your **35**
- 7.1 Updating NEMO . . . 35

8 Installation **37**
- 8.1 Installation from github . . . 37
- 8.2 Rebuilding . . . 38

8.3 Advanced Installation 38
8.4 mknemo 38
8.5 python 38

9 Programmers Guide (*) **41**
9.1 The NEMO Programming Environment 41
9.2 The NEMO Macro Packages 42
9.3 Building NEMO programs 52
9.4 Extending NEMO environment 56
9.5 Programming in C++ 57
9.6 Programming in FORTRAN 57
9.7 Debugging 59

10 Potentials and Accellerations (*) **61**
10.1 Example Potentials 61
10.2 Accellerations 71

11 Units and Coordinate Systems **73**
11.1 Coordinate Systems 73
11.2 Units 74

12 Troubleshooting (*) **75**
12.1 List of Run Time Errors 75
12.2 Environment Variables used by NEMO 77

13 References **79**

14 Related Codes (*) **81**
14.1 AMUSE 81
14.2 Martini 83
14.3 ClusterTools (*) 83
14.4 ZENO 83
14.5 List of Related Codes 84
14.6 Categories 86

15 Glossary **87**

16 Research Software Engineering **89**

17 Todo List **91**

18 RST reminders **93**
18.1 Lists 93
18.2 Code blocks 93
18.3 Images 94
18.4 Tables 96

19 Indices and tables **97**

Index **99**

$$\ddot{\mathbf{r}}_i \;=\; -G \sum_{j=1;\, j\neq i}^{N} \frac{m_j \left(\mathbf{r}_i - \mathbf{r}_j\right)}{(r_{ij}^2 + \epsilon^2)^{3/2}}$$

Todo: This draft is a work in progress, and started on April 27. It is an updated version of the old (latex) NEMO Users and Programmers Guide. We hope to have this converted sometime in this Summer (2021) when version 4.3 is blessed. Not all sections from the old manual were taken, but we also added some new sections.

Other online entry points for NEMO are: github pages and github code . We also keep an index to (unix) man pages for all programs.

Although this manual should be on https://astronemo.readthedocs.io we also keep a local copy readthedocs on astroumd.

Note: **(*)** are sections that have not been fully cleaned up and can contain old latex markup

CHAPTER

ONE

INTRODUCTION

Note: NEMO is a toolbox with many programs to perform various stellar dynamics functions.

For the user NEMO is a collection of programs, running under a standard UNIX shell, capable of running various stellar dynamical codes and related utilities (initialization, analysis, gridding, orbits). It can be thought of as a collection of various *groups* (packages) of programs, a group being defined by their common file structure, described below.

A common command line *User Interface (*)* is defined with which the user communicates with a program. In order to run NEMO programs, your shell environment has to be modified. See *Using NEMO* on how to setup NEMO, and of course *Installation* if that had not been done yet. There is also a section with many *Examples (*)*. NEMO stores its information in binary files, obeying a *Filestructure (*)*.

Here are the main *groups* of programs, clearly showing the structure of NEMO:

- The *N-body group* is defined by a common file structure of *snapshots*. In this group we find various programs to create N-body systems (spherical, elliptical, disk), methods to compute the gravitational field (softened Newtonian, hierarchical, Fourier expansion), and time-integrators (leapfrog, Runge-Kutta). Many utilities exist to manipulate, analyze and display these data.
- The *Orbit group* is defined by a common file structure of *orbits*. It is mainly intended to calculate the path of an individual orbit in a static potential and then analyze it. This group is closely related to the before mentioned N-body group, and utilities in both groups can interact with each other. For example, it is possible to freeze the potential of an N-body snapshot, and calculate the path of a specific star in it, now conserving energy exactly. Or to extract the path of a selected star in a simulation, and extract an orbit from it.
- The *Image group* is defined by a common file structure of *images*, i.e. two dimensional rectangular pixel arrays with a 'value' defined for every pixel. Actually an image may also have a third axis, although this axis often has a slightly different meaning *e.g.* Doppler velocity. It is possible to generate arbitrary two-(and three-) dimensional images from snapshots, FITS files of such images can be created, which can then be exported to other familiar astronomical data reduction packages. There exists a variety of programs in the astronomical community to manipulate data through the FITS format.
- The *Table group* appears quite commonly among application programs in all of the above mentioned groups. Most of the time it is a simple ASCII file in the form of a matrix of numbers (like a spreadsheet). A few programs in NEMO can manipulate, display and plot such table files, although there are many superior programs and packages outside of NEMO available with similar functionality. It is mostly through these table files that we leave the NEMO environment, and persue analysis in a different environment. The obvious advantage of storing tables in binary form is the self-documenting nature of NEMO's binary files. For historical reasons, most tables are displayed and created in ASCII, though you will find a few binary tables as well.

More groups and file structures are readily defined, as NEMO is also an excellent development system. We encourage users to define their own (or extend existing) data structures as the need arises. In *Programmers Guide (*)* we will detail some 'rules' how to incorporate/add new software into the package, and extend your own NEMO environment.

The remaining chapters of this manual outline various concepts that you will find necessary to work with NEMO. The *User Interface (*)* outlines the user interface (commandline, shells etc.). NEMO stores most data in files, and *Filestructure (*)* explains how data is stored on disk and can be manipulated, including the concept of function descriptors in NEMO. In *Graphics and Image Display (*)* we details how data can be graphically displayed, either using NEMO itself or programs outside of NEMO.

CHAPTER

TWO

USER INTERFACE (*)

Note: The command line user interface is a series of *keyword=value* pairs, where we differentiate between *program keyword* and *system keyword*. The `--help` or `help=` options will describe the keywords.

A NEMO program is invoked just as any other application program under the operating system, entering its name from a Unix shell, for example:

```
mkplummer
```

We first explain the command line interface to NEMO programs. Subsequently, followed by some of the more advanced concepts of this user interface. We also discuss the overall documentation system in NEMO, and how to get different types of help.

2.1 Keywords

2.1.1 Program Keywords

Note: The **help=h** *system keyword* gives a nice overview of the *program keywords*. Since it is so common, using **–help** invokes the default **help=** showing the program keywords and their defaults.

The most basic user interface is formed by the command line interface. Every NEMO program accepts input through a list of **program keywords**, constructed as '*keyword=value*' string pairs on the commandline. We shall go through a few examples and point out a few noteworthy things as we go along. The first example runs an N-body simulation and writes the results in a file **r001.dat**:

```
1% hackcode1 out=r001.dat

Hack code: test data

   nbody        freq         eps         tol
     128       32.00      0.0500      1.0000

    options: mass,phase

    tnow        T+U         T/U       nttot       nbavg       ncavg     cputime
   0.000    -0.2943     -0.4940        4363          15          18        0.01
```

(continues on next page)

(continued from previous page)

```
            cm pos   -0.0000   -0.0000   -0.0000
            cm vel    0.0000    0.0000   -0.0000

    particle data written

    tnow       T+U       T/U     nttot     nbavg     ncavg   cputime
   0.031   -0.2940   -0.4938      4397        15        18      0.01

            cm pos    0.0000    0.0000   -0.0000
            cm vel    0.0001    0.0001   -0.0000

    tnow       T+U       T/U     nttot     nbavg     ncavg   cputime
   0.062   -0.2938   -0.4941      4523        16        18      0.02

            cm pos    0.0000    0.0000   -0.0000
            cm vel    0.0002    0.0002    0.0000

    ...
```

will integrate an (automatically generated) stellar system with 128 particles for 64 time steps. If your CPU is to slow, abort the program with `<control>-C` and re-run it with fewer particles:

```
....
<Control>-C
2% hackcode1 out=r001.dat nbody=32 > r001.log
### Fatal error [hackcode1] stropen in hackcode1: file "r001.dat" already exists
3% rm r001.dat
4% hackcode1 out=r001.dat nbody=32 > r001.log
```

This example already shows a few peculiarities of the NEMO user interface, as shown by the line starting with **###**. It is generated by the *error* routine, which immediately aborts the program with a message to the terminal, even if the normal output was diverted to a log-file, as in this example. The error shows that in general NEMO programs do not allow files to be overwritten, and hence the `r001.dat` file, which was already (partially) created in the previous run, must be deleted before **hackcode1** can be re-run with the same keywords. The datafile, **r001.dat**, is in a peculiar binary format, which we shall discuss in the next chapter.

Now, plotting the first snapshot of the run, i.e. the initial conditions, can be done as follows:

```
5% snapplot in=r001.dat times=0
```

It plots an X-Y projection of the initial conditions from the data file `r001.dat` at time 0.0. Your display will hopefully look something like the one displayed in Figure ???

There are many more keywords to this particular program, but they all have sensible default values and don't have to be supplied. However, an invocation like

```
6% snapplot
```

will generally result in an error message, and shows you the minimum list of keywords which need a value. `snapplot` will then output something like

```
Insufficient parameters, try keyword 'help=', otherwise:
Usage: snapplot in=??? ...
plot particle positions from a snapshot file
```

which already suggests that issuing the `help=` keyword will list all possible keywords and their associated defaults:

```
7% snapplot help=
```

results in something like:

```
snapplot in=??? times=all xvar=x xlabel= xrange=-2.0:2.0
    yvar=y ylabel= yrange=-2.0:2.0 visib=1 psize=0
    fill_circle=t frame= VERSION=1.3f
```

As you see, `snapplot` happens to be a program with quite an extensive parameter list. Also note that `help` itself is not listed in the above list of program keywords because it is a **system keyword** (more on these later).

There are a few *short-cut* in this user interface worth mentioning at this stage. First of all, keywords don't have to be specified by name, as long as you specify values in the correct order, they will be associated by the appropriate keyword. The order of program keywords can be seen with the keyword `help=`. The moment you deviate from this order, or leave gaps, all values must be accompanied by their keywords, *i.e.* in the example

```
8% snapplot r001.dat 0,2 xrange=-5:5 yrange=-5:5 "visib=i<10"
```

the second argument `0,2` binds to `times=0,2`; but if a value `"i<10"` for `visib` (the keyword immediately following `yrange=`) would be needed, the full `"visib=i<10"` would have to be supplied to the command line, anywhere after the first `0,2` where the keywords are explicitly named. Also note the use of quotes around the `visib=` keyword, to prevent the UNIX shell from interpreting the < sign for I/O redirection. In this particular case double as well as single quotes would have worked.

There are two other user interface short-cuts worth knowing about. The `macro-include` or `keyword include` allows you to prefix an existing filename with the `@`-symbol, which causes the contents of that file to become the keyword value. In UNIX the following two are nearly equivalent (treatment of multiple lines may cause differences in the subsequent parsing of the keyword value):

```
 9% program a=@keyfile
10% program a="`cat keyfile`"
```

Also useful is the `reference include`, which uses the `$`-symbol to prefix another program keyword, and causes the contents of that keyword to be included in-place. An obvious warning is in place: you cannot use recursion here. So, for example,

```
11% program a=$b b=$a           <---- illegal !!!
```

will probably cause the user interface to run out of memory or return something meaningless. Also, since the `$`-symbol has special meaning to the UNIX shell, it has to be passed in a special way, for example

```
12% program a=5 b=3+\$a
13% program a=5 'b=3+$a'
```

are both equivalent.

2.1.2 System Keywords

As just mentioned before, there are a fixed set of keywords to every NEMO program which are the *hidden* **system keywords** their values are defined automatically for the user by the user-interface routines from environment variables or, when absent, sensible preset defaults. They handle certain global (system) features and are not listed through the `help=` keyword. Of course their values can always be overridden by supplying it as a system parameter on the command line. To get an active list of the system keywords, try

```
tsf help=\?
```

In summary, the system keywords are:

- **help=** The help= keyword itself, gives you a list of all available keywords to this specific program but can also aid you in command completion and/or explanation of keywords.
- **debug=** The debug= keyword lets you upgrade the debug output level. This may be useful to check proper execution when a program seemingly takes too long to complete, or to trace weird errors. Output is to *stderr* though. Default level is 0. Some unix tools how to deal with pipes is useful (redir, ...)
- **error=** The error= keyword allows you to override a specified number of fatal error calls. Not adviced really, but it's there to use in case you really know what you're doing (bypassing existence of an output file is a very common use). Default is 0.
- **yapp=** The yapp= keyword lets you (re)define the graphics output device. Usually no default.
- **outkeys=** This is a new feature under development, effectively allows exporting information in text strings back to the shell.
- **review=** The review= keyword jumps the user into the REVIEW section before the actual execution of the NEMO program for a last review of the parameters before execution starts. (see also next section).
- **review=** Interrupt mode to review keyword before execution
- **tcl=** Deprecated
- **np=** Number of processors (for OpenMP) to maximally use. Default is max.

For a more detailed description of the system keywords and all their options see aiface. The actual degree of implementation of the system keywords can be site dependent. Use the `help=\?` argument to any NEMO program to glean into the options the user interface was compiled with. Recent updates can also be found in NEMO's online manual pages, *getparam(3NEMO)*.

2.2 Advanced User Interfaces

The command-line interface, as we described it above, makes it relatively straightforward to *plug in* any other front-end as a new user interface with possibly a very different look-and-feel. In fact, the command-line interface is the most primitive front-end that we can think of: most host shell interpreters can be used to perform various short-cuts in executing programs. Modern interactive UNIX shells like `tcsh` and `bash` can be used very efficiently in this mode. In batch mode shell scripts, if used properly, can provide a very powerful method of running complex simulations. Other plug-compatible interfaces that are available are `mirtool` and `miriad`, described in more detail in Appendix~ref{s:mirtool} and ref{s:miriad} There was also a Khoros (cantata, under khoros V1) interface (`http://www.khoral.com`) available, but this product is not open source anymore. Lastly, lets not forget scripting languages like python, perl and ruby. Although the class UNIX (c)sh shell is very WYSIWYG, with a modest amount of investment the programmability of higher level scripts can give you a very powerful programming environment.

2.2.1 tkrun

The `tkrun` program can take directives strategically placed in the comment fields of a shell script, and provide a dynamical GUI frontend to the command line parameters. Since the GUI is built up automatically, the number of keyword should be limited to a dozen or so, as vertical space is limited in most desktop managers.

2.2.2 Interrupt to the REVIEW section

Warning: Interrupting to the REVIEW section is not enabled by default, and is likely being deprecated in some future release.

NEMO programs are generally not interactive, they are of the so-called *load-and-go* type, i.e. at startup all necessary parameters are supplied either through the commandline, or, as will be described later, a keyword file or even a combination thereof. The actual program is then started until it's all done. There is no feedback possible to the user. This is particularly convenient when combining programs into a script or batch type environments.

There are of course a few exceptions. Certain graphics interfaces require the user to push a button on the keyboard or click the mouse to advance to a next frame or something like that; a few very old NEMO programs may still get their input through user defined routines (they will become obsolete).

2.3 Help

The HELP system in NEMO is manyfold, nice but with the obvious danger that things get updated in one place and outdated in another. With that caveat, here are various help options:

- **Inline help,** The `help=` system keyword is available for each NEMO program. Since this is compiled into the program, you can copy a program to another system, without all the NEMO system support, and still have a little bit of help. Use `help=h` to get the keyword descriptions and more vertical space.

  The special `--help` option is allowed for those with gnu fingers.

  The special `--man` option delivers the unix style man page (see next item).

- **Unix manual pages** for programs, functions, and file formats, all in good old UNIX tradition. All these files live in `$NEMO/man` and below. Several interfaces to the manual pages are now available:
  - **man** good old UNIX `man` (this relies on **$MANPATH** environment variable) The `manpdf` script can print out the manual pages in a pretty decent form.
  - **xman** The X-windows utility {it xman(1)} provides a point-and-click interface, and also has a decent {it whatis} interface.
  - **tkman** The Tcl/Tk X-windows utility `tkman` formats manual pages on-the-fly and allows hypertextual moving around. and has lots of good options, such as dynamic manipulation of the **$MANPATH** elements, a history and bookmark mechanism etc.
  - **gman** Under GNOME the `gman` formats tool has nice browsing capabilities.
  - **html** The html formatted manual pages. Has some limited form of hypertext, but contains the links to general UNIX manual pages, if properly addressed. Try the github link or local pages

- **The old manual, the *The NEMO User and Programmers Guide*,** contains information on a wide level, aimed at beginners as well as advanced users, and at is being coverted to this RST manual, outdated.

- This manual, in **reStructuredText** might be available in many different formats. html and pdf are the common ones.

Every NEMO program accepts input through a user supplied parameter list of *keyword=value* arguments. In addition to these program specific **program keywords**, there are a number of system wide defined **system keywords**, known to every NEMO program.

2.4 Program keywords

Program keywords are unique to a program, and need to be looked up in the online manual page or by using the `help=` system keyword (dubbed the **inline** help). Parsing of *values* is usually done, though sometimes primitive. Program keywords also have the ability to read the value(s) of a keyword from a file through the `keyword=@file` construct. This is called the **include keyword file**, and is very handy for long keyword values, not having to escape shell characters etc. Newlines are replaced by blanks.

2.5 System keywords

The 'hidden' system keywords, although overridden by any program defined counterpart, can also be set by an equivalent environment variable (in upper case).

- **help=** Sets the help level to a program. As with all system keywords, their value can be fixed for a session by setting the appropriate environment variable in upper case, *e.g.* `expor HELP=5`.

  By using the keyword form, the value of the environment variable will be ignored.

  The individual help levels are numeric and add up to combine functionality, and are hence powers of 2:

  - `1` Remembers previous usage of a program, by maintaining a keyword file from program to program. These files are normally stored in the current directory, but can optionally be stored in one common directory if the environment variable {bf NEMODEF}footnote{mirtool also uses this environment variable} is set. The keyword files have the name {{it "progname"}{bf.def}}, {it e.g.} {tt snapshot.def}footnote{This may result in long filenames, Unix SYS5 allows only 14 characters - a different solution is needed here}. When using this lowest help-level it is still possible to use UNIX I/O redirection. This help level reads, as well as writes the keyword file during the program execution; hence the user needs both read and write permission in the keyword directory. As can also be seen, programs cannot run in parallel while using this help-level: they might compete for the same keyword file. Within the simple commandline interface it is not possible to maintain a global keyword database, as is {it e.g.} the case in AIPS; you would have to use the {tt miriad} shell.
  - `2` prompts the user for a (new) value for every keyword; it shows the default (old) value on the prompt line, which can then be edited. It is not possible to combine this level with UNIX I/O redirection. By combining the previous helplevel with this one, previous values and modified ones are maintained in a keyword file.
  - `4` provides a simple fullscreen menu interface, by having the user edit the keyword file. The environment variable {bf EDITOR} can be used to set any other editor than good old {it vi(1)}. It is not possible to combine this level with UNIX I/O redirection.
  - `8,16,...` although not processed, higher powers of 2 are reserved for future options

    Example: `help=3` will remember old keywords in a local keyword file, prompt you with new values, and puts the new values in the keyword file for the next time. The `help=5` option happen to be somewhat similar to the way `AIIPS` and `IRAF` appear to the user.

    Help levels can also include an alpha-string, which generally display the values of the keyword, their default values or their help strings.
  - ? lists all these options, as a reminder. It also displays the version index{version, user interface} of the {tt getparam} user interface package.

- h list all the keywords, plus a help string what the keywords does/expects. This is really what we call the inline manual or inline help. index{inline, help} index{manual, inline} index{help, inline}
- a list all arguments in the form {it keyword=value}.
- p,k list parameters (keywords) of all arguments in the form {it keyword}.
- d,v list defaults (values) of all arguments in the form {it value}.
- n add a newline to every {it keyword/value} string on output. In this way a keyword file could be build manually by redirecting this output.
- t output a documentation file according to the %N,%A specifications index{mirtool} of {tt miriad}footnote{Both {tt mirtool} and {tt miriad} need such a doc-file index{doc file, miriad} to lookup keywords and supply help}. Is mainly intended to be used by scripts such as {tt mktool}. The procedure in NEMO to update a {tt .doc} file would be:

```
% program help=t > $NEMODOC/program.doc
```

- q quit, do not start program. Useful when the helpstring contains options to print.

  Example: **key=val help=1q** redefines a keyword in the keywordfile, but does not run the program. This is also a way to 'repair' a keyword file, when the program has been updated with new keywords. **key=val help=1aq** redefines the keyword, shows the results but does still not run the program. Finally, **key=val help=1a** redefines a keyword, shows the result and then runs the program.

- **debug=** Changes the debug output level. The higher the debug level, the more output can appear on the standard error output device stderr. The default value is either 0 or the value set by the **DEBUG** environment variable. The use of the debug= keyword will override your default setting. A value of '0' for debug may still show some warning messages. Setting debug to -1 will prevent even those warning/debug messages. Legal values are 0 through 9. Values of **DEBUG** higher than 9 are not used, or you may get some weird screen output. Values larger than 5 cause an error to coredump, which can then be used with debug utilities like *abd(1)* and *gdb(1)*.
- **error=** Specifies how many times the fatal error routine can be bypassed. The **ERROR** environment variable can also be set for this. The default, if neither of them present, is 0.
- **yapp=** Defines the device to which graphics output is send. Currently only interpreted for a limited number of yapp devices. Some yapp devices do not even listen to this keyword. Check *yapp(5NEMO)* or your local NEMO guru which one is installed. The default device is either 0 or the value set by the **YAPP** environment variable.
- **np=** Defines the number of processors (e.g. in an OpenMP setting) that can be used. This would override the OMP_NUM_THREADS environment variable, if it was present.
- **outkeys=** TBD
- **argv=** TBD

2.6 YAPP

2.6.1 yapp_ps

By default NEMO is compile with a very simple PostScript device driver, as specified in yapp_ps. This YAPP interface produces a simple PS (supposedly correctly calibrated to be 20 x 20 cm), and the yapp= keyword value specifies the PS filename.

2.6.2 yapp_pgplot

The YAPP interface to the common PGPLOT library is the most used interface, and allow one to select from a variety of graphics output devices without having to recompile the program.

A graphics device in PGPLOT is defined by preceding it with a slash Optional parameters (e.g. filename, X device etc.) can be supplied before the slash. The following list gives an overview of some of the available devices (your list may be a lot shorter (see ?) in list below):

```
   ?           Get a list of all currently defined graphics devices
   /XTERM     (XTERM Tek terminal emulator)
   /XWINDOW   (X window window@node:display.screen/xw)
   /XSERVE    (A /XWINDOW window that persists for re-use)
Non-interactive file formats:
   /NULL      (Null device, no output)
   /PNG       (Portable Network Graphics file)
   /TPNG      (Portable Network Graphics file - transparent background)
   /PS        (PostScript file, landscape orientation)
   /VPS       (PostScript file, portrait orientation)
   /CPS       (Colour PostScript file, landscape orientation)
   /VCPS      (Colour PostScript file, portrait orientation)
   /EPS       (Encapsulated Postscript, colour)
```

See also manual pages such as *getparam(3NEMO)* and *yapp(5NEMO)*

A special script yapp_query is available for **yapp_pgplot** in order to provide script writers with a way to select between possibly not implemented device drivers

```
dev=$(yapp_query png ps gif)
mkplummer - 100 | snapplot -  yapp=fig1.$dev/$dev
```

CHAPTER

THREE

FILESTRUCTURE (*)

Note: NEMO stores its persistent data in binary files, which under most circumstances can also be used in a Unix pipe by using the `-` symbol. The `tsf` program will show the contents of such files in more human readable form.

Here we give an overview of the file structure of NEMO's persistent data stored on disk. The popular memory (object) models, and how they interact with persistent data on disk, are discussed in *Programmers Guide (*)*. Most of the data handled by NEMO is in the form of a specially designed XML-like binary format (well before XML was conceived) although exceptions like ASCII files/tables will also be discussed. Ample examples illustrate creation, manipulation and data transfer. We also mention a few examples of function descriptors, a dataformat that make use of the native object file format of your operating system (*a.out(5)* and *dlopen(3)*) that are dynamically loaded during runtime.

3.1 Binary Structured Files

Note: There is also a program called **bsf**, which benchmarks a regression value of the floating point values in the file.

Most of the data files used by NEMO share a common low level binary file structure, which can be viewed as a sequence of tagged data items. Special symbols are defined to group these items hierarchically into sets. Data items are typically scalar values or homogeneous arrays constructed from elementary C data types, but the programmer can also add more complex structures, such as C's `struct` structure definition, or any user defined data structure. In this last case tagging by type is not possible anymore, and support for a machine independent format is not guaranteed. Using such constructs is not recommended if the data needs to be portable accross platforms.

The hierarchical structure of a binary file in this general format can be viewed in human-readable format at the terminal using a special program, `tsf` ("*type structured file*"). Its counterpart, `rsf` ("*read structured file*"), converts such human-readable files (in that special ASCII Structured File format (ASF) into binary structured files (BSF). In principle it is hence possible to transfer data files between different types of computers using `rsf` and `tsf` (see examples in Section~ref{s:exch-data}).

Let us start with a small example: With the NEMO program `mkplummer` we first create an N-body realization of a spherical Plummer model:

```
1% mkplummer i001.dat 1024
```

Note that we made use of the shortcut that `out=` and `nbody=` are the first two *program keywords*, and they were assigned their value by position rather than by associated name. We can now display the contents of the binary file `i001.dat` with `tsf`:

```
2% tsf i001.dat

char Headline[33] "set_xrandom: seed used 706921861"
char History[36] "mkplummer i001.dat 1024 VERSION=2.5"
set SnapShot
  set Parameters
    int Nobj 01750
    double Time 0.00000
  tes
  set Particles
    int CoordSystem 0201402
    double Mass[1024] 0.00195313 0.00195313 0.00195313
      0.00195313 0.00195313 0.00195313 0.00195313 0.00195313
      0.00195313 0.00195313 0.00195313 0.00195313 0.00195313
      0.00195313 0.00195313 0.00195313 0.00195313 0.00195313
      . . .
    double PhaseSpace[1024][2][3] 4.92932 0.425103 -0.474249
      0.342025 -0.112242 4.60796 -0.00388599 -0.389558 -0.958787
      0.220561 0.213904 3.47561 0.0176012 1.22146 -0.903484
      -0.705422 4.26963 -0.263561 1.04382 -0.199518 -0.480749
      . . .
  tes
tes
```

This is an example of a data-file from the N-body group, and consists of a single *snapshot* at time=0.0. This snapshot, with 1024 bodies with double precision masses and full 6 dimensional phase space coordinates, totals 57606 bytes, whereas a straight dump of only the essential information would have been 57344 bytes, a mere 0.5% overhead. The overhead will be larger with small amounts of data, *e.g.* diagnostics in an N-body simulation, or small N-body snapshots.

Besides some parameters in the `Parameters` set, it consists of a `Particles` set, where (along the type of coordinate system) all the masses and phase space coordinates of all particles are defined. Note the convention of integers starting with a `0` in octal representation. This is done for portability reasons.

A comment about **online** help: NEMO uses the Unix *man(5)* format for more detailed online help, although the **inline** help (system `help=` keyword) is most of the times sufficient enough to remind a novice user of the keywords and their meaning. The `man` command is a last resort, if more detailed information and examples are needed.

```
3% man tsf
```

Note that, since the online manual page is a different file from the source code, information in the manual page can easily get outdated, and the inline (`help=`) help, although very brief, is more likely to be up to date since it is generated from the source code (executable) itself:

```
4% tsf help=h
in                 : input file name  [???]
maxprec            : print nums with max precision  [false]
maxline            : max lines per item  [4]
allline            : print all lines (overrides maxline)  [false]
indent             : indentation of compound items  [2]
margin             : righthand margin  [72]
item               : Select specific item []
xml                : output data in XML format? (experimental) [f]
octal              : Force integer output in octal again? [f]
```

(continues on next page)

(continued from previous page)

```
VERSION          : 29-aug-02 PJT  [3.1]
```

3.2 Pipes

In the UNIX operating system pipes can be very effectively used to pass information from one process to another. One of the well known textbook examples is how one gets a list of misspelled (or unknown) words from a document:

```
% spell file | sort | uniq | more
```

NEMO programs can also pass data via UNIX pipes, although with a slightly different syntax: a dataset that is going to be part of a pipe (either input or output) has to be designated with the `-` (*dash*) symbol for their filename. Also, and this is very important, the receiving task at the other end of the pipe should get data from only one source. If the task at the sending end of the pipe wants to send binary data over that pipe, but in addition the same task would also write *normal* standard output, the pipe would be corrupted with two incompatible sources of data. An example of this is the program `snapcenter`. The keyword `report` must be set to `false` instead, which is actually the default now. So, for example, the output of a previous N-body integration is re-centered on it's center of mass, and subsequently rectified and stacked into a single image as follows:

```
% snapcenter r001.dat . report=t  | tabplot - 0 1,2,3

% snapcenter r001.dat - report=f        |\
    snaprect - - 'weight=-phi*phi*phi' |\
    snapgrid - r001.sum stack=t
```

If the keyword `report=f` would not have been set properly, `snaprect` would not have been able to process it's convoluted input. Some other examples are discussed in Section~ref{ss:data}.

3.3 History of Data Reduction

Most programs in NEMO will automatically keep track of the history of their data-files in a self-describing and self-documenting way. If a program modifies an input file and produces an output file, it will prepend the command-line with which it was invoked to its data history. The data history is normally located at the beginning of a data file. Comments entered using the frequently used program keyword `headline=` will also appear in the history section of your data file.

A utility, `hisf` can be used to display the history of a data-file. This utility can also be used to create a pure history file (without any data) by using the optional `out=` and `text=` keywords. Of course `tsf` could also be used by scanning its output for the string `History` or `Headline`:

```
5% tsf r001.dat | grep History
```

which shows that `tsf`, together with it's counterpart `rsf` has virtually the same functionality as `hisf`.

3.4 Table format

Many programs are capable of producing standard output in (ASCII) tabular format. The output can be gathered into a file using standard UNIX I/O redirection. In the example

```
6% radprof r001.dat tab=true > r001.tab
```

the file `r001.tab` will contain (amongst others) columns with surface density and radius from the snapshot `r001.dat`. These (ASCII) *table* files can be used by various programs for further display and analysis. NEMO also has a few programs for this purpose available (*e.g.*} `tabhist` for analysis and histogram plotting, `tablsqfit` for checking correlations between two columns and `tabmath` for general table handling. The manual pages of the relevant NEMO programs should inform you how to get nice tabular output, but sometimes it is also necessary to write a shell/awk script or parser to do the job.

A usefull (open source domain) program *redir(1NEMO)* has been included in NEMO

```
7% redir -e debug.out tsf r001.dat debug=2
```

would run the `tsf` command, but redirecting the *stderr* standard error output to a file `stderr.out`. There are ways in the C-shell to do the same thing, but they are clumsy and hard to remember. In the bourne/bash shell this is accomplished much easier:

```
7$ tsf r001.dat debug=2  2>debug.out
```

One last word of caution regarding tables: tables can also be used very effectively in pipes, for example take the first example, and pipe the output into `tabplot` to get a quick look at the profile:

```
8% snapprint r001.dat r | tabhist -
```

If the snapshot contains more than 10,000 points, `tabhist` cannot read the remainer of the file, since the default maximum number of libes for reading from pipes is set by a keyword `nmax=10000`. To properly read all lines, you have to know (or estimate) the number of lines. In the other case where the input is a regular file, table programs are always able to find the correct amount to allocate for their internal buffers by scanning over the file once. For very large tables this does introduce a little extra overhead.

3.5 Dynamically Loadable Functions

A very peculiar data file format encountered in NEMO is that of the function descriptors. They present themselves to the user through one or more keywords, and in reality point to a compiled piece of code that will get loaded by NEMO (using *loadobj(3NEMO)*). We currently have 4 of these in NEMO:

3.5.1 Potential Descriptors

The potential descriptor is used in orbit calculations and a few N-body programs. These are actually binary object files (hence extremely system dependent!!), and used by the dynamic object loader during runtime. Potentials are supplied to NEMO programs as an input variable (*i.e.* a set of keywords, normally called `potname=`, `potpars=` and `potfile=`. For this, a mechanism is needed to dynamically load the code which calculates the potential. This is done by a dynamic object loader that comes with NEMO. If a program needs a potential, and it is present in the default repository (`$POTPATH` or {`$NEMOOBJ/potential`), it is directly loaded into memory by this dynamic object loader. If only a source file is present, *e.g.* in the current directory, it is compiled on the fly and then loaded. The source code can be written in C or FORTRAN. Rules and more information can be found in *potential(3NEMO)* and *potential(5NEMO)* The program *potlist(1NEMO)* can be used to test potential descriptors.

3.5.2 Bodytrans Functions

Another family of object files used by the dynamic object loader are the *bodytrans(5NEMO)* functions. These were actually the first one of this kind introduced in NEMO. They are functions generated from expressions containing body-variables (mass, position, potential, time, ordinal number etc.). They frequently occur in programs where it is desirable to have an arbitrary expression of body variables *e.g.* plotting and printing programs, sorting program etc. Expressions which are not in the standard repository (currently `$BTRPATH` or `$NEMOOBJ/bodytrans`) will be generated on the fly and saved for later use. The program *bodytrans(1NEMO)* is available to test and save new expressions. Examples are given in Section~ref{s-dispanal}, a table of the precompiled ones are in Table~ref{t:bodytrans}.

3.5.3 Nonlinear Least Squares Fitting Functions

The program *tabnllsqfit(1NEMO)* can fit (linear or non-linear, depending on the parameters) a function to a set of datapoints from an ASCII table. The keyword `fit=` describes the model (*e.g.* a line, plane, gaussian, circle, etc.), of which a few common ones have been pre-compiled with the program. In that sense this is different from the previous two function descriptors, which always get loaded from a directory with precompiled object files. The keyword `load=` can be used to feed a user defined function to this program. The manual page has a lot more details.

3.5.4 Rotation Curves Fitting Functions

Very similar to the Nonlinear Least Squares Fitting Functions are the Rotation Curves Fitting Functions, except they are peculiar to the 1- and 2-dimensional rotation curves one find in galaxies as the result of a projected circular streaming model. The program *rotcurshape(1NEMO)* is the only program that uses these functions, the manual page has a lot more details.

CHAPTER FOUR

GRAPHICS AND IMAGE DISPLAY (*)

Note: Most NEMO graphics programs select their graphics output with the `yapp=` system keyword.

NEMO programs also need to display their data of course. Here we will make a distinction between *graphics* and *image* data. A simple but flexible *graphics* interface has been defined in NEMO and is used extensively in programs that need such output.

To display *image* data we rely mostly (but see *ccdplot(1NEMO)* for an exception) on external software. Often images would need to be copied to a FITS file for this (but see `nds9` for an example that can use NEMO's image format).

4.1 The YAPP graphics interface

The programs in NEMO which use *graphics* are rather simple and generally allow no interactive processing, except perhaps for a simple 'hit-the-return-key' or 'push-a-mouse-button' between successive plots or actions. A very simple interface (API) was defined (**yapp**, Yet Another Plotting Package) with basic plot functions. There are currently a few yapp implementations available, such as a postscript-only device, and pgplot. If your output device is not supported by the ones available in the current yapp directory (`$NEMO/src/kernel/yapp`), you may have to write a new one! A reasonably experienced programmer writes a functional yapp-interface in a few hours.

Although this method results in a flexible graphics interface, a program can currently only be linked with one yapp-interface, which is selected at NEMO's installation time via the **configure** script. This might result in the existence of more than one version of the same program, each for another graphics output device. We use the convention that the ones for a postscript printer have a {tt _ps} appended to their original name: the program which has the original name is the one whose display is the current screen, Hence we may see program names such as {tt snapplot} (the default), `snapplot_ps` (postscript), or `snapplot_cg` (color Sun screen when this was popular) . Again: actual names may differ on your system.

If programs are linked with the multiplexing libraries *yapp_pgplot* interface, several device drivers are transparently present through pgplot, and the system keyword `yapp=` is then used to select a device (a default can be set by using the **YAPP** environment variable). See also Appendix~ref{a:iface}.

However, despite these grim sounding words, we currently almost exclusively use the PGPLOT implementation of yapp, which is very flexible. You will need to have pgplot installed on your system, which is another story in itself.

4.2 pyplot=: python matplotlib

An experimental `pyplot=` keyword has been added to `tabplot` and `tabhist`, which creates a simple python script that reproduces (to some degree) what those program intended to do. The intent is that these can be edited and made more functionality.

4.3 General Graphics Display

Another convenient way to present data in graphical form is by using the table format. We have already encountered the *tables* created by many NEMO programs. These tables can be used by NEMO programs such as *tabplot(1NEMO)*, *tabhist(1NEMO)*, and other packages such as *gnuplot*, *xgobi*, *xmgrace*, *xgraphic*, and *glueviz*.

4.4 Image Display Interface

Data in *image(5NEMO)* format can be transferred in *fits(5NEMO)* format and subsequently displayed and analyzed within almost any astronomical image processing system. They are generally much better equipped to display and manipulate data of this kind of format. A number of standalone display programs can also understand FITS format. An excellent example of this is *ds9*, although it understands FITS files, can be used in a client-server setting and NEMO image files can be directly sent to the display server (a temporary fits file is created, which can have drawbacks if they are large):

```
% ds9 &
% nds9 map.ccd
```

Other programs you can consider are: *carta*, *qfitsview*, *ginga*, and *fv*. See also the list on https://blends.debian.org/astro/tasks/viewers

CHAPTER

FIVE

EXAMPLES (*)

Now that we have a reasonable idea how NEMO is structured and used, we should be ready to go through some real examples Some of the examples below are short versions of scripts available online in one of the directories (check `$NEMO/scripts` and `$NEMOBIN`). The manual pages *programs(8NEMO)* and *intro(1NEMO)* are useful to find (and cross-reference) programs if you're a bit lost. Each program manual should also have some references to closely related programs, in the "SEE ALSO" section.

5.1 N-body experiments

In this section we will describe how to set up an N-body experiment, run, display and analyze it. In the first example, we shall set up a head-on collision between two spherical "galaxies" and do some simple analysis.

5.1.1 Setting it up

In *Filestructure (*)* we already used `mkplummer` to create a Plummer model; so here we shall use the program `mkommod` (*"MaKe an Osipkov-Merritt MODel"*) to make two random N-body realizations of a King model with dimensionless central potential $W_c = 7$ and 100 particles each. The small number of particles is solely for the purpose of getting results within a reasonable time. Adjust it to whatever you can afford on your CPU and test your patience and integrator.

```
1% mkommod in=$NEMODAT/k7isot.dat out=tmp1 nbody=100 seed=123
```

These models are produced in so-called RMS-units in which the gravitational constant $G = 1$, the total mass $M = 1$, and binding energy $E = -1/2$. In case you would like virial units (see also: Heggie & Mathieu, $E = -1/4$, in: *The use of supercomputers in stellar dynamics* ed. Hut & McMillan Springer 1987, pp.233) the models have to be rescaled using `snapscale`:

```
2% snapscale in=tmp1 out=tmp1s rscale=2 "vscale=1/sqrt(2.0)"
```

Note the use of the quotes in the expression, to prevent the shell to give special meaning to the parenthesis, which are shell **meta** characters.

The second galaxy is made in a similar way, with a different seed:

```
3% mkommod in=$NEMODAT/k7isot.dat out=tmp2 nbody=100 seed=987
```

This second galaxy needs to be rescaled too, if you want virial units:

```
4% snapscale in=tmp2 out=tmp2s rscale=2 "vscale=1/sqrt(2.0)"
```

We then set up the collision by stacking the two snapshots, albeit with a relative displacement in phase space. The program `snapstack` was exactly written for this purpose:

```
5% snapstack in1=tmp1s in2=tmp2s out=i001.dat deltar=4,0,0 deltav=-1,0,0
```

The galaxies are initially separated by 4 unit length and approaching each other with a velocity consistent with infall from infinity (parabolic encounter). The particles assembled in the data file `i001.dat` are now ready to be integrated.

To look at the initials conditions we could use:

```
6% snapplot i001.dat xrange=-5:5 yrange=-5:5
```

which is displayed in Figure X

5.1.2 Integration using hackcode1

We then run the collision for 20 time units, with the standard N-body integrator based on the Barnes "hierarchical tree" algorith:

```
7% hackcode1 in=i001.dat out=r001.dat tstop=20 freqout=2 \
   freq=40 eps=0.05 tol=0.7 options=mass,phase,phi > r001.log
```

The integration frequency relates to the integration timestep as `freq=40`, the softening length `eps=0.05`, and opening angle or tolerance `tol=$\theta$`. A major output of masses, positions and potentials of all particles is done every `1/freqout = 0.5` time units, which corresponds to about 1/5 of a crossing time. The standard output of the calculation is diverted to a file `r001.log` for convenience. This is an (ASCII) listing, containing useful statistics of the run, such as the efficiency of the force calculation, conserved quantities etc. Some of this information is also stored in diagnostic sets in the structured binary output file `r001.dat`.

As an exercise, compare the output of the following two commands:

```
8% more r001.log
9% tsf r001.dat | more
```

5.1.3 Display and Initial Analysis

As described in the previous subsection, `hackcode1` writes various diagnostics in the output file. A summary of conservation of energy and center-of-mass motion can be graphically displayed using `snapdiagplot`:

```
10% snapdiagplot in=r001.dat
```

The program `snapplot` displays the evolution of the particle distribution, in projection (in the **yt** package this is called a phase plot):

```
11% snapplot in=r001.dat
```

Depending on the actual graphics (*yapp*) interface snapplot was compiled with, you may have to hit the RETURN key, push a MOUSE BUTTON or just WAIT to advance from one to the next frame.

The `snapplot` program has a very powerful tool built into it which makes it possible to display any *projection* the user wants.

As an example consider:

```
12% snapplot in=r001.dat xvar=r yvar="x*vy-y*vx" xrange=0:10 yrange=-2:2 \
            "visib=-0.2<z&&z<0.2&&i%2==0"
```

plots the angular momentum of the particles along the z axis, $J_z = x.v_y - y.v_x$, against their radius, r, but only for the even numbered particles, (`i%2==0`) within a distance of `0.2` of the X-Y plane ($-0.2 < zz < 0.2$). Again note that some of the expressions are within quotes, to prevent the shell of giving them a special meaning.

The `xvar`, `yvar` and `visib` expressions are fed to the C compiler (during runtime!) and the resulting object file is then dynamically loaded into the program for execution. The expressions must contain legal C expressions and depending on their nature must return a value in the context of the program. {it E.g.} {tt xvar} and {tt yvar} must return a real value, whereas {tt visib} must return a boolean (false/true or 0/non-0) value. This should be explained in the manual page of the corresponding programs.

In the context of snapshots, the expression can contain basic body variables which are understood to the *body-trans(3NEMO)* routine. The real variables {tt x, y, z, vx, vy, vz} are the cartesian phase-space coordinates, {tt t} the time, {tt m} the mass, {tt phi} the potential, {tt ax,ay,az} the cartesian acceleration and {tt aux} some auxiliary information. The integer variables are {tt i}, the index of the particle in the snapshot (0 being the first one in the usual C tradition) and {tt key}, another spare slot.

For convenience a number of expressions have already been pre-compiled (see also Table ref{t:bodytrans}), *e.g.* the radius r= $\sqrt{x^2 + y^2 + z^2}$ = `sqrt(x*x+y*y+z*z)`, and velocity v = $\sqrt{v_x^2 + v_y^2 + v_z^2}$ = `sqrt(vx*vx+vy*vy+vz*vz)`. Note that {tt r} and {tt v} themselves cannot be used in expressions, only the basic body variables listed above can be used in an expression.

When you need a complex expression that has be used over and over again, it is handy to be able to store these expression under an alias for later retrieval. With the program {tt bodytrans} it is possible to save such compiled expressions object files under a new name.

Table 1: Some precompiled *bodytrans* expressions

name	type	expression
0	int	`0`
1	int	`1`
i	int	`i`
key	int	`key` (see also *real* version below)
0	real	`0.0`
1	real	`1.0`
r	real	`sqrt(x*x+y*y+z*z)`
		or: ||
ar	real	`sqrt(x*ax+y*ay+z*az)/sqrt(x*x+y*y+z*z)`
ar	real	`(x*ax+y*ay+z*az)/sqrt(x*x+y*y+z*z)` or: (rvec$cdot$avec)/$\|$rvec$\|$ \
aux	real	`aux`
ax	real	`ax`
ay	real	`ay`
az	real	`az`
etot	real	`phi+0.5*(vx*vx+vy*vy+vz*vz)` or: phi + vvec^2/2 \
i	real	`i`
jtot	real	`sqrt(sqr(x*vy-y*vx)+sqr(y*vz-z*vy)+sqr(z*vx-x*vz))` or: $\|$rvec$times$vvec$\|$ \
key	real	`key` (see also **int**} version above)
m	real	`m`
phi	real	`phi`
r	real	`sqrt(x*x+y*y+z*z)`
		or: $\|$rvec$\|$ \
t	real	`t`
v	real	`sqrt(vx*vx+vy*vy+vz*vz)`
		or: $\|$vvec$\|$
vr	real	`(x*vx+y*vy+z*vz)/sqrt(x*x+y*y+z*z)`

continues on next page

Table 1 – continued from previous page

name	type	expression
		or: or: (rvec$cdot$vvec)/$\|$rvec$\|$
vt	real	`sqrt((vx*vx+vy*vy+vz*vz)-sqr(x*vx+y*vy+z*vz)/(x*x+y*y+z*z))`
		or: $sqrt{ }$(vvec^2-(rvec$cdot$vvec)^2/$\|$rvec$\|^2$)\
vx	real	`vx`
vy	real	`vy`
vz	real	`vz`
x	real	`x`
y	real	`y`
z	real	`z`
glon	real	l , `atan2(y,x)*180/PI` [-180,180]
glat	real	b , `atan2(z,sqrt(x*x+y*y))*180/PI` [-90,90]
mul	real	`(-vx\sin{l} + vx\cos{l})/r`
mub	real	$(-vx \cos l \sin b - vy \sin l \sin b + vz \cos b)/r$
xait	real	Aitoff projection x [-2,2] T.B.A.
yait	real	Aitoff projection y [-1,1] T.B.A.

As usual an example:

```
13% bodytrans expr="x*vy-y*vz" type=real file=jz
```

saves the expression for the angular momentum in a real valued bodytrans expression file, {tt btr_jz.o} which can in future programs be referenced as {tt expr=jz} (whenever a real-valued bodytrans expression is required), {it e.g.}

```
14% snapplot i001.dat xvar=r yvar=jz xrange=0:5
```

Alternatively, one can handcode a {it bodytrans} function, compile it, and reference it locally. This is useful when you have truly complicated expressions that do not easily write themselves down on the commandline. The (x,y) AITOFF projection are an example of this. For example, consider the following code in a (local working directory) file {tt btr_r2.c}:

```
#include <bodytrans.h>

real btr_r2(b,t,i)
Body *b;
real t;
int  i;
{
    return sqrt(x*x + y*y);
}
```

By compiling this:

```
15% cc -c btr_r2.c
```

an object file {tt btr_r2.o} is created in the local directory, which could be used in any real-valued bodytrans expression:

```
16% snapplot i001.dat xvar=r2 yvar=jz xrange=0:5
```

For this your environment variable {bf BTRPATH} must have been set to include the local working directory, designated by a dot. Normally your NEMO system manager will have set the search path such that the local working directory is searched before the system one (in {tt $NEMOOBJ/bodytrans}).

5.1.4 Advanced Analysis

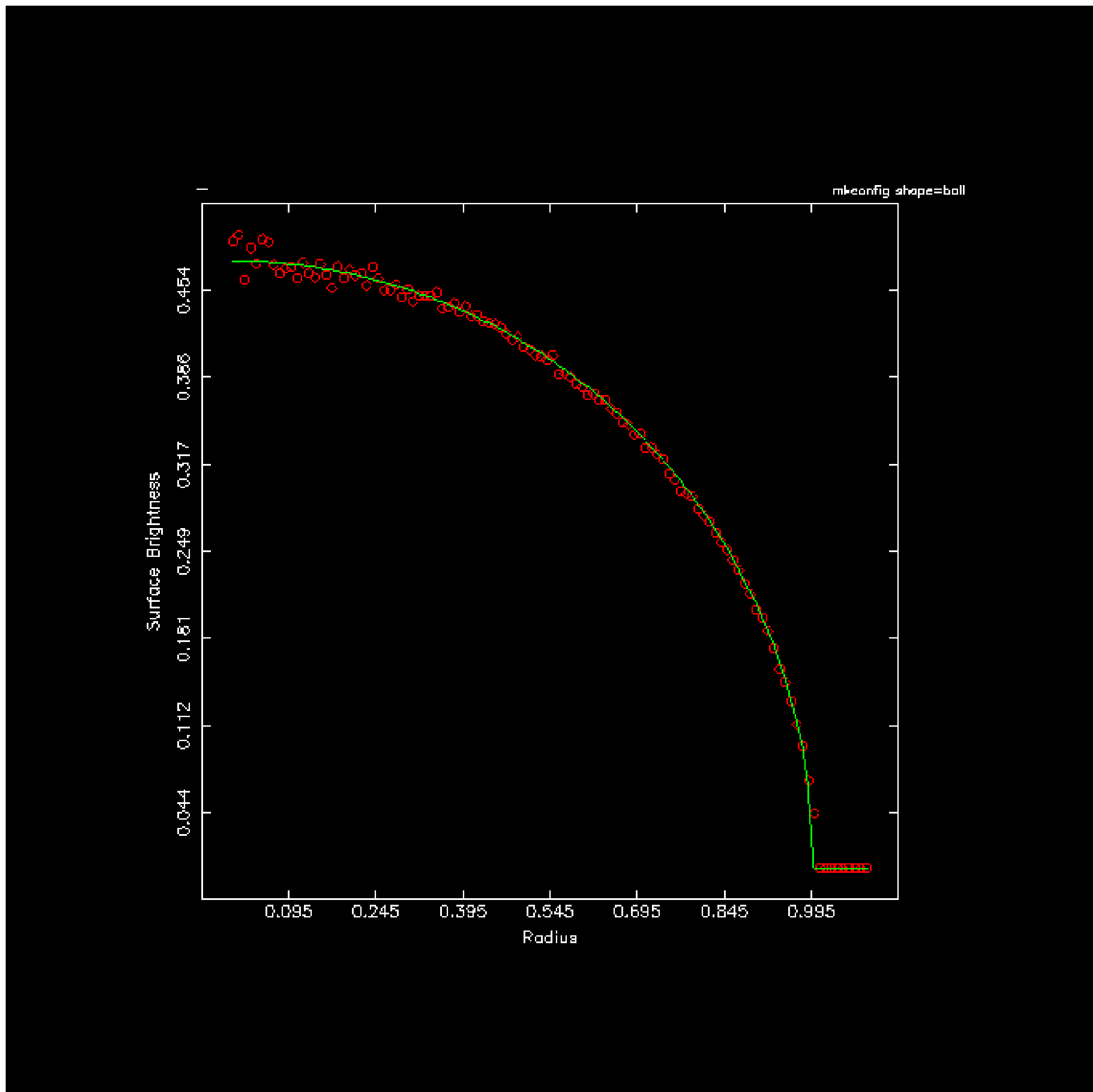
5.1.5 Generating models

5.1.6 Using Unix pipes

In most cases a NEMO file can be used in a pipe (usually via `in=-` and `out=-`), therefore limiting the need to write files. Here is an example of plotting the measured and expected surface brightness of a homogeneous sphere of 1,000,000 particles with unit mass and unit radius:

```
% mkconfig - 1000000 ball seed=0 |\
    snapgrid - - nx=800 ny=800  |\
    ccdellint - 0:1.1:0.01 inc=0 out=- |\
    ccdprint - x=  newline=t label=x |\
    tabmath - - '1.5/pi*sqrt(1-%1**2)' |\
    tabplot -  1 2,3 color=2,3 line=0,0,1,1 point=2,0.1,0,0 \
      xlab="Radius" ylab="Surface Brightness" headline="mkconfig shape=ball" yapp=ball.
↪png/png
```

A few comments on the highlighted lines: In **line 3** the `out=` keyword is not the second keyword, hence the explicit way it was written with the `out=-`. In **line 5** the expected surface brightness expression is added as the 3rd column to the table in the pipe, then passed on for a quick and dirty plot (shown below).



5.1.7 Handling large datasets

One of NEMOs weaknesses is also it's strong point: programs must generally be able to fit all their data in (virtual) memory. Although programs usually free memory associated with data that is not needed anymore, there is a very clear maximum to the number of particles it can handle in a snapshot. By defaultfootnote{one can recompile NEMO in single precision and define {tt body.h} with less wastefull members} a particle takes up about 100 bytes, which limits the size of a snapshots on workstations.

It may happen that your data was generated on a machine which had a lot more memory then the machine you want to analyze your data on. As long as you have the diskspace, and as long as you don't need programs that cannot operate on data in serial mode, there is a solution to this problem. Instead of keeping all particles in one snapshot, they are stored in several snapshots of (equal number of) bodies, and as long as all snapshots have the same time and are stored back to back, most programs that can operate serially, will do this properly and know about it. Of course it's best to

split the snapshots on the machine with more memory

```
% snapsplit in=run1.out out=run1s.out nbody=10000
```

If it is just one particular program (e.g. snapgrid that needs a lot of extra memory, the following may work:

```
% snapsplit in=run1.out out=- nbody=1000 times=3.5 |\
    snapgrid in=- out=run1.ccd nx=1000 ny=1000 stack=t
```

Using *tcppipe(1NEMO)* one can also pipe data from the machine with large memory (A) to your workstation with smaller memory (B), as can be demonstrate with the following code snippet:

```
A% mkplummer - 1000 | tcppipe
B% tcppipe A | tsf -
```

5.2 Images

Todo: examples/Images

5.3 Tables

Todo: examples/Tables

5.4 Potential

Todo: examples/Potential

5.5 Orbits

In this section we will describe how to integrate individual stellar orbits, display and analyze them. Be aware that although 3D orbits can be computed the number of utilities to analyze them is rather limited.

Orbits are normally stored in datafile (see also {it orbit(5NEMO)}), and a close conceptual relationship exists between a (single-particle type) {bf snapshot} and an {bf orbit}: an orbit is an ordered series of phase-space coordinates whereas a snapshot is a series of particles with no particular order, but all at the same time.

Since orbits will be computed in an analytical potential, we assume for the remainder of this section that you have familiarized yourself with how to supply potentials to orbit integrator programs. They all share the same triple `potname=, potpars=, potfile=` keyword interface, as described in Section ref{s:potential}. Many examples of the tricky {tt potpars=} keyword are given in Appendix ref{a:potential}.

5.5.1 Initializing

There are a few programs with which orbits can be initialized:

- **mkorbit** is the most straightforward program. You can give simply give it all 6 phase space coordinates, and an orbit file consisting of this one point is generated. It is also possible to give the potential in which the particle is to move, and 5 phase space coordinates plus the energy, or even 4 phase space coordinates and the energy plus the total angular momentum or angular momentum along the Z axis (for axisymmetric systems).

Let's start with an example of creating a simple orbit by itself with no associated potential.

```
% mkorbit out=orb1 x=1 y=0 z=0 vx=0 vy=0.2 vz=0
### Warning [mkorbit]: Potential potname= not used; set etot=0.0
pos: 1.000000 0.000000 0.000000
vel: 0.000000 0.200000 0.000000
etot: 0.000000
lz=0.200000

% tsf orb1
char History[59] "mkorbit out=orb1 x=1 y=0 z=0 vx=0 vy=0.2 vz=0 VERSION=3.2b"
set Orbit
  set Parameters
    int Ndim 03
    double Mass 1.00000
    double IOM[3] 0.00000 0.200000 0.00000
    int Nsteps 01
  tes
  set Potential
  tes
  set Path
    double TimePath[1] 0.00000
    double PhasePath[1][2][3] 1.00000 0.00000 0.00000 0.00000 0.200000 0.00000
  tes
tes
```

- **perorb** is a program that for given initial conditions (similar to the ones described in {tt mkorbit} above) attempts to calculate periodic orbits in that potential. The output file will be a file with one (or more) orbits. This is a bit of an advanced program, and will not be covered here.
- **stoo** is a program that can take a particle position from a snapshot, and turn it into an orbit. For example, sampling some initial conditions from the positions of stars in a Plummer sphere, we could use the following small C-shell code to find some statistical properties of this selected set of orbitsfootnote{For the careful reader: {tt mkplummer} and {tt potname=plummer} actually have different units, and as such this experiment is not properly set up.}

```
mkplummer out=p100 nbody=p100
foreach i (`nemoinp 0:100:10`)
    stoo in=p100 out=orb$i ibody=$i
    orbint orb$i orb$i.out 10000 0.01 10000 potname=plummer
    orbstat orb$i.out
end
```

The reverse program, {tt otos} turns an orbit into a snapshot, and may come in handy since the snapshot package has far more advanced analysis programs.

5.5.2 Integration

- **orbint** integrates orbits from given initial conditions. If the input orbit has more than 1 step, the last step is taken as the initial conditions. Although the {tt potname=, potpars=, potfile=} keywords can be given, if the input orbit contains…
- **perorb** finds periodic orbits, and stores a full period which should close the orbit. This program finds periodic orbits in the XY plane (i.e. currently it will only find 2D orbits) by searching for the centers of invariant curves in the surface of section.
- **henyey** also finds periodic orbits, but uses Henyey's methodfootnote{see also van Albada & Sanders, (1982, MNRAS, 201, 303)}. This program has however not been released to the public version of NEMO, and in fact it seem the source code was lost.

5.5.3 Display

- **orbplot** is the only orbit plotting program we currently have. For more sophisticated display {tt tabplot} and/or {tt snapplot} would have to be used after transforming the data. Also {tt snapplot} uses the powerful {it bodytrans} expression parser to plot arbitrary body related expressions, although {tt orbplot} can handle both {tt x, y, z} and {tt vx, vy, vz} for the {tt xvar=} and {tt yvar=} keywords. An example of the output of {tt orbplot} is given in Figure ref{f:orbit1}.

5.5.4 Analysis

- **orbstat** is an example of a simple program that reads orbits, and displays statistics of it's 2D (x-y-) coordinates: maximum extent, as well as statistics of the angylar momentum. This program is not suited for 3D orbits yet.

```
% orbint orb1 orb1.long
% orbstat orb1.out
# T     E        x_max    y_max    u_max    v_max    j_mean  j_sigma
1000 -0.687107 1 0.999958 0.746764 0.746611 0.2 3.83111e-09
```

- **orbfour** performs a variety of fourier analysis on the coordinates

```
% orbint orb1 orb1.long 100000 0.01 10000 10 plummer
INIPOTENTIAL Plummer: [3d version]
Pattern speed=0
0.000000 0.020000 -0.707107     -0.6871067811865
100.000000 0.277794 -0.964901     -0.6871067811856
200.010000 0.020912 -0.708019     -0.6871067812165
300.020000 0.271222 -0.958329     -0.6871067812194
400.030000 0.023376 -0.710483     -0.6871067812465
500.040000 0.259253 -0.946360     -0.6871067812551
600.050000 0.027415 -0.714522     -0.6871067812765
700.060000 0.242979 -0.930086     -0.6871067812904
800.070000 0.033056 -0.720163     -0.6871067813065
900.080000 0.223694 -0.910801     -0.6871067813241
Energy conservation: 2.00138e-10
% orbfour orb1.long amode=t
<R> N A0 A1 A2 A3 A4 B1 B2 B3 B4
1 10001 0.000360461 0.334714     0.000150399 -0.000472581 -0.000158864
                   -0.000667155  0.000228086 -0.000725406  0.000103029
```

(continues on next page)

(continued from previous page)

```
% orbfour orb1.long amode=f
<R> N C0 C1 P1 C2 P2 C3 P3 C4 P4
1 10001 0.000360461
        0.334715      -0.114202
        0.000273209   56.5992
        0.000865763 -123.083
        0.000189349  147.035
```

- **orbsos** computes surface of section coordinates. Since this program does not plot, but produces a simple ascii table, you can pipe the output into **tabplot**:

```
% orbsos orb1.long y | tabplot - 3 4  xlab=Y ylab=VY
% orbsos orb1.long x | tabplot - 3 4  xlab=X ylab=VX
```

will plot either a Y-VY or X-VX surface of section.

- **orbdim** computes the dimensionality of an orbit, i.e. how many integrals of motions it has. Although it requires very long integration times to accurately compute this, it is completely automatic, and does not require an analysis like that for a surface of section (which is also graphic). It is based on an interesting paper by Carnevali & Santangelo (1984, ApJ 281 473-476).
- **otos** transforms an orbit back into a snapshot, thereby giving you the much richer set of analysis tools that are available for {it snapshot}'s.

5.6 Exchanging data

Todo: examples/Exchanging Data

5.7 Potential(tmp)

Here we list some of the standard potentials available in NEMO, in a variety of units, so not always $G = 1$!

Recall that most NEMO program use the keywords **potname=** for the identifying name, **potpars=** for an optional list of parameters and **potfile=** for an optional text string,for example for potentials that need some kind of text file. The parameters listed in **potpars=** will always have as first parameter the pattern speed in cases where rotating potentials are used. A Plummer potential with mass 10 and core radius 5 would be hence be supplied as: `potname=plummer potpars=0,10,5`. The plummer potential ignored the **potfile=** keyword.

plummer Plummer potential (BT, pp.42, eq. 2.47)

$$\Phi = -\frac{M}{(r_c^2 + r^2)^{1/2}}$$

- Ω_p : Pattern Speed (always the first parameter in **potpars=**)
- M : Total mass (clearly G=1 here)
- r_c : Core radius

potname=plummer potpars= Ω_p, M, r_c

5.8 Images

5.8.1 Initializing Images

There are a few programs with which images can be initialized:

- **ccdmath** is the most straightforward program. Here is an example of creating an image from scratch:

```
% ccdmath out=ccd1 fie=%x+%y size=2,4
Generating a map from scratch

% tsf ccd1
set Image
  set Parameters
    int Nx 2
    int Ny 4
    int Nz 1
    double Xmin 0.00000
    double Ymin 0.00000
    double Zmin 0.00000
    double Dx 1.00000
    double Dy 1.00000
    double Dz 1.00000
    double MapMin -4.00000
    double MapMax 0.00000
    int BeamType 0
    double Beamx 0.00000
    double Beamy 0.00000
    double Beamz 0.00000
    double Time 0.00000
    char Storage[5] "CDef"
  tes
  set Map
    double MapValues[2][4] -4.00000 -3.00000 -2.00000 -1.00000
      -3.00000 -2.00000 -1.00000 0.00000
  tes
tes

% ccdprint ccd1 x= y= label=x,y
 Y\X 0 1

3  -1 0
2  -2 -1
1  -3 -2
0  -4 -3
```

- **snapgrid** converts a snapshot to an image.
- **fitsccd** converts a FITS file to an image. The inverse of this, **ccdfits** also exists.

```
nx,ny ->    data[nx][ny]

e.g.  ccdmath out=ccd1   nx=10 ny=5
      gives       double MapValues[10][5]
```

(continues on next page)

(continued from previous page)

```
ccdmath "" - %x 3,2 | tsf - margin=100 | grep MapVal
      MapValues[3][2] -2.00000 -2.00000 -1.00000 -1.00000 0.00000 0.00000
ccdmath "" - %y 3,2 | tsf - margin=100 | grep MapVal
      MapValues[3][2] -2.00000 -1.00000 -2.00000 -1.00000 -2.00000 -1.00000
```

5.8.2 Galactic Velocity Fields

As an example, a special section is devoted here to the analysis of galactic velocity fields.footnote{In this example shell variables such as `r=$(nemoinp 0:60)` have been replaced with the more portable macro files like `@tmp.r`. Although the example uses `0:60` and works fine in the shell the example was used under, increasing the number to 256 would fail because of overflowing the maximum characters allowed on the commandline}

The following programs are available:

```
    ccdvel          create a model velocity field, from scratch
    rotcur          tilted ring model velocity field fitting
    rotcurshape     annulus rotation curve shape fitting to a velocity field
    ccdmath         perform math on images, or use math to create images
    ccdplot         plot (contour/greyscale) an image
    ccdprint        print out pixel values in an imamge


% nemoinp 0:60 > tmp.r
% tabmath tmp.r - "100*%1/(20+%1)" all > tmp.v
% ccdvel out=map1.vel rad=@tmp.r vrot=@tmp.v pa=30 inc=60
% rotcurshape in=map1.vel radii=0,60 pa=30 inc=60 vsys=0 units=arcsec,1 \
              rotcur1=core1,100,20,1,1 tab=-


% ccdmath out=map0.vel fie=0 size=128,128
% rotcurshape map0.vel 0,40 30 45 0 blank=-999 resid=map2.vel \
              rotcur1=plummer,200,10,0,0 fixed=all units=arcsec,1
```

Since rotcurshape computes a residual velocity field, one can easily create nice model velocity fields from any selected shape by *fitting* a rotation curve shape to a velocity field of all 0s and keeping all parameters fixed to the requested values:

```
% ccdmath out=map0.vel fie=0 size=128,128
% rotcurshape map0.vel 0,40 30 45 0 blank=-999 resid=map.vel \
           rotcur1=plummer,200,10,0,0 fixed=all units=arcsec,1
% ccdplot map.vel -100:100:10 blankval=0 cmode=1
```

CHAPTER

SIX

USING NEMO

Note: The `nemo_start.(c)sh` file needs to be sourced to set up your shell environment for using NEMO.

CHAPTER

SEVEN

IN ORDER TO USE NEMO, YOU WILL NEED TO MODIFY YOUR

shell environment, for example in the `bash` shell this could be

```
source /opt/nemo/nemo_start.sh
```

assuming NEMO was installed in `/opt/nemo/`, and

```
source /opt/nemo/nemo_start.csh
```

for users of a `csh` like shell. Normally this line would be added to your `~/.bashrc` or `/.cshrc` file.

After this the following commands should work for you

```
tsf --help
man tsf
```

7.1 Updating NEMO

Although a full update is out of scope for the discussion here, a common case is when a program is not available, or a collaborator had made a new or updated program available via github. The following procedure generally works, assuming you have write permission in the `$NEMO` directory tree:

```
mknemo -u tsf
mknemo -h
man mknemo
```

and an updated version should now be available (check the value of the `VERSION=` value in the output of `--help`).

Writing NEMO program programs is covered in *Programmers Guide (*)*, or see also *Installation*.

CHAPTER

EIGHT

INSTALLATION

Note: NEMO is available on https://github.com/teuben/nemo

8.1 Installation from github

Installation is normally done via github. Here is a simple example, just 3 lines in your (bash) shell, using a configurable helper script:

```
wget https://teuben.github.io/nemo/install_nemo.sh
bash install_nemo.sh  nemo=$HOME/opt/nemo yapp=pgplot bench5=1 python=1
source  $HOME/opt/nemo/nemo_start.sh
```

where the arguments to the install_nemo.sh script are optional, but a few are given to show some often use non-defaults. See that script for more details.

A more manual install, bypassing this script, can be:

```
git clone https://github.com/teuben/nemo
cd nemo
./configure --with-yapp=pgplot
make build check bench bench5 python
source nemo_start.sh
```

On a Mac with their new SIP protection, the `--disable-shared` flag needs to be added

```
git clone https://github.com/teuben/nemo
cd nemo
./configure --with-yapp=pgplot --disable-shared
make build check bench bench5
source nemo_start.sh
```

Disabling SIP is not recommended.

8.2 Rebuilding

If you have an existing installation, but many things have change, this is probably the preferred method:

cd $NEMO git pull make rebuild

as it will preserve the possibly peculiar options for configure that you passed the first time it was installed.

8.3 Advanced Installation

It's a fact of life that you will not be satisified with the compiler or libraries that your system provides. Add to this that if you don't have admin privilages, and you might be in for a rude awakening.

No worries, NEMO has you covered (to some degree). We provide an environment (a poor man's container) where most open source libraries can be installed with a supported `$NEMO/opt` prefix. This means you can configure packages using

```
--with-prefix=$NEMO/opt
```

of for *cmake* based packages

```
-DCMAKE_INSTALL_PREFIX=$NEMO/opt
```

as NEMO generally adds the $NEMO/opt tree search for include and library files.

For some packages this has been automated using the `mknemo` command, described in the next section.

8.4 mknemo

Although the `mknemo` script was intended to quickly compile a NEMO program, without the need to know where the source code lives. It is also used to aid the installation of a number of supported libraries that can be used by NEMO. They are compiled within `$NEMO/local`, and will be installed in `$NEMO/opt`, as described in the previous section. The supporting scripts are generally located `$NEMO/src/scripts/mknemo.d` for you to examine.

Examples:

```
mknemo cfitsio fftw gsl hdf4 hdf5 hypre netcdf4 wcslib
```

The *Programmers Guide (*)* will give some advanced examples how to deal with other libraries, and writing your own programs or one of the plugins.

8.5 python

With so many useful python packages around, and so many different methods (anaconda, cond, venv etc.), we will not recommend a method, as this will likely depend on your own situation. The installation examples below should give you enough information how to adapt it for your python installation. It goes without saying (this is 2021) we only support python3.

However, if you install python from within NEMO, there will be a `$NEMO/anaconda3` directory, that gets automatically activated once NEMO is loaded. Here is how you can install that version:

```
cd $NEMO
make python
```

This will install a few python modules we often wind up using: **amuse-framework**, **amuse-galactics**, **amuse-gadget2**, **amuse-bhtree**, **astromartini**, **gala**, **galpy**, **pynbody**, **python-unsio**, **python-unsiotools**, and **yt**

For a number of these we have small test scripts to see if they are functional:

```
cd $NEMO/src/scripts/python
make tests
```

For the cases where you want some control and be in developer mode, we suggest the recommended practice of placing the code in `$NEMO/local`, as can be seen in the example below

```
cd $NEMO/local
git clone https://github.com/webbjj/clustertools
pip install -e clustertools
```

For a few packages, we have a few existing examples in the `$NEMO/usr` tree (e.g. amuse, martini, unsio and uns_projects)

CHAPTER

NINE

PROGRAMMERS GUIDE (*)

In this section an introduction is given how to write programs within the NEMO environment. It is based on an original report *NEMO: Elementary Mechanics Observatory* by Joshua Barnes (1986).

To the application programmer NEMO consists of a set of (C) macro definitions and object libraries, designed for numerical work in general and stellar dynamics in particular. A basic knowledge how to program in C, and use the local operating system to get the job done, is assumed. If you know how to work with *Makefile*'s, even better.

We start by looking at a typical *Hello Nemo* program:

```
#include <nemo.h>                          /* standard (NEMO) definitions */

string defv[] = {       /* standard keywords and default values and help */
    "n=10\n                Number of interations",           /* key1 */
    "VERSION=1.2\n         25-may-1992 PJT",                 /* key2 */
    NULL,               /* standard terminator of defv[] vector          */
};

string usage = "Example NEMO program 'hello'";        /* usage help text */

void nemo_main()                   /* standard start of any NEMO program */
{
    int n = getiparam("n");                          /* get n            */

    printf("Hello NEMO!\n");                         /* do some work ... */
    if (n < 0)                                       /* deal with fatal  */
       error("n=%d is now allowed, need >0",n);      /* errors           */
}
```

9.1 The NEMO Programming Environment

The modifications necessary to your UNIX environment in order to access NEMO are extensively described elsewhere. This not only applies to a user, but also to the application programmer.

In summary, the essential changes to your environment consist of one simple additions to your local shell startup file or you can do it interactively in your shell (be it sh or csh like).

```
   % source /opt/nemo/nemo_start.sh
or
   % source /opt/nemo/nemo_start.csh
```

where the location of NEMO=/opt/nemo is something that will likely be different for you.

9.2 The NEMO Macro Packages

We will describe a few of the most frequently used macro packages available to the programmer. They reside in the form of header include files in a directory tree starting at `$NEMOINC`. Your application code would need references like:

```
#include <nemo.h>              // this will include a few basic core NEMO include files
#include <snapshot/body.h>
#include <snapshot/get_snap.c>
```

Some of the macro packages are merely function prototypes, to facilitate modern C compilers (we strive to follow the C99 standard), and have associated object code in libraries in `$NEMOLIB` and programs need to be linked with the appropriate libraries (normally controlled via a Makefile).

9.2.1 stdinc.h

The macro package `stdinc.h` provides all basic definitions that ALL of NEMO's code must include as the first include file. It also replaces the often used {tt stdio.h} include file in C programs. The `stdinc.h` include file will provide us with a way to standardize on future expansions, and make code more machine/implementation independent (*e.g.* POSIX.1). In addition, it defines a more logical standard for C notation. For example, the normal C practice of using pointers to character for pointer to byte, or integer for bool, tends to encourage a degree of sloppy programming, which can be hard to understand at a later date.

A few of the basic definitions in this package:

- `NULL`: macro for 0, used to distinguish null characters and null pointers. This is often already defined by `stdio.h`.
- `bool`: typedef for `short int` or `char`, used to specify boolean data. See also next item. NOTE: the *curses* library also defines `bool`, and this made us change from `short int` to the more space saving `char`.
- `TRUE, FALSE`: macros for 1 and 0, respectively, following normal C conventions.
- `byte`: typedef for `unsigned char`, used to specify byte-sized data.
- `string`: typedef for `char *`, used to point to strings. Don`t use `string` for pointers you increment, decrement or explicitly follow (using *); such pointers are really `char *`.
- `real, realptr`: typedef for `float` or `double` (`float *` or `double *`, respectively), depending on the use of the `SINGLEPREC` flag. The default is `double`.
- `proc, iproc, rproc`: typedefs for pointers to procedures (void functions), integer-valued functions and real-valued functions respectively. % This always confusing issue will become clear later on.
- `local, permanent`: macros for {tt static}. Use {tt local} when declaring variables or functions within a file to be local to that file. They will not appear in the symbol table be usable as external symbols. Use {tt permanent} within a function, to retain their value upon subsequent re-entries in that function.
- `PI, TWO_PI, FOUR_PI, HALF_PI, FRTHRD_PI`:] macros for π, 2π, 4π, $4\pi/3$, respectively.
- `ABS(x), SGN(x)`: macros for absolute value and sign of `x`, irrespective of the type of `x`. Beware of side effects, it's a macro!
- `MAX(x,y), MIN(x,y)`: macros for the maximum and minimum of `x,y`, irrespective of the type of `x,y`. Again, beware of side effects.

- `stream`: typedef for `FILE *`. They are mostly used with the NEMO functions `stropen` and `strclose`, which are functionally similar to *fopen(3)* and *fclose(3)*, plus some added NEMO quircks like (named) pipes and the dot (.) `/dev/null` file.

9.2.2 getparam.h

The command line syntax is implemented by a small set of functions used by all conforming NEMO programs. A few function calls generally suffice to get the values of the input parameters. A number of more complex parsing routines are also available, to be discussed below.

First of all, a NEMO program must define which **program keywords** it will recognize. For this purpose it must define an array of *string*s with the names and the default values for the keywords, and optionally, but STRONGLY recommended, a one line help string for that keyword:

```
#include <nemo.h>        /*   every NEMO module needs this   */

string defv[] = {        /* definitions of the keywords */
    "in=???\n           Input file (a snapshot)",
    "n=10\n             Number of particles to view",
    "VERSION=1.1\n      14-jul-89 - 200th Bastille Day - PJT",
    NULL,
};

string usage = "example program";   /* def. of the usage line */
```

The *keyword=value* and *help* part of the string must be separated by a newline symbol (`\n`). If no newline is present, as was the case in NEMO's first release, no help string is available. ZENO uses a slightly different technique, where strings starting with `;` are the help string. This example in ZENO would look as follows:

```
string defv[] = {        "; example program",
    "in=???",            ";Input file (a snapshot)",
    "n=10",              ";Number of particles to view",
    "VERSION=1.1",       ";14-jul-89 - 200th Bastille Day - PJT",
    NULL,
};
```

You can see the first string is actually the same as NEMO's *usage* string. The current NEMO *getparam* package is able to parse both NEMO and ZENO style `defv[]` initializers.

The `help=h` command line option displays the *help* part of string during execution of the program for quick inline reference. The *usage* part defines a string that is used as a one line reminder what the program does. It's used by the various invocations of the user interface.

The first thing a NEMO program does, is comparing the command line arguments of the program (commonly called `string argv[]` in a C program) with this default vector of *keyword=value* strings (`string defv[]`), and replace appropriate reset values for later retrieval. This is done by calling `initparam` as the first step in your MAIN program:

```
main (int argc, string argv[])
{
    initparam(argv,defv);
    . . .
```

It also checks if keywords which do not have a default value (*i.e.* were given `???`) have really been given a proper value on the command line, if keywords are not specified twice, enters values of the system keywords etc.

There is a better alternative to define the main part of a NEMO program: by renaming the main entry point `main()` to `nemo_main()`, without any arguments, and calling the array of strings with default *key=val*'s `string defv[]`, the linker will automatically include the proper startup code (`initparam(argv,defv)`), the worker routine `nemo_main()` itself, and the stop code (`finiparam()`). The above section of code would then be replaced by a mere:

```
void nemo_main()
{
    . . .
```

This has the obvious advantage that various NEMO related administrative details are now hidden from the application programmers, and occur automatically. Remember that standard `main()` already shields the application programmer from a number of tedious setups (e.g. {it stdio} etc.). Within NEMO we have taken this one step further. A recent example that was added to `nemo_main` is the management of the number of processors in an OpenMP enhanced computing mode.

Once the user interface has been initialized, keyword values may be obtained at any point during execution of the program by calling `getparam()`, which returns a string

---

Note: ANSI rules say you can't write to this location in memory if they are direct references `string defv[]`; this is something that may well be fixed in a future release.

```
if (streq(getparam("n"),"0")
    printf(" You really mean zero or octal?\n");
```

There is a whole family of `getXparam()` functions which parse the string in a value of one of the basic C types `int, long, bool,` and `real`. It returns that value in that type:

```
#include <getparam.h>     //   included by <nemo.h>
. . .
int nbody;
. . .
if ( (nbody = getiparam("n")) <= 0) {
    printf("Cannot handle %d particles\n",nbody);
    exit(0);
}
```

Finally, there is a macro called `getargv0()`, which returns the name of the calling program, mostly used for identification:

```
if (getbparam("quit"))
    error("%s: early quit", getargv0());
```

This is very useful in library routines, who normally would not be able to know who called them. Actually, NEMO's `error` function already uses this technique, since it cannot know the name of the program by whom it was called. The `error` function prints a message, and exits the program.

More detailed information can also be found in the appropriate manual page: *getparam(3NEMO)* and *error(3NEMO)*.

9.2.3 Advanced User Interface and String Parsing

Here we describe *setparam* to add some interactive capabilities in a standard way to NEMO. Values of keywords should only be accessed and modified this way. Since keywords are initialized/stored within the source code, most compilers will store their values in a read-only part of data area in the executable image. Editing them may cause unpredictable behavior.

If a keyword string contains an array of items of the same type, one can use either `nemoinpX` or `getXrange`, depending if you know how many items to expect in the string. The `getXrange` routines will allocate a new array which will contain the items of the parsed string. If you do already have a declared array, and know that all items will fit in there, the `nemoinpX` routines will suffice.

An example of usage:

```
double *x = NULL;
double y[NYMAX];
int nxret, nyret;
int nxmax=0;

nyret = nemoinpd(getparam("y"), y, NYMAX);

nxret = getdrange(getparam("x"), &x, &nxmax);
```

Warning: May 24, 2021: The text below here in this chapter has not been latex->rst sanitized

In the first call the number of elements to be parsed from an input keyword {tt y=} is limited to {tt NYMAX}, and is useful when the number of elements is expected to be small or more or less known. The actual number of elements returned in the array {tt y[]} is {tt nyret}.

When the number of elements to be parsed is not known at all, or one needs complete freedom, the dynamic allocation feature of {tt getdrange} can be used. The pointer {tt x} is initialized to {tt NULL}, as well as the item counter {tt nxmax}. After calling {tt getdrange}, {tt x} will point to an array of length {tt nxmax}, in which the first {tt nxret} element contain the parsed values of the input keyword {tt x=}. Proper re-allocation will be done when a larger space is need on subsequent calls.

Both routines return negative error return codes, see {it nemoinp(3NEMO)}.

More complex parsing is also done by calling {tt burststring} first to break a string in pieces, followed by a variety of functions.

9.2.4 Alternatives to nemo_main

It is not required for your program to define with {tt nemo_main()}. There are cases where the user needs more control. An example of this is the {tt falcON}index{falcON} N-body code in {tt $NEMO/use/dehnen/falcON}. A header file (see e.g. {tt inc/main.h}) now defines main, instead of through the NEMO library:

```
// in main.h:

extern string defv[];
extern string usage;

namespace nbdy {
   char version [200];                              // to hold version info
```

(continues on next page)

(continued from previous page)

```
   extern void main();                            // to be defined by user
};

int main(int argc, char *argv[])                 // ::main()
{
   snprintf(nbdy::version,200,"VERSION=" /*..*/ ); // write version info
   initparam(argv,defv);                          // start NEMO
   nbdy::main();                                  // call nbdy::main()
   finiparam();                                   // finish NEMO
}
```

and the application includes this header file, and defines the keyword list in the usual way :

```
// in application.cc

#include <main.h>

string defv[] = { /*...*/, nbdy::version, NULL }; // use version info
string usage  = /*...*/ ;

void nbdy::main() { /*...*/ }                    // nbdy::main()
```

9.2.5 filestruct.h

The *filestruct* package provides a direct and consistent way of passing data between NEMO programs, much as *getparam* provides a way of passing (command line) arguments to programs. For reasons of economy and accuracy, much of the data manipulated by NEMO is stored on disk in binary form. Normally, data stored this way is completely unintelligible, except to specialized programs which create and access it. Furthermore, this approach to data handling tends to be very brittle: a trivial addition or alteration to the data stored in such a file can force the tedious and error-prone revision of many programs. To get around these problems and provide an explicit, flexible, and structured method of storing binary data, we developed a collection of general purpose routines to access binary data files.

From the programmers point of view, a structured binary file is a stream of tagged data objects. In XML parlor, you could view this as binary XML. These objects come in two classes. An *item* is a single instance or a regular array of one of the following C primitive types: `char`, `short`, `int`, `long`, `float` or `double`. A *set* is an unordered sequence of {it items} and {it sets}. This definition is recursive, so fully hierarchical file structures are allowed, and indeed encouraged. Every set or item has a name {it tag} index{tag, item} associated with it, used to label the contents of a file and to retrieve objects from a set. Data items have a {it type} index{type, item} and array {it dimension} index{dimension, item} attributed associated with them as well. This of course means that there is a little overhead, which may become too large if many small amounts of data are to be handled. For example, a snapshot with 128 bodies (created by {tt mkplummer}) with double precision masses and full 6 dimensional phase space coordinates totals 7425 bytes, whereas a straight dump of only the essential information would be 7168 bytes, a mere 3.5% overhead. After an integration, with 9 full snapshots stored and 65 snapshots with only diagnostics output, the overhead is much larger: 98944 bytes of data, of which only 64512 bytes are masses and phase space coordinates: the overhead is 53% (of which 29% though are the diagnostics output, such conservation of energy and angular momentum, cputime, center of mass, etc.).

The filestruct package uses ordinary {it stdio(3)} streams to access input and output files; index{stream} hence the first step in using filestruct is to open the file streams. For this job we use the NEMO library routine {tt stropen()}, index{stropen} which itself is not part of filestruct. {tt stropen({it name,mode})} is much like {tt fopen()} of {it stdio}, but slightly more clever; it will not open an existing file for output, unless the {it mode} string is {tt "w!"}. % append ??? An additional oddity to {tt stropen} is that it treats the dash filename {tt "-"}, index{-,filename} as standard in/output,footnote{Older versions of {it filestruct} cannot handle binary files in pipes, since filestruct uses fseek(3)}

and {tt "s"} as a scratch file. Since {it stdio} normally flushes all buffers on exit, it is often not necessary to explicitly close open streams, but if you do so, use the matching routine {tt strclose()}. Thisindex{strclose} also frees up the table entries on temporary memory used by the filestruct package. As in most applications/operating systems a task can have a limited set of open files associated with it. Scratchindex{scratch files} files are automatically deleted from disk when they are closed. index{scratch files}

Having opened the required streams, it is quite simple to use the basic data I/O routines. For example, suppose the following declarations have been made:

```
#include <stdinc.h>
#include <filestruct.h>

stream instr, outstr;
int    nbody;
string headline;

#define MAXNBODY 100
real    mass[MAXNBODY];
```

(note the use of the {tt stdinc.h} conventions). And now suppose that, after some computation, results have been stored in the first {tt nbody} components of the {tt mass} array, and a descriptive message has been placed in {tt headline}. The following piece of code will write the data to a structured file:

```
outstr = stropen("mass.dat", "w");

put_data(outstr, "Nbody", IntType, &nbody, 0);
put_data(outstr, "Mass", RealType, mass, nbody, 0);
put_string(outstr, "Headline", headline);

strclose(outstr);
```

Data (the 4th argument in {tt put_data}, is always passed by address, even if one element is written. This not only holds for reading, but also for writing, as is apparent from the above example. Note that no error checking is needed when the file is opened for writing. If the file {tt mass.dat} would already have existed, {tt error()} would index{error(3)} have been called inside {tt stropen()} and aborted the program. Names of tags are arbitrary, but we encourage you to use descriptive names, although an arbitrary maximum of 64 is enforced by chopping any incoming string.

The resulting contents of {tt mass.dat} can be viewed with the {tt tsf} utility: index{tsf}

```
% tsf mass.dat
int Nbody 010
double Mass[8] 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
char Headline[20] "All masses are equal"
```

Note the octal representation {tt 010=8} of {tt Nbody}. **OLD**

% CANNOT DO THIS YET - CONCEPT NOT INTRODUCED HERE % Also note % that it is not {bf required} to use sets, subsets, subsubsets etc., % but such encapsulation has certain advantages.

It is now trivial to read data from this file:

```
instr = stropen("mass.dat", "r");

get_data(instr, "Nbody", IntType, &nbody, 0);
get_data(instr, "Mass", RealType, mass, nbody, 0);
headline = get_string(instr, "Headline");
```

(continues on next page)

(continued from previous page)

```
strclose(instr);
```

Note that we read the data in the same order as they were written.

During input, the filestruct routines normally perform strict type-checking; the tag, type and dimension supplied to {tt get_data()} must match the attributes of the data item, written previously, exactly. Such strict checking helps prevent many common errors in using binary data. Alternatively, you can use {tt get_data_coerced()}, which is called exactly like {tt get_data()}, but interconverts {tt float} and {tt double} valuesfootnote{The implementors of NEMO will not be held responsible for any loss of precision resulting from the use of this feature.}.index{real}

To provide more flexibility in programming I/O, a series of related items may be hierarchically wrapped into a set:

```
outstr = stropen("mass.dat", "w");

put_set(outstr, "NotASnapShot");
   put_data(outstr, "Nbody", IntType, &nbody, 0);
   put_data(outstr, "Mass", RealType, mass, nbody, 0);
   put_string(outstr, "Headline", headline);
put_tes(outstr, "NotASnapShot");

strclose(outstr);
```

Note that each {tt put_set()}index{put_set} must be matched by an equivalent {tt put_tes()}index{put_tes}. For input, corresponding routines {tt get_set()} and {tt get_tes()} are used. These also introduce a significant additional functionality: between a {tt get_set()}index{get_set} and {tt get_tes()}index{get_tes}, the data items of the set may be read in any orderfootnote{tagnames must now be unique within an item, as in a C {tt struct}}, or not even read at all. For example, the following is also a legal way to access the {tt NotASnapShot}footnote{This is a toy model, shown for its simplicity. The full {tt SnapShot} format is discussed in Section ref{ss:snapshot}}:

```
instr = stropen("mass.dat", "r");

if (!get_tag_ok(instr,"NotASnapShot"))
    error("File mass.dat is not a NotASnapShot\n");

get_set(instr,"NotASnapShot");
headline = get_string(instr, "Headline");
get_tes(instr,"NotASnapShot");

strclose(instr);
```

This method of *filtering* a data input stream clearly opens up many ways of developing general-purpose programs. Also note that the {tt bool} routine {tt get_tag_ok()} can be used to control the flow of the program, as {tt get_set()} would call {tt error()} when the wrong tag-name would be encountered, and abort the program.

The UNIX program `cat` can also be used to catenate multiple binary data-sets into one, *i.e.*

```
%   cat mass1.dat mass2.dat mass3.dat > mass.dat
```

The `get_tag_ok` routine can be used to handle such multi-set data files. The following example shows how loop through such a combined data-file.

```
instr = stropen("mass.dat", "r");
```

(continues on next page)

(continued from previous page)

```
while (get_tag_ok(instr, "NotASnapShot") {
   get_set(instr, "NotASnapShot");
   get_data(instr, "Nbody", IntType, &nbody, 0);
   if (nbody > MAXNBODY) {
        warning("Skipping data with too many (%d) items",nbody);
        get_tes(instr,"NotASnapShot");
        continue;
   }
   get_data(instr, "Mass", RealType, mass, nbody, 0);
   headline = get_string(instr, "Headline");
   get_tes(instr,"NotASnapShot");
   /*   process data   */
}

strclose(instr);
```

The loop is terminated at either end-of-file, or if the next object in {tt instr} is not a {tt NotASnapShot}.

It is easy to the skip for an item if you know if it is there:

```
while(get_tag_ok(instr,"NotASnapShot"))     /*  ??????? */
    skip_item(instr,"NotASnapShot");
```

The routine {tt skip_item()} is only effective, or for that matter required, when doing input at the top level, {it i.e.} not between a {tt get_set()} and matching {tt get_tes()}, since I/O at deeper levels is random w.r.t. items and sets. In other words, at the top level I/O is sequential, at lower levels random.

A relative new feature in data access is the ability to do random and blocked access to the data. Instead of using a single call to {tt get_data} and {tt put_data}, the access can be sequentially blocked using {tt get_data_blocked} and {tt put_data_blocked}, provided it is wrapped using {tt get_data_set} and {tt get_data_tes}, for example:

```
get_data_set    (instr, "Mass", RealType, nbody, 0);
real *mass = (real *) allocate((nbody/2)*sizeof(real));
get_data_blocked(instr, "Mass", mass, nbody/2);
get_data_blocked(instr, "Mass", mass, nbody/2);
get_data_tes    (instr, "Mass");
```

would read in the {tt Mass} data in two pieces into a smaller sized {tt mass} array. A similar mode exists to randomly access data with an item. A current limitation of this mode is that such access is only allowed on one item at a time. In this mode an item must be closed before the next one can be opened in such a mode.index{blocked I/O}index{filestruct,blocked I/O} index{filestruct,random I/O}index{random access}

9.2.6 vectmath.h

The {tt vectmath.h} macro package index{vectmath.h} provides a set of macros to handle some elementary operations on two, three or general N dimensional vectors and matrices. The dimension N can be picked by providing the package with a value for the preprocessor macro {bf NDIM}. If this is not supplied, the presence of macros {bf TWODIM} and {bf THREEDIM} will be checked, in which case {bf NDIM} is set to 2 or 3 respectively. The default of {bf NDIM} when all of the above are absent, is 3. Of course, the macro {bf NDIM} must be provided before {tt vectmath.h} is included to have any effect. Resetting the value of {bf NDIM} after that, if your compiler would allow it anyhow without an explicit {tt #undef}, may produce unpredictable results.

There are also a few of the macro's which can be used as a regular C function, returning a real value, {it e.g.} {tt absv()} for the length of a vector.

Operations such as {bf SETV} (copying a vector) are properly defined for every dimension, but {bf CROSSVP} (a vector cross product) has a different meaning in 2 and 3 dimensions, and is absent in higher dimensions.

It should be noted that the matrices used here are true C matrices, a pointer to an array of pointers (to 1D arrays), unlike FORTRAN arrays, which only occupy a solid 2D block of memory. C arrays take slightly more memory. For an example how to make C arrays and FORTRAN arrays work closely together see e.g. {it Numerical Recipes in C} by {it Press et al.} (MIT Press, 1988). index{Numerical Recipes} index{Press W.}

In the following example a 4 dimensional vector is cleared:

```
#define NDIM 4
#include <vectmath.h>

nemo_main()
{
    vector a;           /* same as:   double a[4]  */

    CLRV(a);
}
```

{it some more examples here - taken from some snap code}

9.2.7 snapshots: get_snap.c and put_snap.c

mylabel{ss:snapshot}

These routines exemplify an attempt to provide truly generic I/O of N-body data. They read and write structured binary data files conforming to the overall form seen in earlier sections. Internally they operate on {tt Body} structures; index{Body, structure} A {tt Body} has components accessed by macros such as {tt Mass} for the mass, {tt Pos} and {tt Vel} for the position and velocity vectors, {it etc.}. Since {tt get_snap.c} and {tt put_snap.c} use only these macros to declare and access bodies, they can be used with any suitable {tt Body} structure. They are thus provided as C source code to be included in the compilation of a program. Definitions concerning Body's and snapshots are obtained by including the files {tt snapshot/body.h} and {tt snaphot/snapshot.h}. index{snapshot.h}index{body.h}

A program which should handle a large number of particles, may decide to include a more simple {tt Body} structure, as is {it e.g.} provided by the {tt snapshot/barebody.h} macro file. This body only includes the masses and phase space coordinates, which would only occupy 28 bytes per particle (in single precision), as opposed to the 100/104 bytes per particle for a double precision {tt Body} from the standard {tt snapshot/body.h} macro file. This last one contains {tt Mass, PhaseSpace, Phi, Acc, Aux} and {tt Key}.

In the example listed under Table~ref{src:snapfirst} the first snapshot of an input file is copied to an output file.

begin{table}[tb] caption[source: snapfirst.c]{$NEMO/src/tutor/snap/snapfirst.c} mysrcfile{snapfirst.src} mylabel{src:snapfirst} small verbatimlisting{snapfirst.src} normalsize end{table}

```
#include <stdinc.h>         /* general I/O */
#include <getparam.h>       /* for the user interface */
#include <vectmath.h>       /* to define NDIM */
#include <filestruct.h>

#include <snapshot/snapshot.h>  /* Snapshot macros */
#include <snapshot/body.h>
#include <snapshot/get_snap.c>  /*   and I/O routines */
#include <snapshot/put_snap.c>

string defv[] = {
```

(continues on next page)

(continued from previous page)

```
    "in=???\n        Input snapshot",
    "out=???\n       Output snapshot",
    "VERSION=0.0\n  21-jul-93 PJT",
    NULL,
};

string usage="copy the first snapshot";

nemo_main()
{
   Body *btab = NULL;   /* pointer to the whole snapshot */
   int  nbody, bits;
   real tsnap;
   stream instr, outstr;

   instr = stropen(getparam("in"),"r");
   outstr = stropen(getparam("out"),"w");
   get_history(instr);
   get_snap(instr, &btab, &nbody, &tsnap, &bits);
   put_history(outstr);
   put_snap(outstr,&btab, &nbody, &tsnap, &bits);
   strclose(instr);
   strclose(outstr);
}
```

Notice that the first argument of {tt stropen()}, the filename, is directly obtained from the user interface. The input file is opened for reading, and the output file for writing. Some historyfootnote{See next section for more details on history processing} is obtained from the input file (would we not have done this, and the input file would have contained history, a subsequent {tt get_snap()} call would have failed to find the snapshot), and the first snapshot is read into an array of bodies, pointed to by {tt btab}. Then the output file has the old history written to it (although any command line arguments were added to that), followed by that first snapshot. Both files are formally closed before the program then returns.

9.2.8 history.h

When performing high-level data I/O, as is offered by a package such as {tt get_snap.c} and {tt put_snap.c}, there is an automated way to keep track of data history. index{history.h}

When a NEMO program is invoked, the program name and command line arguments are saved by the {tt initparam()} in a special history database. Most NEMO programs will write such history items to their data-file(s) before the actual data. Whenever a data-file is then opened for reading, the programmer should first read these data-history items. Conversely, when writing data, the history should be written first. In case of the {tt get/put_snap} package:

```
get_history(instr);
get_snap(instr, &btab, &nbody, &time, &bits);
   /*     process data     */
put_history(outstr);
put_snap(outstr,&btab, &nbody, &time, &bits);
```

index{get_history} index{put_history} Private comments should be added with the {tt app_history()} index{app_history}index{add_history}index{readline, GNU} footnote{The old name, {tt add_history} was already used by the GNU {it readline} library} When a series of snapshot is to be processed, it is recommended that the program should only be output the history once, before the first output of the snapshot, as in the following example:

```
get_history(instr);
put_history(outstr);
for(;;) {
    get_history(instr);     /* defensive but in-active */
    get_snap(instr, &btab, &nbody, &time, &bits);
        /*     process data  and decide when done */
    put_snap(outstr,&btab, &nbody, &time, &bits);
}
```

Note that the second call to {tt get_history()}, within the for-loop, is really in-active. If there happen to be history items sandwiched between snapshots, they will be read and added to the history stack, but not written to the output file, since {tt put_history()} was only called before the for-loop. It is only a defensive call: {tt get_snap()} would fail since it expects only pure {tt SnapShot} sets (in effect, it calls {tt get_set(instr,"SnapShot")} first, and would call {tt error()} if no snapshot encountered).

9.3 Building NEMO programs

mylabel{s:example}

Besides writing the actual code for a program, an application programmer has to take care of a few more items before the software can be added and formally be accepted to NEMO. This concerns writingindex{compiling, NEMO programs}index{linking, NEMO programs} the documentation and possibly a Makefile, the former one preferably in the form of standard UNIX manual pages ({it man(5)}). We have templates for both Makefile's and manual pages. Both these are discussed in detail in the next subsections.

Because NEMO is a development package within which a multitude of people are donating software and libraries, linking a program can become cumbersome. In the most simple case however (no graphics or mathematical libraries needed), only the main NEMO library is needed, and the following command should suffice to produce an executable:

```
% cc -g -o snapprint snapprint $NEMOLIB/libnemo.a -lm
```

For graphics programs a solution would be to use the **YAPPLIB** environment variable.

Warning: YAPPLIB was deprecated a while back

An example of the compilation of a graphics program:

```
% cc -g -o snapplot snapplot.c -lnemo $YAPPLIB -lm
```

Each user is given a subdirectory in {tt $NEMO/usr}, under which code may be donated which can be compiled into the running version of NEMO. Stable code, which has been sufficiently tested and verified, can be placed in one of the appropriate {tt $NEMO/src} directories. For proper inclusion of user contributed software a few rules in the {tt Makefile} have to be adhered to.

The {tt bake}index{bake, make script} and {tt mknemo}index{mknemo, script} script should handle compilation and installation of most of the standard NEMO cases. Some programs, like the N-body integrators, are almost like complicated packages themselves, and require their own Makefile or install script. For most programs you can compile it by:

```
% bake snapprint
```

or to install:footnote{this assumes you have some appropriate NEMO permissions}

```
% mknemo snapprint
```

9.3.1 Manual pages

It is very important to keep a manual file (preferably in the UNIX man format) online for every program. A program that does not have an accompanying manual page is not complete. Of course there is always the inline help `help=`) that every NEMO program has. We have a script `checkpars.py` that will check if the parameters listed in `help=` are the same (and in the same order) as the ones listed in the MAN page.

To a lesser degree this also applies to the public libraries. A template roff sample can be found in `example.8`. We encourage authors to have a MINIMUM set of sections in a man-page as listed below. The ones with a '*' are considered somewhat less important:

- item[{bf NAME}] the name of the beast.
- item[{bf SYNOPSIS}] command line format or function prototype, include files needed etc.
- item[{bf DESCRIPTION}] maybe a few lines of what it does, or not does.
- item[{bf PARAMETERS}] description of parameters, their meaning and default values. This usually applies to programs only.
- **item[{bf EXAMPLES}]** (*) in case non-trivial, but recommended anyhow
- **item[{bf DEBUG}]** (*) at what {tt debug} levels what output appears.
- **item[{bf SEE ALSO}]** (*) references to similar functions, more info
- **item[{bf BUGS}]** (*) one prefers not to have this of course
- **item[{bf TIMING}]** (*) performance, dependence on parameters if non-trivial
- **item[{bf STORAGE}]**

  (*) storage requirements - mostly of importance when programs allocate memory dynamically, or when applicable for the programmer.
- **item[{bf LIMITATIONS}]** (*) does it have any obvious limitations?
- **item[{bf AUTHOR}]** who wrote it (a little credit is in its place) and/or who is responsible.
- **item[{bf FILES}]** (*) in case non-trivial
- item[{bf HISTORY}] date, version numbers, why updated, by whom (when created)

9.3.2 Makefiles

Makefiles are scripts in which "the rules are defined to make targets", see *make(1)* for many more details. In other words, the Makefile tells how to compile and link libraries and programs. NEMO uses Makefiles extensively for installation, updates and various other system utilities. Sometimes scripts are also available to perform tasks that can be done by a Makefile.

There are several types of Makefiles in NEMO:

- item{1.} The first (top) level Makefile. It lives in NEMO's root directory (normally referred to as {tt $NEMO}) and can steer installation on any of a number of selected machines, it includes some import and export facilities (tar/shar) and various other system maintenance utilities. At installation it makes sure all directories are present, does some other initialization, and then calls Makefile's one level down to do the rest of the dirty work. The top level Makefile is not of direct concern to an application programmer, nor should it be modified without consent of the NEMO system manager.

- item{2.} Second level Makefiles, currently in {tt $NEMO/src} and {tt $NEMO/usr}, steer the building of libraries and programs by calling Makefiles in subdirectories one more level down. Both this 2nd level Makefile and the one described earlier are solely the responsibility of NEMO system manager. You don't have to be concerned with them, except to know of their existence because your top level Makefile(s) must be callable by one of the second level Makefiles. This interface will be described next.
- item{3.} Third level Makefiles live in source or user directories {tt $NEMO/src/topic} and {tt $NEMO/usr/name} (and possibly below). They steer the installation of user specific programs and libraries, they may update NEMO libraries too. The user writes his own Makefile, he usually splits up his directory in one or more subdirectories, where the real work is done by what we could then call level 4 or even level 5 Makefiles. However, this is completely the freedom of a user. The level 3 Makefiles normally have two kinds of entry points (or 'targets'): the user 'install' targets are used by the user, and make sure this his sources, binaries, libraries, include files etc. are copied to the proper places under {tt $NEMO}. The second kind of entry point are the 'nemo' targets and never called by you, the user; they are only called by Makefiles one directory level up from within {tt $NEMO} below during the rebuilding process of NEMO, {it i.e.} a user never calls a nemo target, NEMO will do this during its installation. Currently we have NEMO install itself in two phases, resulting in two 'nemo' targets: 'nemo_lib' (phase 1) and 'nemo_bin' (phase 2). A third 'nemo' target must be present to create a lookup table of directories and targets for system maintenance. This target must be called 'nemo_src', and must also call lower level Makefiles if applicable.
- **Testfile** : this is a Makefile for testing (the `make test` command will use it)
- **Benchfile** : this is a Makefile for benchmarks. Under development.

This means that user Makefiles {bf MUST} have at least these three targets in order to rebuild itself from scratch. In case a user decides to split up his directories, the Makefiles must also visit each of those directories and make calls through the same entry points 'nemo_lib' and 'nemo_bin', 'nemo_src'; a sort of hierarchical install process.

For more details see the template Makefiles in NEMO's sec subdirectories and the example below in section~ref{ss-example}.

We expect a more general install mechanism with a few more strict rules for writing Makefiles, in some next release of NEMO.

9.3.3 An example NEMO program

mylabel{ss-example} Under Table~ref{src:hello} below you can find a listing of a very minimal NEMO program, ``{tt hello.c}":index{nemo_main}index{hello.c}

begin{table}[tb] caption[source: hello.c]{$NEMO/src/tutor/hello/hello.c} mylabel{src:hello} mysrcfile{hello.src} small verbatimlisting{hello.src} normalsize end{table}

```
#include <nemo.h>                          /* standard (NEMO) definitions */

string defv[] = {       /* standard keywords and default values and help */
    "n=10\n                 Number of interations",           /* key1 */
    "VERSION=1.2\n          25-may-1992 PJT",                 /* key2 */
    NULL,               /* standard terminator of defv[] vector          */
};

string usage = "Example NEMO program 'hello'";        /* usage help text */

void nemo_main()                    /* standard start of any NEMO program */
{
    int n = getiparam("n");                           /* get n            */

    printf("Hello NEMO!\n");                          /* do some work ... */
```

```
    if (n < 0)                                      /* deal with fatal  */
       error("n=%d is now allowed, need >0",n);     /* errors           */
}
```

and a corresponding example Makefile to install by {em user} and {em nemo} could look like the one shown under Table~ref{src:makefile}

Note that for this simple example the {tt Makefile} actually larger than the source code, {tt hello.c}, itself. Fortunately not every programs needs their own Makefile, in fact most programs can be compiled with a default rule, via the {tt bake}index{bake, make script} script. This generic makefile is used by the {tt bake} command, and is normally installed in {tt $NEMOLIB/Makefile}, but check out your {tt bake} command or alias.

% newpage % centerline{bf Example Makefile}

begin{table}[htb] caption[source: makefile]{Sample makefile - cf. $NEMOLIB/Makefile} mylabel{src:makefile} mysrcfile{makefile.src} footnotesize verbatimlisting{makefile.src} normalsize end{table}

```
#       template Makefile to install NEMO binaries and libraries....
#       Usually installed as $NEMOLIB/Makefile and use by the 'bake' replace
#       ment of 'make'

CFLAGS = -g
FFLAGS = -g -C -u

#
L = $(NEMOLIB)/libnemo.a
OBJFILES=
BINFILES=
TESTFILES=
#       Define an extra SUFFIX for our .doc file
.SUFFIXES: .doc

.c.doc: $*
        $* help=t > $*.doc
        @echo "### Normally this $*.doc file would be moved to NEMODOC"
        @echo "### You can also use mkpdoc to move it over"

help:
        @echo "Standard template nemo Makefile"
        @echo " No more help to this date"

clean:
        rm -f core *.o *.a *.doc $(BINFILES) $(TESTFILES)

cleanlib:
        ar dv $(L) $(OBJFILES)
        ranlib $(L)

$(L):   $(LOBJFILES)
        echo "*** Now updating all members ***"
        ar ruv $(L) $?
        $(RANLIB) $(L)
        rm -f $?
```

```
lib:   $(L)

bin: $(BINFILES)

#       NEMO compile rules
.o.a:
        @echo "***Skipping ar for $* at this stage"

.c.o:
        @echo "***Compiling $*"
        $(CC) $(CFLAGS) -c $<

.c.a:
        @echo "***Compiling $* for library $(L)"
        $(CC) $(CFLAGS) -c $<

.c:
        @echo "***Compiling and linking $*"
        $(CC) $(CFLAGS) -o $* $*.c $(BL) $(L) $(AL) -lm

.o:
        @echo "***Compiling and linking $*"
        $(CC) $(CFLAGS) -o $* $*.o $(BL) $(L) $(AL) -lm

# any non-standard targets follow here
```

{bf Warning:}The structure of this so-called 'standard' NEMO Makefiles is still under debate, and will probably drastically change in some future release. Best is to check some local Makefiles. A possible candidate is the GNUindex{make, GNU} makeindex{gmake, GNU} facility.

9.4 Extending NEMO environment

Let us now summarize the steps to follow to add and/or create new software to NEMO. The examples below are suggested steps taken from adding Aarseth's {tt nbody0} program to NEMO, and we assume him to have his original stuff in a directory {tt ~/nbody0}.

- **item[1:] Create a new directory, {tt "cd $NEMO/usr ; mkdir aarseth"}** and inform the system manager of NEMO that a new user should be added to the user list in {tt $NEMO/usr/Makefile}. You can also do it yourself if the file is writable by you.
- **item[2:] Create working subdirectories in your new user directory,** {tt "cd aarseth ; mkdir nbody0"}.
- **item[3:] Copy a third level Makefile from someone else, and substitute** the subdirectory names to be installed for you, i.e. your new working subdirectories ({tt 'nbody0'} in this case): {tt "cp ../pjt/Makefile . ; emacs Makefile"}.
- **item[4:] Go 'home' and install, {tt "cd ~/nbody0 ; make install"}, assuming** the Makefile there has the proper install targets. Check the target Makefile in the directory {tt $NEMO/usr/aarseth/nbody0} what this last command must have done.

Actually, only step 1 is required. If a user cannot or does not want to confirm to the level 3/4 separation, he may do so, as long as the Makefile in level 3 (e.g. {tt $NEMO/usr/aarseth/Makefile}) contains the nemo_lib, nemo_bin and nemo_src install targets. An example of adding a foreign package that way is the {tt GRAVSIM} package index{GRAVSIM},

which has it's own internal structure. In the directory tree starting at {tt $NEMO/usr/mbellon/gravsim} an example of a different approach is given. Sometimes public domain packages have been added to NEMO, and its Makefiles have been adapted slightly to the NEMO install procedure.

9.5 Programming in C++

Most relevant header files from the NEMO C libraries have been made entrant for C++. This means that all routines should be available through:

```
extern "C"  {
    .....
}
```

The only requirement is of course that the {tt main()} be in C++. For this you have to link with the NEMO++ library {bf before} the regular NEMO library. So, assuming your header (-I) and library (-L) include flags have been setup, you should be able to compile your C++ programs as follows:

```
% g++ -o test test.cc -L$NEMOLIB -lnemo++ -lnemo -lm
```

9.6 Programming in FORTRAN

Programming in FORTRAN can also be done, but since NEMO is written in C and there is no 'standard' way to link FORTRAN and C code, such a description is always bound to be system dependent (large differences exist between UNIX, VMS, MSDOS, and UNICOS is somewhat of a peculiar case). Even within a UNIX environment there are a number of ways how the industry has solved this problem (cf. Alliant). Most comments that will follow, apply to the BSD convention of binding FORTRAN and C.

In whatever language you program, we do suggest that the startup of the program is done in C, preferably through index{nemo_main} the {tt nemo_main()} function (see Section~ref{ss-example}). As long as file I/O is avoided in the FORTRAN routines, character and boolean variables are avoided in arguments of C callable FORTRAN functions, all is relatively simple. Some care is also needed for multidimensional arrays which are not fully utilized. The only thing needed are C names of the FORTRAN routines to be called from C. This can be handled automatically by a macro package.

Current examples can be found in the programs {tt nbody0} and {tt nbody2}. In both cases data file I/O is done in C in NEMO's {it snapshot(5NEMO)} format, but the CPU is used in the FORTRAN code.

Examples of proposals for other FORTRAN interfaces can be found in the directory {tt $NEMOINC/fortran}.

Again this remark: the {it potential(5NEMO)} assumes for now a BSD type f2c interface, because character variables are passed. This has not been updated yet. You would have to provide your own f2c interface to use FORTRAN potential routines on other systems.

Simple FORTRAN interface workers within the snapshot interface are available in a routine {tt snapwork(n,m,pos,vel,…)}.

9.6.1 Calling NEMO C routines from FORTRAN

The NEMO user interface, with limited capabilities, is also available to FORTRANindex{fortran, calling C} programmers. First of all, the keywords, their defaults and a help string must be made available (see Section~ref{ss:getparam}). This can be done by supplying them as comments in the FORTRAN source code, as is show in the following example listed under Table~ref{src:testf2c}

begin{table}[htb] caption[testf2c.f]{$NEMO/src/kernel/fortran/test.f} mylabel{src:testf2c} mysrcfile{testf2c.src} small verbatimlisting{testf2c.src} normalsize end{table}

```
C
C       Test program for NEMO's footran interface
C               25-jun-91  1.0
C               24-may-92  1.1
C               21-jul-93  1.2 for manual src file
C       Note the special comments C: C+ C- for 'ftoc'
C:      Test program for NEMO's footran interface
C+
C   in=???\n            Required (dummy) filename
C   n=1000\n            Test integer value
C   pi=3.1415\n         Test real value
C   e=2.3\n             Another test value
C   text=hello world\n  Test string
C   VERSION=1.1\n       24-may-92 PJT
C-
C
      SUBROUTINE nemomain         ! note the name !
C
C#include "getparam.inc"          ! if cpp is used to get at $NEMOINC
      INCLUDE 'getparam.inc'      ! use defs from $NEMOINC

      INTEGER n
      DOUBLE PRECISION pi,e
      CHARACTER text*40, file*80

      file = getparam('in')       ! get the CL parameters
      n = getiparam('n')
      pi = getdparam('pi')
      e = getdparam('e')
      text = getparam('text')

      WRITE (*,*) 'n=',n,' pi=',pi,' e=',e,' text='//text

      END
```

The documentation section between {tt C+} and {tt C-} can be extracted with a NEMO utility, {tt ftoc}, to the appropriate C module as follows:

```
% ftoc test.f test_main.c
```

after which the new {tt test_main.c} file merely has to be included on the commandline during compilation. To avoid having to include FORTRAN libraries explicitly on the commandline, easiest is to use the {tt f77} command, instead of {tt cc}:

```
% g77 -o test test.f test_main.c -I$NEMOINC -L$NEMOLIB -lnemo
```

This only works if your operating supports mixing C and FORTRAN source code on one commandline. Otherwise try:

```
% gcc -c test_main.c
% g77 -o test test.f test_main.o -L$NEMOLIB -lnemo
```

where the NEMO library is still needed to resolve the user interface of course.

9.6.2 Calling FORTRAN routines from NEMO C

No official support is needed, although for portablility it would be nice to include a header file that maps the symbol names and such.

9.7 Debugging

Apart from the usual debugging methods that everybody knows about, NEMO programs usually have the following additional properties which can cut down in debugging time. If not conclusive during runtime, you can either decide to compile the program with debugging flags turned on, and run the program through the debugger, or add more {tt dprintf}index{dprintf(3)} or {tt error}index{error(3)} function calls:

- During runtime you can set the value for the {tt debug=}index{debug, system keyword} (or use the equivalent {tt DEBUG} environment variable) system keyword to increase the amount of output. Note that only levels 0 (the default) through 9 are supported. 9 should produce a lot of output.
- During runtime you can set the value for the {tt error=}index{error, system keyword} (or use the equivalent {tt ERROR} environment variable) system keyword to bypass a number of fatal error messages that you know are not important. For example, to overwrite an existing file you would need to increase {tt error} by 1.

CHAPTER

TEN

POTENTIALS AND ACCELLERATIONS (*)

Here we lists a number of potentials, taken from from CTEX comments in the **$NEMO/src/orbit/potential/data** source code directory. Most NEMO programs that deal with potentials have three program keywords associated with potentials:

- **potname=** describes the name
- **potpars=** optional parameters
- **potfile=** optional associated filenames (or other textual information)

Each section below details a potential and explains the usage of the **potpars=** and **potfile=** keywords. The section title is the actual **potname=** to be used for this potential. Mostly **G=1**, unless otherwise mentioned.

Todo: describe the newer **accelerations** from *falcON*

Todo: potentials auto-build from source code ???

10.1 Example Potentials

In the latex manual this chapter is derived from the code. We thus need a new `crst` script that produces this. Needs to handled embedded math, such as $\frac{\sum_{t=0}^{N}f(t,k)}{N}$

10.1.1 bar83

% File: bar83.c

potname=bar83 potpars={it $Omega,f_m,f_x,{cover a}$}}

Barred potential as described by Teuben and Sanders (1983), see also {bf teusan83}.

Note: the potential only valid in the z=0 plane!

10.1.2 bulge1

% File: bulge1.c

{bf potname=bulge1 potpars={it $Omega,M,R,c/a$}}

homogeneous oblate bulge with mass M, radius R, and axis ratio c/a

10.1.3 ccd

% File: ccd.c

{bf potname=ccd potpars={it $Omega,Iscale,Xcen,Ycen,Dx,Dy$} potfile={it image(5NEMO)}}

This potential is defined using a simple cartesian grid on which the potential values are stored. Using bilinear interpolation the values and derivatives are computed at any point inside the grid. Outside the grid (as defined by the WCS in the header) the potential is not defined and assumed 0. The lower left pixel of an image in NEMO is defined as (0,0), with WCS values Xmin,Ymin derived from the header. If the (Xcen,Ycen) parameters are used, these are the 0-based pixel coordinates of the center pixel. If (Dx,Dy) are used, these are the pixel separations. To aid astronomical images where $Dx < 0$, these are interpreted as positive. Also note that potentials are generally negative, so it is not uncommon to need $Iscale = -1$. Programs such as {it potccd} can create such a {bf ccd} grid potential from a regular potential.

Note: Since these forces are defined only in the Z=0 plane, the Z-forces are always returned as 0.

10.1.4 cp80

% File: cp80.c

{bf potname=cp80 potpars={it $Omega,epsilon$}}

Contopoulos & Papayannopoulos (1980, A&A, 92,33) used this potential in the study of orbits in barred galaxies. Note that their *bar* is oriented along the Y-axis, an axis ratio is not well defined, and for larger values of $epsilon$ the density can be negative. The potential used is given by adding an axisymmetric component to a m=2 fourier component:

$$\Phi = \Phi_1 + \Phi_2$$

where Φ_1 is the Isochrone potential with unit scalelength and mass, and Φ_2 the Barbanis & Woltjer (1965) potential:

$$\Phi_1 = -\frac{1}{(1+\sqrt{1+r^2})}$$

and

$$\Phi_2 = \epsilon r(16-r)cos(2\phi)$$

A value of $\epsilon = 0.00001$ is the default for a moderate bar, whereas 0.001 is a strong bar!

10.1.5 dehnen

% File: dehnen.c

{bf potname=dehnen potpars={it $Omega,M,a,gamma$}}

Walter Dehnen (1993, MN {bf 265}, 250-256) introduced a family of potential-density pairs for spherical systems:

The potential is given by:

$$\Phi = \frac{GM}{a} \frac{1}{2-\gamma} \left[1 - \left(\frac{r}{r+a} \right)^{2-\gamma} \right]$$

cumulative mass by

$$M(r) = M \frac{r}{(r+a)^{3-\gamma}}$$

and density by

$$\rho = \frac{(3-\gamma)M}{4\pi} \frac{a}{r^{\gamma}(r+a)^{4-\gamma}}$$

with \$0 <= gamma < 3\$. Special cases are the Hernquist potential (\$gamma=1\$), and the Jaffe model (\$gamma=2\$). The model with \$gamma=3/2\$ seems to give the best comparison withe de Vaucouleurs \$R^{1/4}\$ law.

See also Tremaine et al. (1994, AJ, 107, 634) in which they describe the same density models with \$eta=3-gamma\$ and call them \$eta\$-models.

10.1.6 dublinz

% File: dublinz.c

{bf potname=dublinz potpars={it \$Omega,r_0,r_1,v_1,dvdr,s,h\$}}

Forces defined by a double linear rotation curve defined by (\$r_1,v_1\$) and a gradient \$dvdr\$ between \$r_0\$ and \$r_1\$. As in {bf flatz} (from which this one is derived), the potential is quasi harmonic in \$Z\$ (linear forces), with radial scalelength \$h\$ and scale height \$s\$

10.1.7 expdisk

% File: expdisk.c

{bf potname=expdisk potpars={it \$Omega,M,a\$}}

Exponential disk (BT, pp.77)

$$\Phi = -\frac{M}{r_d} x \left[I_0(x) K_1(x) - I_1(x) K_0(x) \right]$$

10.1.8 flatz

% File: flatz.c

potname=flatz potpars=:math:*Omega,r_0,v_0,s,h*

forces defined by a rotation curve that is linear to \$(r_0,v_0)\$ and flat thereafter and quasi harmonic in \$Z\$, with radial scalelength \$h\$ and scale height \$s\$. See also {bf dublinz} for a variation on this theme.

10.1.9 halo

% File: halo.c

{bf potname=halo potpars={it $Omega,v_0,r_c$}}

10.1.10 hh64

% File: hh64.c

potname=hh64 potpars=:math:*Omega,lambda*

$$\Phi = \frac{1}{2}(x^2 + x^2) + \lambda(x^2 y - \frac{1}{3}y^3)$$

10.1.11 grow_plum

% File: grow_plum.c

10.1.12 grow_plum2

% File: grow_plum2.c

10.1.13 harmonic

% File: harmonic.c

{bf potname=harmonic potpars={it $Omega,omega_x^2,omega_z^2,omega_z^2$}}

Harmonic potential

$$\Phi = \frac{1}{2}\omega_x^2 x^2 + \frac{1}{2}\omega_y^2 y^2 + \frac{1}{2}\omega_z^2 z^2$$

10.1.14 hernquist

% File: hernquist.c

{bf potname=hernquist potpars={it $Omega,M,r_c$}}

The Hernquist potential (ApJ, 356, pp.359, 1990) is a special $gamma=1$ case of the Dehnen potential. The potential is given by:

$$\Phi = -\frac{M}{(r_c + r)}$$

and mass

$$M(r) = M\frac{r^2}{(r + r_c)^2}$$

and density

$$\rho = \frac{M}{2\pi}\frac{r_c}{r}\frac{1}{(r + r_c)^3}$$

10.1.15 hom

% File: hom.c

{bf potname=hom potpars={it $Omega,M,R,tau$}}

10.1.16 hubble

% File: hubble.c

{bf potname=hubble potpars={it $Omega,M,R,b,c$}} where M and R are the core mass and radius. b and c are, if given, the intermediate and short axes can be different from the core radius.

The Hubble profile (BT, pp 39, req. 2-37 and 2-41) has a density law:

$$\rho = \rho_h (1 + (r/r_h)^2)^{-3/2}$$

and an equally simple expression for the projected surface brightness:

$$\Sigma = 2 \rho_h r_h (1 + (r/r_h)^2)^{-1}$$

The derivation of the potential is a bit more involved, since there is no direct inversion, and integration in parts is needed. The cumulative mass is given by:

$$M_h(r) = 4 \pi r_h^3 \rho_h \{ \ln[(r/r_h) + \sqrt{1 + (r/r_h)^2}] - \frac{r/a}{\sqrt{1 + (r/r_h)^2}} \}$$

and the potential

$$\Phi(r) = - \frac{G M_h(r)}{r} - \frac{4 \pi G \rho_h r_h^2}{\sqrt{1 + r}}$$

10.1.17 kuzmindisk

% File: kuzmindisk.c

{bf potname=kuzmin potpars={it $Omega,M,a$}}

Kuzmin (1956) found a closed expression for the potential of an infinitesimally thin disk with a Plummer potential in the plane of the disk (see also BT pp43, eq. 2-49a and 2-49b):

$$\Phi = - \frac{G M}{\sqrt{r^2 + (a + |z|)^2}}$$

and corresponding surface brightness ({it check units})

$$\Sigma = \frac{a M}{2 \pi (a^2 + r^2)^{-3/2}}$$

With $GMa^2 = V_0^2$. This potential is also known as a Toomre n=1 disk, since it was re-derived by Toomre (1963) as part of a series of disks with index n, where this disk has $n=1$.

10.1.18 isochrone

% File: isochrone.c

{bf potname=isochrone potpars={it $Omega,M,R$}}

10.1.19 jaffe

% File: jaffe.c

{bf potname=jaffe potpars={it $Omega,M,r_c$}}

The Jaffe potential (BT, pp.237, see also MNRAS 202, 995 (1983))), is another special $gamma=2$ case of the Dehnen potential.

$$\Phi = -\frac{M}{r_c} \ln \left(\frac{r}{r_c + r} \right)$$

10.1.20 log

% File: log.c

% CTEX Line: 8 {bf potname=log potpars={it $Omega,M_c,r_c,q$}}

The Logarithmic Potential (BT, pp.45, eq. 2.54 and eq. 3.77) has been often used in orbit calculations because of its flat rotation curve. The potential is given by

$$\Phi = \frac{1}{2} v_0^2 \ln \left(r_c^2 + r^2 \right)$$

with $ M_c equiv {1over 2} r_c v_0^2 $ defined as the *core mass*.

10.1.21 mestel

% File: mestel.c

% CTEX Line: 10 {bf potname=mestel potpars={it $Omega,M,R$}}

10.1.22 miyamoto

% File: miyamoto.c

% CTEX Line: 20 {bf potname=miyamoto potpars={it $Omega,a,b,M$}}

$$\Phi = -\frac{M}{....}$$

10.1.23 nfw

% File: nfw.c % CTEX Line: 29

The NFW (Navarro,Frank & White) density is given by

$$\rho = \frac{M_0}{r(r+a)^2}$$

and the potential by

$$\Phi = -4\pi M_0 \frac{\ln(1+r/a)}{r}$$

10.1.24 null

% File: null.c

% CTEX Line: 5

This potential has no other meaning other than to fool the compiler. It has no associates potential, thus the usual potname, potpars,potfile will have no meaning. Use {bf potname=zero} if you want a real potential with zero values.

10.1.25 op73

% File: op73.c

% CTEX Line: 14 {bf potname=op73 potpars={it $Omega,M_H,r_c,r_h$}}

Ostriker-Peebles 1973 potential (1973, ApJ {bf 186}, 467). Their potential is given in the form of the radial force law in the disk plane:

$$F = \frac{M}{R_h^2} \frac{(R_h + R_c)^2}{(r + R_c)^2} \frac{r}{R_h}$$

10.1.26 plummer

% File: plummer.c

% CTEX Line: 8 {bf potname=plummer potpars={it $Omega,M,R$}}

Plummer potential (BT, pp.42, eq. 2.47, see also MNRAS 71, 460 (1911))

$$\Phi = -\frac{M}{(r_c^2 + r^2)^{1/2}}$$

10.1.27 plummer2

% File: plummer2.c

10.1.28 rh84

% File: rh84.c

% CTEX Line: 20 {bf potname=rh84 potpars={it $Omega,B,a,A,r_0,i_0,j$}}

This 2D spiral and bar potential was used by Robert and collaborators in the 70s and 80s. For counterclockwise streaming, this spiral is a trailing spiral when the pitch angle (i_0) is positive. Within a radius r_0 the potential becomes barlike, with the bar along the X axis. At large radii the spiral is logarithmic. References:

Roberts & Haussman (1984: ApJ 277, 744)

Roberts, Huntley & v.Albada (1979: ApJ 233, 67)

10.1.29 rotcur0

% File: rotcur0.c

% CTEX Line: 9 {bf potname=rotcur0 potpars={it $Omega,r_0,v_0$}}

The forces returned are the axisymmetric forces as defined by a linear-flat rotation curve as defined by the turnover point r_0,v_0. The potential is not computed, instead the interpolated rotation curve is returned in as the potential value.

10.1.30 rotcur

% File: rotcur.c

% CTEX Line: 14 {bf potname=rotcur potpars={it $Omega$} potfile={it table(5NEMO)}}

The forces returned are the axisymmetric forces as defined by a rotation curve as defined by a table given from an ascii table. The potential is not computed, instead the interpolated rotation curve is returned in as the potential value.

This version can only compute one version; i.e. on re-entry of inipotential(), old versions are lost.

10.1.31 sh76

% File: sh76.c

{bf potname=sh76 potpars={it $Omega,A,alpha,epsilon$}}

This bar potential was used by Sanders and Huntley (1976) and also used in Sanders (2019). The density perturbation is given by

$$\sigma(r,\theta) = Ar^{-\alpha}(1 + \epsilon \cos 2\theta)$$

and the potential

$$\Phi(r,\theta) = -2\pi G c_1 A r^{-\alpha+1} \frac{1}{1-\alpha}(1 + \beta(\alpha-1)\cos 2\theta)$$

where

$$\beta = \frac{(2-\alpha)}{\alpha(3-\alpha)}\epsilon$$

$$c_1 = \frac{\Gamma[\frac{1}{2}(2-\alpha)]\Gamma[\frac{1}{2}(\alpha+1)]}{\Gamma[\frac{1}{2}\alpha]\Gamma[\frac{1}{2}(3-\alpha)]}$$

10.1.32 teusan85

% File: teusan85.c

% CTEX Line: 25 {bf potname=teusan85}

This potential is that of a barred galaxy model as described in Teuben & Sanders (1985) This bar is oriented along the X axis. This is the 2D version for forces. This version should give (near) identical results to {bf bar83} and very simlar to {bf athan92}.

10.1.33 triax

% File: triax.c

% CTEX Line: 11 {bf potname=triax}

A growing bi/triaxial potential

10.1.34 twofixed

% File: twofixed.c

% CTEX Line: 16 {bf potname=twofixed potpars={it $Omega,M_1,x_1,y_1,z_1,M_2,x_2,y_2,z_2$}}

This potential is defined by two fixed points, with different masses and positions. Orbits in this potential exhibit a number of interesting properties. One well known limit is the {tt stark problem}, where one of the two bodies is far from the other and near-circular orbits near the central particles are studied. Another is the limit or two particles near to other and orbits that circumscribe both particles.

10.1.35 plummer4

% File: plummer4.c

% CTEX Line: 10 potname=plummer potpars=:math:*Omega,M,R*

Plummer potential (BT, pp.42, eq. 2.47, see also MNRAS 71, 460 (1911))

$$\Phi = -\frac{M}{\left(r_c^2 + r^2\right)^{1/2}}$$

10.1.36 vertdisk

% File: vertdisk.c

10.1.37 tidaldisk

% File: tidaldisk.c % CTEX Line: 8

Tidal field exerted by a (plane-parallel) stellar disk on a cluster passing through with constant vertical velocity. Useful for simulations of disk-shocking of, say, globular clusters

The following three density models are available

1. thin disk:

$$\rho(z) = \Sigma\delta(z)$$

2. exponential disk:

$$\rho(z) = \frac{\Sigma}{2h} \exp \frac{-|z|}{h}$$

3. sech^2 disk:

$$\rho(z) = \frac{\Sigma}{4h} sech^2 \frac{z}{2h}$$

Parameters (to be given by potpars=...) are:

```
par[0] = not used (reserved for pattern speed in NEMO)
par[1] = h scale-height par[1] = 0 -> thin disk
par[1] > 0 -> vertically exponential disk
par[1] < 0 -> sech^2 disk with h=|par[1]|
par[2] = Sig disk surface density
par[3] = Vz constant vertical velocity of cluster center
par[4] = Z0 cluster center z-position at t=0
par[5] = add boolean: add tidal potential or not?
```

We always assume G=1.

If you want to include the acceleration of the disk on the cluster as a whole, rather than assume a constant velocity, use vertdisk.c

Some words on the mechanics

Assume that the plane-parallel disk potential and force are given by

$$\Phi(Z), F(Z) = -\Phi'(Z).$$

Then, the tidal force exerted on a star at position z w.r.t. to cluster center, which in turn is at absolute height Zc = Z0 + t Vz, is simply

$$F_t(z) = F(Zc + z) - F(Zc).$$

Integrating this from z=0 to z gives the associated tidal potential as

$$\Phi_t(z) = \Phi(Zc + z) - \Phi(Zc) + zF(Zc).$$

Whenever the tidal force & potential are desired at a new time t, we pre-compute Zc and the plane-parallel potential and force at $Z=Zc$. Note that when both Zc and $Zc+z$ are outside of the mass of the disk (and $Z=0$ is not between them), both tidal force and potential vanish identically. The standard treatment of tidal forces corresponds to approximating (2) by $F(Zc) + z * F'(Zc)$. This method, however, breaks down for disks that are thin compared to the cluster, while our method is always valid, even for a razor thin disk.

10.1.38 polynomial

% File: polynomial.c

% CTEX Line: 9 {bf potname=polynomial potpars={it $Omega,a0,a1,a2,a3,....$}}

Polynomial potential

$$\Phi = a_0 + a_1 r + a_2 r^2 + ...a_N r^N$$

where any unused coefficients will be set to 0. Up to 16 (defined as MAXPOW) can be used.

10.1.39 wada94

% File: wada94.c

% CTEX Line: 11 {bf potname=wada94 potpars={it $Omega,c,a,epsilon$}}

Wada (1994, PASJ 46, 165) and also Wada & Have (1992, MN 258, 82) used this potential in the study of gaseous orbits in barred galaxies.

$$\Phi = \Phi_0 + \Phi_b$$

where Phi_1 is the Toomre potential with scalelength a

$$\Phi_0 = -\frac{1}{\sqrt{R^2 + a^2}}$$

and

$$\Phi_b = -\epsilon \frac{aR^2}{(R^2 + a^2)^2}$$

A relationship for the axisymmetric component is

-sqrt(27/4)

10.1.40 zero

% File: zero.c

% CTEX Line: 6 {bf potname=zero}

Zero potential

$$\Phi = 0$$

10.2 Accellerations

This is a falcON addition. They are defined in `$FALCON/src/public/acc`, where their list (for installation) is defined in `$FALCON/makepub`.

For a new potential, say `GasPotential.cc`, add to `$FALCON/makepub`:

```
acc_pub         :=
                        ...
                        $(ACC)GasPotential.so
```

and a proper dependency as well:

```
$(ACC)GasPotential.so: $(SACC)GasPotential.cc $(ACCT) $(defacc_h) $(makefiles)
                       $(MAKE_ACC)
```

CHAPTER

ELEVEN

UNITS AND COORDINATE SYSTEMS

11.1 Coordinate Systems

Astronomy is well known for its confusing coordinate systems: nature and math don't always look at things through the same mirror. For example, the so-common (mathematical) right handed coordinate system that we call X-Y-Z does not neatly fit in with our own Galaxy, which rotates counter-clockwise (meaning the angular moment vector points to the galactic south pole). Sky coordinates (e.g. Right-ascension Declination) are is pinned on the sky, looking up, instead of on a sphere, looking down, and become a left-handed coordinate system where the "X" coordinate increases to the left.

Here are some examples, and their respective NEMO programs that deal with this. See also their {it manual} pages for more detailed information.

- mkgalorbit

orbits in our galaxy have to deal with the UVW space velocities in the galactic coordinate system. There is no formal definition of a spatial XYZ system, other than Z=0 being the galactic plane. So, where to put the sun if the galactic center is (0,0,0) is purely by convention. It so happens that (-R0,0,0) is convenient since galactic longitude and latitude can be easily expressed as `atan2(y,x)` and `atan2(z,sqrt{x^2+y^2})` resp.

- mkspiral, mkdisk

These programs use the `sign=` keyword to set the sign of the angular moment vector. Positive means thus counter-clockwise rotation in this convention. Of course with tools like `snapscale` and `snaprotate` snapshots can always be re-arranged to fit any schema.

- snaprotate

in order for a known object (e.g. a disk) to be viewed as an ellipse with given position angle and ellipticity this program uses a series of Eulerian angle rotations. Apart from the astronomical convention to using position angle as a counter clockwise angle measured from north, these numbers do no trivially convert to the angles we use on the projected sky. There are some examples in the manual page, which we briefly highlight here.

item The first example describes the projection of disks of spiral galaxies. If kinematic information is also known, the convention is to use the position angle of the receding side of the galaxy. The inclination still has an ambiguity, which could be used to differentiate based on which is the near and far side, but this would result in values outside the commonly used 0..90 range. Thus the sense of rotation (sign of the angular momentum) is a more natural way, with again `sign=-1` for counter-clockwise rotating (as seen for us projected on the sky)

Here are NEMO commands to create an example velocity fields of these two galaxies with the right orientation and velocity field (with an arbitrary rotation curve of course):

```
mkdisk - 1000 sign=+1 mass=1 |\                     # clock wise rotating
  snaprotate - - theta=-30,-160 order=yz |\
  snapgrid - vel-m33.ccd moment=-1
```

(continues on next page)

(continued from previous page)

```
mkdisk - 1000 sign=-1 mass=1 |\                    # counter clock wise rotating
  snaprotate - - theta=+22,170  order=yz |\
  snapgrid - vel-m51.ccd moment=-1
```

- The second example is that of a barred galaxy, or two nested disks if you wish. Here an addition angle, the difference between the major axis and that of the bar), is a parameter.
- snapgalview
- ccdfits

11.2 Units

Perhaps better to refer to a few man pages we have on this: units(1NEMO) , units(5NEMO) , and constants(5NEMO) .

CHAPTER

TWELVE

TROUBLESHOOTING (*)

Fatal errors are caught by most NEMO programs by calling the function *error*; it reports the name of the invoked program and some offending text that was the argument to the function, and then exits. If the **$ERROR** level is larger than 0, an error call can postpone the exit for the specified amount of times. If the **$DEBUG** level is positive, programs also produce a coredump, which can be further examined with local system utilities such as {it adb(1)} or {it dbx(1)}. Most of the error messages should be descriptive enough, but a list is being compiled for the somewhat less obvious ones.

Another annoying feature can be large amount of environment variables used by packages. NEMO is no exception. In Section~ref{s-envvar} below all of the environment variables used by NEMO are listed and their functionality. Sometimes they interfere if used in conjunction with other packages.

12.1 List of Run Time Errors

This section presents an alphabetical list of some of the fatal error messages, as generated by *error(3NEMO)* Although this list is not meant to be complete, it hopes to report on the most obscure errors found when using NEMO, and their possible cures. The ones not listed here should be descriptive enough to guide the user to a solution. Sometimes execution errors can be better understood when the **DEBUG** environment variable is set to a high(er) value, or the **debug=** system keyword is added to the command-line. See Appendix~ref{a:iface} on it's use.

Now the list of error messages and their possible cures:

`assertion failed:  file f line n`

The program was compiled with an active {it assert(2)} macro. An expression was expected to be true at this point in the program. Program exits when this was not the case. An infamous failed assertion is {tt file load.c line 91} or thereabouts, part of the `hackcode1` N-body integrator. Two particles were too close (or on top of each other) such that space could not be subdivided within 32 levels of the oct tree. There is no good solution to this problem.

`findstream:  No free slots`

Too many open files. Either the program didn't cleanup (close) its files after usage, or your current application really needs more than the default 16 which is {tt #define}d in {tt filesecret.h} by the macro {tt StrTabLen}. Recompile the {tt filesecret.c} and the appropriate application tasks.

`get_tes:  set=A tes=B`

This points to a programming error or error in the logic during {it filestruct(3NEMO)} I/O. During program execution a hierarchical set A was requested to be closed, but the program was still within set B (set B had not been closed yet).

`gethdr:  ItemFlag = 0164`

Input was attempted on a file assumed to be in old {it filestruct(5NEMO)} format. Apparently it was not, file may also have been in new filestruct format. {bf Hints:} Try {it tsf(1NEMO)}, {it hd(1NEMO)}, {it od(1)} and as last resort {it more(1)}.

`gethdr: bad magic: 0164`

Input was attempted on a file assumed to be in {it filestruct(5NEMO)} format. Apparently it was not. {bf Hints:} Try {it tsf(1NEMO)}, {it hd(1NEMO)}, {it od(1)} and as last resort {it more(1)}.

`loadobj: file must be in .o format`

It is possible that the non-portable dynamic object loader (loadobj.c) indeed proves to be non-portable here. Either you requested a wrong file, which is not in object format, or this UNIX version has a different object file structure. {bf Cure:} a lot of hacking in loadobj.c, assuming no pilot error.

`loadobj: undefined symbol _XXXX`

There are three possible causes. It might be that you have just supplied a 'new' object file to a program, which happens to call a function which was not linked in by the calling program. Find out in which filestructure 'group' (Nbody, Orbit, Image) your invoked program falls, and add appropriate dummy code to the library function. {it E.g.} in the case of {it potential(5NEMO)} data files, you might have to add a specific math function to `$NEMO/inc/mathlinker.h`, or add some coding to the end of `$NEMO/src/orbit/potential/potential.c` and rebuild the appropriate {it orbit(1/3NEMO)} library/commands. The standard UNIX utility {it nm(1)} help finding all undefined symbols in an object file. Cross check this with the executable.

The second possibility is that the executable was stripped, i.e. it had no symbol table. Try {tt nm(1)} to find out, or use {tt file(1)}.

The last cause is much more serious: the non-portable dynamic object loader (loadobj.c) indeed seems to be non-portable here. This might mean serious hacking in loadobj.c, we cannot give any advice on this right now.

`loadobj: word relocation not supported`

It is possible that the non-portable dynamic object loader (loadobj.c) indeed proves to be non-portable here. This might mean serious hacking in loadobj.c.

`makecell: need more than XX cells; increase fcells=`

This is actually sort of a pilot error, but may sound a bit obscure to a beginning user. Space for cells used in the hackcode force calculation is allocated dynamically, as well as for the particles. 'fcells' is the ratio of allocated cells to particles and is a parameter to most programs who use the hackcode force calculation. For small N-body systems (less then about 100) this ratio may have to be increased, 2 usually is enough. Note that in the regime where fcells is required larger, the hackcode force calculation is usually not the most efficient method to compute forces anyhow.

`mysymbols: file must be executable`

It is possible that the non-portable dynamic object loader (loadobj.c) indeed proves to be non-portable. The program which you just executed does not have the executable format the dynamic object loader thinks it should have.

`No man entry for XXX.Y`

`No manual entry for XXX`

No online manual page for this, although perhaps the MANPATH environment variable has not been properly set, or your UNIX version does not support multiple man-root directories, in which case consult the manual page of man(1).

`put_snap_XXX: not implemented`

Here 'XXX' may be 'key' or 'aux' or something else. You have an old version of the code, while the datastructure of the snapshot has an 'XXX' (You may confirm this with `tsf`. Recompile the program with a more recent version of $langle$snapshot/put_snap.c$rangle$ and possibly $langle$snapshot/body.h$rangle$.

`readparam: No interactive input allowed`

The keyword {bf help=} or the equivalent environment variable {bf HELP} has been assigned a digit to request interactive input. In addition you requested some file I/O through either redirection or piping. Get rid of at least one of them.

`rsh: could not execute rsh`

Program could not execute itself on a remote machine. It may have various reasons for failing. The {tt rsh} program may not exist on your host, in which case the {it getparam(3NEMO)} might as well have been compiled without the {bf REMOTE} flag. The other possibility is that the {tt .rhosts} file on your system does not contain an entry for the machine you wanted to rsh to. In interactive usage it will then ask for a password, executed through {it execvp(3)} normally fails. A third possibility is that the remote machine did not have the executable present.

`Badly placed ()'s`

`bash: syntax error near unexpected token (`

You tried to pass an expression with parentheses, but since the UNIX shell gives them special meaning, you need to *escape* them from the shell, e.g.

```
% snapplot in=snap001 xvar=r yvar=log(aux)
```

you need to type any of:

```
% snapplot in=snap001 xvar=r 'yvar=log(aux)'
% snapplot in=snap001 xvar=r yvar=log\(aux\)
```

12.2 Environment Variables used by NEMO

Occasionally NEMO's environment can interfere with those of other packages. The following list of environment variables have some meaning to NEMO. A default is usually activated when the environment variable is absent.

- **BELL** If BELL is set (1), a number of user-interface routines become noisy. The default is 0.
- **BTRPATH** List of directories where {it bodytrans(3NEMO)} functions can be stored for retrieval. The default is {tt /usr/nemo/obj/bodytrans}. Normally set to {tt ".:$NEMOOBJ/bodytrans"} in {tt NEMORC}.
- **DEBUG** Debug level, must be between 0 and 9. The higher the number, the more debug output appear on {it stderr}. The default is 0. See {it getparam(3NEMO)}. DEBUG is also used as system keyword, in which case the environment variable is ignored.
- **EDITOR** Editor used when helplevel 4 is included. The default is {tt vi} (see {it vi(1)}). See also {it getparam(3NEMO)}.
- **ERROR** Error level for irrecoverable errors. If this environment variable is present, and its numeric value is positive, this is the number of times that such fatal error calls are bypassed; after that the the program really stops. See also {it getparam(3NEMO)}.
- **HELP** Help level, can be any combination of numerically adding 0, 1, 2, and 4, and any combination of '?', 'a', 'h', 'p', 'd', 'q', 't' and 'n'. See {it getparam(3NEMO)}. HELP is also used as system keyword, in which case the environment variable is ignored. The numeric part of the help string should come first.
- **HISTORY** Setting it to 0 causes history data NOT to be written, the default is 1 (see {it getparam(3NEMO)}). A few old programs may use the keyword {tt amnesia=} for this.
- **HOSTTYPE** In case of multiCPU environment, which has to be served from the same {tt NEMORC} and/or {tt .cshrc} file, this variable will have the CPU type in it, {it e.g.} {tt sun3} or {tt sun4}, which are used to break up the {tt bin}, {tt lib} and {tt obj} directories. It is also used in some Makefiles.
- **MANPATH** Used by UNIX to be able to address more than one area of manual pages. Normally set to {tt $NEMO/man:/usr/man} by the {tt NEMORC} file. Does not work in Ultrix 3.0, but perhaps the {tt -P} switch may be used.

- **NEMO** The root directory for NEMO. Normally the only environment variable which a user has to define himself, in his {tt .cshrc} startup file. No default.
- **NEMOBIN** Directory where nemo's binaries live, defined in {tt NEMORC}. No default.
- **NEMODOC** Directory where the *.doc files for mirtool and miriad shell should be looked for. The system default is `$NEMO/man/doc`, set by NEMORC. No default.
- **NEMODEF** Directory where keyword files from {tt mirtool/miriad } are stored/retrieved. The default is the current directory.
- **NEMOLIB** Directory where nemo's libraries live. Normally set by NEMO. No default.
- **NEMOLOG** Filename used as logfile for tasks submitted through `nemotool`.
- **NEMOOBJ** Directory were (binary) object files live. They are used by a variety of nemo programs, and generally do not concern the user. Usually set by `NEMORC`.
- **NEMOPATH** Same as NEMO, but kept for historical reasons. It is normally defined in the `NEMORC` file. *–deprecated–*
- **NEMOSITE** The site name, which is also an alias used in case the import/export features with the central site are to be maintained.
- **PATH** UNIX search-path for executables, normally set in your own shell startup file (.cshrc or .login). Should contain NEMOBIN early in the path definition, before /usr/bin and /bin to redefine the cc and make programs. See Appendix~ref{a:setup}
- **POTPATH** List of directories where *potential(5NEMO)* functions can be stored. The default is `/usr/nemo/obj/potential`.
- **REVIEW** If this variable is set, the REVIEW section is entered before the program is run. [default: not set or 0]
- **YAPP** Yapp graphics output device. Usage depends which {it yapp(3NEMO)} the program was linked with. See also {it getparam(3NEMO)} and {it yapp(5NEMO)}. YAPP is also used as system keyword, in which case the environment variable is ignored.

See also the manual pages of {it files(1NEMO)}.

CHAPTER THIRTEEN

REFERENCES

Here are some references. We also keep a list of codes with papers in NEMO.

A Hierarchical O(N log N) Force-Calculation - Barnes, J.E. and Hut, P. (Nature, Vol. 324, pp 446 1986). https://ui.adsabs.harvard.edu/abs/1986Natur.324..446B

Use of Supercomputers in Stellar Dynamics - (S.M. McMillan and P. Hut. Berlin: Springer-Verlag 1987).

Hierarchical N-body Methods - L. Hernquist, (Computer Physics Communications, Vol. 48, p. 107, 1988.) https://ui.adsabs.harvard.edu/abs/1988CoPhC..48..107H

Computer Simulation Using Particles - R. W. Hockney and J. W. Eastwood (Adam Hilger; Bristol and Philadelphia; 1988)

The Numerical Solution of the N-body Problem - (L. Greengard. Comp. in Phys. pp. 142, mar/apr 1990.)

The Art of N-body Building J.A. Sellwood (Ann. Rev. Astron. Astrophys. Vol. 25, pp. 151 1987). https://ui.adsabs.harvard.edu/abs/1987ARA%26A..25..151S

Galactic Dynamics - Binney, J. and Tremaine, S. (Princeton U. Press; Princeton; 1987) [BT1]

Galactic Dynamics 2nd Ed. - Binney, J. and Tremaine, S. (Princeton U. Press; Princeton; 2008) [BT2]

Astrophysical Recipes - The art of AMUSE - Simon Portegies Zwart and Steve McMillan IOP Publishing, Bristol, UK, 2018

CHAPTER

FOURTEEN

RELATED CODES (*)

Here we summarize some codes used in stellar dynamics that are similar to NEMO. We only list codes that are (publically) available. SPH/hydro codes are currently not included. See also ASCL to find more codes.

First we start off with some expanded examples on a few specific codes that have a tighter connection to NEMO:

14.1 AMUSE

AMUSE (Astrophysical Multipurpose Software Environment) originates some ideas from its predecessors: ACS, StarLab and NEMO, but uses the python language. Another feature of AMUSE is that python is also the *glue* shell between legacy codes that can orchestrate simulations taking components from different codes, whereas in NEMO legacy codes have a NEMO CLI interface, at best.

For seasoned AMUSE users, here we highlight some differences between the two, and give some examples how to achieve the same task in NEMO and AMUSE.

14.1.1 Differences

- **Shell**: NEMO uses a Unix shell, AMUSE uses python (ipython, jupyter, . . .).
- **Community Code**: Both packages maintain a tight connection to legacy software and community codes. You can find them in **$AMUSE/src/amuse/community** and **$NEMO/usr** resp.
- **Units**: NEMO uses dimensionless values, and units are implied. Most programs actually use virial units where G=1, but there are a few programs (e.g. galaxy, nbodyX) that use other units. The units(1NEMO) tries to help you converting. AMUSE (optionally?) attaches units to numbers , using a python trick, e.g.

```
from amuse.units import units

mass   = 1.0 | units.MSun
```

astropy users might be a bit baffled, since this looks very different. But

```
m1 = mass.as_astropy_quantity()
```

will look more familiar. In pure **astropy** it might look as follows:

```
from astropy import units as u

m = 1.0 * u.solMass
m2 = m.to(u.kg).value
```

14.1.2 Examples: Creating a Plummer sphere

Here we create a Plummer sphere, in virial units, in NEMO, and display an X-VX projection on the sky in a shell session:

```
source /opt/nemo/nemo_starts.sh

mkplummer p100 100
snapplot p100 xvar=x yvar=vx
```

or in the style of using pipes this can be a one liner

```
mkplummer - 100 | snapplot - xvar=x yvar=vx
```

And in AMUSE the following python session can do something similar:

Todo: figure out the right py-gnuplot

```
from amuse.units import units
from amuse.units import nbody_system
from amuse.ic.plummer import new_plummer_sphere

convert_nbody = nbody_system.nbody_to_si(100.0 | units.MSun, 1 | units.parsec)
plummer = new_plummer_sphere(1000, convert_nbody)


plotter = Gnuplot.Gnuplot()
plotter.splot(plummer.position.value_in(units.parsec))
```

The AMUSE manual has some NEMO I/O examples.

14.1.3 Installation

For the benefit of NEMO users, AMUSE can usually be installed *easily* as follows:

```
pip install amuse
```

but this can take a while as it finds the right dependencies and needs to compile massive amounts of code. Some of these can easily fail if you don't have the correct prerequisites (e.g. MPI).

A potentially faster way is to first install the AMUSE frame work and then the selected module(s):

```
pip install amuse-framework
pip install amuse-seba amuse-brutus
```

There are many more details in the AMUSE installation manual.

14.2 Martini

There is an expanded example in https://teuben.github.io/nemo/examples/eagle.html, and some supporting notes are in `$NEMO/usr/martini`.

It can also be installed with

```
pip install astromartini
```

14.3 ClusterTools (*)

This python package can also read NEMO (gyrfalcON) files. More on this later.

14.4 ZENO

ZENO is Josh Barnes' version of an earlier version of NEMO that he continues to develop. We also keep a zeno manual page highlighting some differences.

> **Warning:** Adding ZENO to NEMO will result in some programs that have duplicated names.

14.4.1 Installation

For the benefit of NEMO users, ZENO can usually be installed as follows:

```
cd $NEMO/usr/zeno
make zeno
```

This will currently download two repos: zeno_jeb and zeno_pjt. Pick one by using a symlink to become the official one for the install:

```
ln -s zeno_pjt zeno
source zeno_start.sh
cd zeno
make -f Zeno
```

Now various ZENO commands are available:

```
ls $ZENOPATH/bin
```

14.5 List of Related Codes

Todo: this list needs to be annotated and spiced up with links.

ACS *The Art of Computational Science - How to build a computational lab.* In C++ and ruby. With ideas from NEMO and StarLab. | http://www.artcompsci.org/

agama *Action-based galaxy modeling framework*. Also usable via `$NEMO/usr/agama`. | http://ascl.net/1805.008

AMIGA *Adaptive Mesh Investigations of Galaxy Assembly*. | http://ascl.net/1007.006

AMUSE *Astrophysical Multipurpose Software Environment*. Also usable via `$NEMO/usr/amuse`. | http://ascl.net/1107.007

arepo *Cosmological magnetohydrodynamical moving-mesh simulation code*. | http://ascl.net/1909.010

bhint High-precision integrator for stellar systems | https://ascl.net/1206.005

bonsai N-body GPU tree-code, in *AMUSE*. | http://ascl.net/1212.001

brutus See also *AMUSE*

Catena Ensemble of stars orbit integration | https://ascl.net/1206.008

CGS Collisionless Galactic Simulator. Also usable via *NEMO* with the runCGS interface. | http://ascl.net/1904.003

ChaNGa Charm N-body GrAvity solver | http://ascl.net/1105.005

clustertools A Python package with tools for analysing star clusters. | https://github.com/webbjj/clustertools

DICE Disk Initial Conditions Environment | https://ascl.net/1607.002

Fewbody Numerical toolkit for simulating small-N gravitational dynamics | http://ascl.net/1208.011

fractal *A parallel high resolution Poisson solver for an arbitrary distribution of particles*. | https://github.com/jensvvillumsen/Fractal

gadgetX *A Code for Cosmological Simulations of Structure Formation*. Several versions available, X=1,2,3,4. gadget2 also available via `$NEMO/usr/gadget`. | http://ascl.net/0003.001

Gala *Galactic astronomy and gravitational dynamics*. | http://ascl.net/1302.011

galaxy *N-body simulation software for isolated, collisionless stellar systems*. The older version still usable via *NEMO* with the **rungalaxy** interface. | http://ascl.net/1904.002

galpy Galactic dynamics package (python) | http://ascl.net/1411.008

GalPot Galaxy potential code | http://ascl.net/1611.006

GANDALF Graphical Astrophysics code for N-body Dynamics And Lagrangian Fluids | http://ascl.net/1602.015

GENGA Gravitational ENcounters with Gpu Acceleration | http://ascl.net/1812.014

Glnemo2 Interactive Visualization 3D Program | http://ascl.net/1110.008

GraviDy Gravitational Dynamics. Also usable via *NEMO* with the **rungravidy** interface. | http://ascl.net/1902.004

gsf galactic structure finder | http://ascl.net/1806.008

gyrfalcON Dehnen's code. Included in *NEMO* | http://ascl.net/1402.031

hermite In *AMUSE*.

HiGPUs *Hermite's N-body integrator running on Graphic Processing Units*. Part of *AMUSE*. | https://ascl.net/1207.002

HUAYNO Hierarchically split-Up AstrophYsical N-body sOlver N-body code. Part of *AMUSE*. | http://ascl.net/2102.019

Hydra A Parallel Adaptive Grid Code | http://ascl.net/1103.010

hnbody also | http://ascl.net/1201.010

ICICLE Initial Conditions for Isolated CoLlisionless systems https://ascl.net/1703.012 | http://ascl.net/1703.012

identikit 1: A Modeling Tool for Interacting Disk Galaxies. | https://ascl.net/1011.001 2: An Algorithm for Reconstructing Galactic Collisions. | https://ascl.net/1102.011

InitialConditions Website with a collection of programs for integrating the equations of motion for N objects, implemented in many languages, from Ada to Swift. | http://www.initialconditions.org/codes

JSPAM Interacting galaxies modeller | http://ascl.net/1511.002

limepy Lowered Isothermal Model Explorer in PYthon. | https://ascl.net/1710.023

MARTINI Mock spatially resolved spectral line observations of simulated galaxies Also usable via `$NEMO/usr/martini`, see example. | http://ascl.net/1911.005

mcluster Make a plummer. Also usable via *NEMO* | http://ascl.net/1107.015

McScatter Three-Body Scattering with Stellar Evolution | http://ascl.net/1201.001

mercury *A software package for orbital dynamics*. In *AMUSE*. | http://ascl.net/1209.010

MYRIAD N-body code for simulations of star clusters | https://ascl.net/1203.009

nbodyX Where X=0,1,2,3,4,5,6,6++,7 Also usable via *NEMO* with the runbodyX interface. | http://ascl.net/1904.027

nbody6tt Tidal tensors in N-body simulations | http://ascl.net/1502.010

nbodykit Massively parallel, large-scale structure toolkit | http://ascl.net/1904.027

nbody6xx Alias for nbody6++ Also usable via *NEMO* | http://ascl.net/1502.010

nemesis Another code to document. | http://ascl.net/1010.004

NEMO Stellar Dynamics Toolbox. Our current version is 4. | http://ascl.net/1010.051

NIGO Numerical Integrator of Galactic Orbits | https://ascl.net/1501.002

N-MODY A code for Collisionless N-body Simulations in Modified Newtonian Dynamics | http://ascl.net/1102.001

octgrav in *AMUSE*. | http://ascl.net/1010.048

partiview Immersive 4D Interactive Visualization of Large-Scale Simulations | https://ascl.net/1010.073

PENTACLE Large-scale particle simulations code for planet formation | http://ascl.net/1811.019

petar Another code to document. | http://ascl.net/2007.005

plumix another | http://ascl.net/1206.007

pNbody A python parallelized N-body reduction toolbox https://ascl.net/1302.004

pycola N-body COLA method code | http://ascl.net/1509.007

pyfalcon Python interface for *gyrfalcON* | https://github.com/GalacticDynamics-Oxford/pyfalcon

pynbody N-Body/SPH analysis for python. | http://ascl.net/1305.002

QYMSYM A GPU-accelerated hybrid symplectic integrator | https://ascl.net/1210.028

RAMSES A new N-body and hydrodynamical code | https://ascl.net/1011.007

Raga Monte Carlo simulations of gravitational dynamics of non-spherical stellar systems | http://ascl.net/1411.010

rebound Multi-purpose N-body code for collisional dynamics | https://ascl.net/1110.016 Also usable via *NEMO*

SecularMultiple Hierarchical multiple system secular evolution model | http://ascl.net/1909.003

sidm-nbody Monte Carlo N-body Simulation for Self-Interacting Dark Matter | http://ascl.net/1703.007

slimplectic Discrete non-conservative numerical integrator | http://ascl.net/1507.005

smalln in *AMUSE*. | http://ascl.net/1106.012

smile orbits? | http://ascl.net/1308.001

SpaceHub High precision few-body and large scale N-body simulations | http://ascl.net/2104.025

SpheCow Galaxy and dark matter halo dynamical properties | http://ascl.net/2105.007

Starlab Also usable via *NEMO* | https://ascl.net/1010.076

Swarm-NG Parallel n-body Integrations | https://ascl.net/1208.012

Torch Coupled gas and N-body dynamics simulator | http://ascl.net/2003.014

TPI Test Particle Integrator | http://ascl.net/1909.004

UNSIO Universal Nbody Snapshot I/O - See examples.

VINE A numerical code for simulating astrophysical systems using particles | http://ascl.net/1010.058

yt A Multi-Code Analysis Toolkit for Astrophysical Simulation Data | https://ascl.net/1011.022

ZENO Barnes version that was derived from NEMO V1. | https://ascl.net/1102.027 Also usable via *NEMO* , but watch out for duplicate names of programs

A large number of these codes can also be found by searching on ASCL, for example: https://ascl.net/code/search/dynamics and https://ascl.net/code/search/hermite and https://ascl.net/code/search/orbit and https://ascl.net/code/search/nbody. The last time this list was cross-checked was … 16-jul-2021.

14.6 Categories

Such a niche list of codes made me wonder what kind of meta-data we could use to categorize such dynamics codes, but then perhaps along the lines of the Unified Astronomy Thesaurus project.

dynamics - nbody, orbit, integrator, sph, hydro, analysis, integrator

CHAPTER

FIFTEEN

GLOSSARY

accelleration Dataformat used in the *falcon*

bodytrans Dataformat that is used to perform arbitrary operations on expression variables used in snapshot's.

ccd Synonymous for image; most programs in NEMO which handle images start or end with `ccd`, e.g. `ccdfits`, `fitsccd`, `ccdmath`.

ECSV (Enhanced Character Separated Values) a popular self-describing ASCII table format popularized by astropy

falcon A subpackage in NEMO that hosts the gyrfalcON code.

fie Most expressions that you give to program keywords are parsed by *nemofie* and eventually `fie`. (Nomenclature borrowed from *GIPSY*)

FITS "Flexible Interchange Transport System", a standard dataformat used to interchange data between machines. Commonly used for images, but has since its inception be modified to store tables as well.

FWHM (Full Width Half Max): the effective resolution of the beam if normally given in **FITS** keywords BMAJ,BMIN,BPA. The term **resolution** is used interchangeably.

GIPSY The Groningen Image Processing System. A number of concepts in NEMO, and a few routines, have been taken liberally from GIPSY.

history Each NEMO dataset normally contains a data history in the form of of history items at the beginnging of the dataset. They are normally kept track of by the data processing programs, and can be displayed with the program `hisf`.

image Dataformat in NEMO, used to represent 2- and 3-D images/data cubes. See also *ccd*.

miriad Another astronomical data reduction package, from which we have borrowed some user interfaces which are plug-compatible with our command-line syntax.

orbit Dataformat in NEMO used to represent a stellar orbit; most programs in NEMO which handle orbits start or end with *orb*.

pixel PIXEL/voxel: an area in 2D or 3D representing

potential Dataformat in NEMO used to represent a potential; most programs in NEMO which handle potentials start or end with *pot*. Related are the *accelleration*

program keyword Keywords that are defined by the program only. They can be seen by using the **help=** keyword (in itself being a system keyword).

review A small user interface that pops up when a program is interrupted. Type `quit` to exit it, or `?` for help. This feature of the user interface may not be installed in your version.

set Compound hierarchical data-structure of a structured file. They are the equivalent of a C structure.

snapshot Dataformat used in NEMO to represent an N-body system. Many programs that handle {it snapshot}'s in NEMO start or end with *snap*.

structured file The binary data NEMO writes is in a hierarchical structured format. Programs like tsf rsf, and csf perform general and basic I/O functions on such files. They are hierarchical structured sets, much like how binary XML files would look.

system keyword Global keyword that every NEMO program knows about, and are not listed in the (program) keywords that can be seen by issuing e.g. **help=** (in itself being a system keyword). This concept originated in GIPSY

table A table consists of rows and columns of values, numbers or text. Most commonly stored in ASCII. Less well defined, it is one of the four data types in NEMO.

yapp "Yet Another Plotting Package", the library definition that is used by all programs that produce graphics output. It is kept very simple. The **yapp=** system keyword controls the graphics device definitions/capabilities.

CHAPTER

SIXTEEN

RESEARCH SOFTWARE ENGINEERING

NEMO was rooted in Unix, and can be called legacy, there has been a constant change in the software engineering (SE) aspects it was and is using. This page hopes to summarize those, especially useful for those with no domain knowledge.

- Like many research programming projects, NEMO grew organically but over the years has tried to absorb good modern SE habbits to make the project sustainable and grow
- NEMO uses a Unix like directory tree, with a few familiar 3 and 4 letter names (src, usr, man, etc, bin, lib, opt, obj, tmp, docs, data, local)
- **We now use git (on github):**
  - since its inception in 1986 we have been sharing the code and absorbed new codes from collaborators.
  - We started out with email (1987) and *shar*, then moved to *CVS* in 2000, and finally to *git* in 2017, hosted on github for the moment
  - issues (for bug reports) and pull requests are the preferred way to collaborate
- installing NEMO occurs in place, we use **configure** to track down the system dependent portions (or force them), and a set of **make build** steps will compile and build NEMO. There is no **install** step, so users will need to modify their *shell environment* to take advantage of the new tools. There is no support for a build tree that's different from the source tree.
- In terms of documentation there are the unix manual (`man`) pages, very hyperlinked, and the old *Users and Programmers Guide* (in latex). The man pages are still used, but also made available via html (see https://teuben.github.io/nemo/man_html/index1.html), and the newer sphinx based documentation shared on https://astronemo.readthedocs.io.
- Q: Could you explain the software development lifecycle that you use in this project?
- Q: Is the package modularized enough to be improved by other software engineers without knowing the domain knowledge? maybe
- Q: Are there sufficient unit tests for the developers? some
- Q: Are there benchmark data to test the new implementations? yes
- Q: Could you highlight the software engineering aspects of the project?
- Q: What is your advice for early-career software engineers who want to work on this project?
- **Q: If you want to redesign the package, what would you do?**
  - writing the core in C was probably good. Portability of C++ is now better. Even our conversion to C99 is not complete.
  - add unit tests, now they are distributed in local Testfile and "make testXXX"
  - better autoconf and/or cmake based install built from the ground up

  - integration in other than bash, eg. python or other high level scripting (julia?)

- a number of peculiar data (e.g. rotcur.5), and benchmark (CPU wise, but also "24+1" body)
- Q: What are lessons learned to keep this code going?
- Q: How many users are using it? There is probably a small core group, and there is good competition from python based environments (yt, amuse, galpy, gala, pynbody). In philosphy. AMUSE comes closest to NEMO.
- Q: How many citations are there? This is the sad story about legacy software. Only recently ADS and ASCL make it possible to track software.
- **Some innovative aspects of NEMO perhaps not seen widely used:**
  - connect code with papers via an ADS bibcode. See our bibcode table
  - help= vs. man and **checkpars**
  - loadobj: body, potential, . . .
  - scattered Testfile to pick up tests for "make check" (like a Makefile)
  - scattered Benchfile to pick up benchmarks for "make bench10" (like a Makefile)
  - the use of the **bsf** tool to check reproducability to N digits.
  - *surely there must be more*

CHAPTER SEVENTEEN

TODO LIST

The following TODO list has been automatically assembled :

Todo: figure out the right py-gnuplot

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/astronemo/checkouts/latest/docs/source/amuse.rst, line 81.)

Todo: figure out the right py-gnuplot

(The original entry is located in amuse.rst, line 81.)

Todo: this list needs to be annotated and spiced up with links.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/astronemo/checkouts/latest/docs/source/codes.rst, line 26.)

Todo: examples/Images

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/astronemo/checkouts/latest/docs/source/examples.rst, line 490.)

Todo: examples/Tables

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/astronemo/checkouts/latest/docs/source/examples.rst, line 496.)

Todo: examples/Potential

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/astronemo/checkouts/latest/docs/source/examples.rst, line 502.)

Todo: examples/Exchanging Data

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/astronemo/checkouts/latest/docs/source/examples.rst, line 513.)

Todo: describe the newer **accelerations** from *falcON*

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/astronemo/checkouts/latest/docs/source/potname.rst, line 28.)

Todo: potentials auto-build from source code ???

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/astronemo/checkouts/latest/docs/source/potname.rst, line 30.)

Todo: This draft is a work in progress, and started on April 27. It is an updated version of the old (latex) NEMO Users and Programmers Guide. We hope to have this converted sometime in this Summer (2021) when version 4.3 is blessed. Not all sections from the old manual were taken, but we also added some new sections.

Other online entry points for NEMO are: github pages and github code . We also keep an index to (unix) man pages for all programs.

Although this manual should be on https://astronemo.readthedocs.io we also keep a local copy readthedocs on astroumd.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/astronemo/checkouts/latest/docs/source/index.rst, line 13.)

CHAPTER

EIGHTEEN

RST REMINDERS

18.1 Lists

- Bullet are made like this
- A link to the NEMO repo, or github pages.
- Links to the man pages: github or local
- Links to some old examples: github or local
- **Links to some Restructuredtext RST guides:**
  - https://sphinx-tutorial.readthedocs.io/step-1/
  - https://www.writethedocs.org/guide/writing/reStructuredText/
  - https://thomas-cokelaer.info/tutorials/sphinx/rest_syntax.html
- **Point levels must be consistent**
  - **Sub-bullets**
    * Sub-sub-bullets
- Lists

Term Definition for term

Term2 Definition for term 2

List of Things item1 - these are 'field lists' not bulleted lists item2 item 3

Something single item

Someitem single item

18.2 Code blocks

There are three equivalents: `code`, `sourcecode`, and `code-block`.

```python
# some python code
import os
print(help(os))
if True:
   print("yes")
```

(continues on next page)

(continued from previous page)

```
else:
   print("no")
```

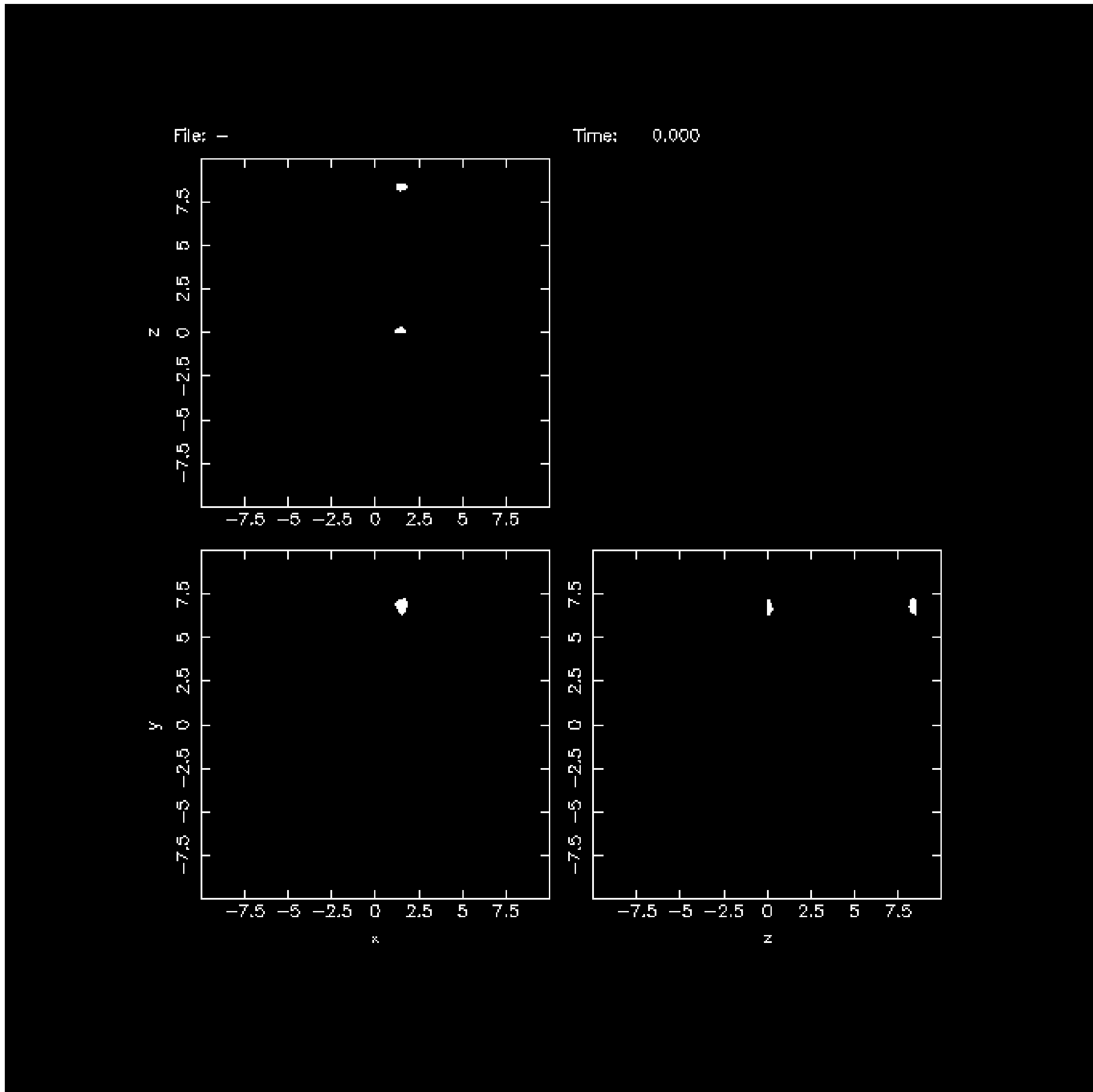
```
# Equivalent
```

```
# Equivalent
```

```
# some bash code
if [ -e /tmp ]; then
   echo "You have /tmp"
fi
```

18.3 Images

Here is an image …



from the eagle example.

18.4 Tables

Time	Number	Value
12:00	42	2
23:00	23	4

18.4.1 Math

Some example math, taken from latex

CHAPTER

NINETEEN

INDICES AND TABLES

- modindex
- search

INDEX

A

accelleration, 87
ACS, 84
agama, 84
AMIGA, 84
AMUSE, 84
arepo, 84

B

bhint, 84
bodytrans, 87
bonsai, 84
brutus, 84

C

Catena, 84
ccd, 87
CGS, 84
ChaNGa, 84
clustertools, 84

D

DICE, 84

E

ECSV, 87

F

falcon, 87
Fewbody, 84
fie, 87
FITS, 87
fractal, 84
FWHM, 87

G

gadgetX, 84
Gala, 84
galaxy, 84
GalPot, 84
galpy, 84
GANDALF, 84
GENGA, 84
GIPSY, 87
Glnemo2, 84
GraviDy, 84
gsf, 84
gyrfalcON, 84

H

hermite, 84
HiGPUs, 84
history, 87
hnbody, 85
HUAYNO, 85
Hydra, 85

I

ICICLE, 85
identikit, 85
image, 87
InitialConditions, 85

J

JSPAM, 85

L

limepy, 85

M

MARTINI, 85
mcluster, 85
McScatter, 85
mercury, 85
miriad, 87
MYRIAD, 85

N

N-MODY, 85
nbody6tt, 85
nbody6xx, 85
nbodykit, 85
nbodyX, 85

nemesis, 85
NEMO, 85
NIGO, 85

O

octgrav, 85
orbit, 87

P

partiview, 85
PENTACLE, 85
petar, 85
pixel, 87
plumix, 85
pNbody, 85
potential, 87
program keyword, 87
pycola, 85
pyfalcon, 85
pynbody, 85

Q

QYMSYM, 85

R

Raga, 85
RAMSES, 85
rebound, 86
review, 87

S

SecularMultiple, 86
set, 87
sidm-nbody, 86
slimplectic, 86
smalln, 86
smile, 86
snapshot, 87
SpaceHub, 86
SpheCow, 86
Starlab, 86
structured file, 88
Swarm-NG, 86
system keyword, 88

T

table, 88
Torch, 86
TPI, 86

U

UNSIO, 86

V

VINE, 86

Y

yapp, 88
yt, 86

Z

ZENO, 86