NEMO
USERS and PROGRAMMERS GUIDE

P. J. Teuben
Astronomy Department
University of Maryland
College Park, MD 20742

Version 3.2-MD
Spring 2004
Last document revised: 11th April 2004 by PJT

NEMO is a Stellar Dynamics Toolbox ...

This document presents the main features of NEMO, and describes how to obtain, install, and use it.

Trademark Acknowledgments

Contents

List of Tables

List of Figures

DISCLAIMER/NO WARRANTY

Because the present version of NEMO is passed around free of charge, we provide absolutely no warranty whatsoever.

You are of course free to pass this package on to your friends and enemies, but we really prefer to receive a mailing address of/from that new user. This is not only to keep us informed of any new users, but also to enable to send out information about updates.

While we very much appreciate receiving bug reports[1], presently we do not have anyone officially assigned to the task of maintaining NEMO. Therefore we cannot guarantee speedy reply.

If the above has not been enough of a disclaimer, let us say that this version of NEMO is still a preliminary version of what might once become a "real" system with the usual support facilities. We strongly discourage usage of this system if you are not in regular contact with the distributor(s), simply because we feel that the system hasn't been sufficiently tested yet. At the same time, we want to make NEMO available as soon as we can, so, voilà: all these disclaimers.

The name NEMO: originally we had set up directories owned by "`nobody`", however it turned out that this is already a name with a predetermined functionality in UNIX. Subsequently the latin analog, *nemo*, was adopted[2]. Not meant to be used as an acronym, we owe the following one to Luis Aguilar: *Not Everyone Must Observe.* Another nice one is hidden in this manual, which we leave as an exercise to the reader.

[1] `nemo@astro.umd.edu`

[2] It's greek analog is KEITAO, Zeno is a NEMO descendant

README

In order to prevent disappointment and frustration this section should be carefully read and understood before you endeavor into NEMO.

You can use the list below as a checklist, it will help you to decide if NEMO is really the right way for you to solve your particular problem. In the end it may be helpful to have the manual pages *programs(8NEMO)* and/or *index(1NEMO)* for names of specific programs and utilities that you may want to use. The items in the list below are in approximate order of importance, the most important ones listed first. The further down the list you come, the easier it will be for you to get along with the NEMO package.

- understand NEMO is nothing more than an extension of the UNIX environment, and that you know how to modify your default UNIX environment to be able to run NEMO programs (See Appendix A). This generally makes it relatively easy to use NEMO in other environments and packages.
- understand the basic user interface: programs have a '**keyword=value**' argument structure, and that there are '**program**' and '**system**' keywords. System keywords can generally be set fixed by an equivalent environment variable (Chapter 2, Appendix B)
- realize NEMO is mainly an N-body package, with utilities to create stellar systems, evolving them, and a large variety of analysis and display programs. However, for a number of problems an N-body system can be effectively and efficiently used to simulate a completely different physical situation. There are interfaces to create/convert data in/to image format, and export them in FITS format (Chapter 5). A small amount of orbit and table manipulation utilities are also available. (See also Chapter 3).
- understand the basic workings of a UNIX shell; how to write shell scripts in which programs are combined in a modular way, and to automate procedures. For this reason NEMO programs are also geared towards batch usage. (see any introductory UNIX book and Appendix C)
- understand that data is stored in binary format and in a general hierarchical/ structured file format. The data can always be viewed in human

readable form with the program *tsf(1NEMO)*. Depending on the kind of data, the format is structured in a specific way, e.g. snapshots, images, orbits. Only problem specific programs can read such datafiles (See Chapter 3).

- you know about the graphics interface `yapp` (see also Chapter 4). Make sure you understand the local NEMO implementation how to get graphics output to for example screen and printer. Sometimes they are hidden in one device driver, and the **yapp=** system keyword is then used to select the device (*e.g.* the *mongo* and *pgplot* yapp drivers), or programs are linked with different device drivers. In this case, the program name will have a underscore and the device name appended, such as *snapplot, snapplot_cg* and *snapplot_ps*. (See Chapter 4).

- realize that NEMO can be extended to suit your own needs. It is fairly easy to write new programs or even define a new data structure (as was done for snapshots, images and orbits). Do realize however that the main language used is C and support for linking FORTRAN and C is minimal and system dependant. (See Chapter 6).

- you realize what it means to install NEMO if this has not been done yet. You need to carefully read Appendix G in detail; any experience you have had with the UNIX shell, and utilities like "*make*" and "*autoconf*" will be useful.

More information

The manual that you are reading here is in LaTeX format and should be in `$NEMO/text/manuals`[3]. Other information on NEMO can be found in the following places:

- manual pages, in `$NEMO/man` and below, using the UNIX *man* or NEMO *nemoman*/*manlaser* commands. For a graphical interface programs like *tkman* and *gman* can be very useful (hypertext) browsers of manual page. We also automatically maintain a fully hyperlinked version formatted in html.

- The `README` file in NEMO's home directory will always contain "last minute" information that has perhaps not found its way into this manual yet.

- The file `faq.tex` in `$NEMO/text` lists *Frequently Asked Questions*, and possibly other files in that directory.

- A summary sheet `summary.tex` in `$NEMO/text` overviews NEMO's usage.

[3]The full NEMO directory structure is outlined in Appendix D

- `survival.texi`, a short texinfo introduction with basic survival techniques to get around.
- The NEMO home page, currently at `http://www.astro.umd.edu/nemo/`, contains many pointers to a variety of helpful pages, and sometimes contradicts information written down in this manual.
- The ManyBody compendium, putting together an introduction to NEMO, starlab and other N-body software, originally written for the first N-body School (Strasbourg, 2004).

ACKNOWLEDGMENTS

Without any doubt Josh Barnes deserves **the** credit for the work that lies in front of you. Most of the visible parts, such as the user interface and the filestructure, were originally developed by Josh Barnes, in collaboration with Piet Hut and Peter Teuben at the Institute for Advanced Study during the 1986/87 academic year. At the initial stages of this project we have also benefitted from contributions by Lyman Hurd, Eric Roberts, Mark Ross, and Panayotis Skordis. You will also find many ideas originally developed in NEMO in derived and evolved products such as Starlab and ZENO.

Of course any package will only improve when it is used. Numerous people, in particular the University of Maryland graduate students and Stefano Casertano have suffered through several stages of testing. Without them and their complaints we would never be where we are now. Needless to say, we need more complaints! We would also like to express our sincere thanks to Sverre Aarseth, for his hard work on the industry standard series "**NBODY***", which has also become a part of NEMO, and Lars Hernquist, for providing a fortran version of the treecode. Mark Bellon has contributed his optimized C version of treecode which specifically runs well on some parallel processors. The `GIPSY` and `MIRIAD` programming groups are also continuesly supplying us with software. Being one of the early guinea pigs of the C version of Numerical Recipes we have also benefitted from some code from Numerical Recipes Software. Finally, the `Usenet` public domain has proven to be of great value, and you will find several utilities in *e.g.* `$NEMO/src/tools` and `$NEMO/usr/lib`. There are too many to be all acknowledged at this place, the manual pages and/or source code will contain appropriate references.

P.T.

College Park, Maryland
November ,1991
August, 1994 April, 2004

Conventions used in this Manual

The following typographical conventions are used in this manual:

Text in *italic*, such as *image(3NEMO)*, mean a reference to a UNIX manual page. In this case the command

```
% man 3 image
```

would bring up the manual page `$NEMO/man/man3/image.3`. In some case (like actually this one), the `3` can be left out. See also *man(1)*.

Text in `verbatim` are used to display the contents of source files, or sample interactive sessions. The latter come in a few categories, where the system prompt denotes the system you're on. Currently you may see samples from a UNIX shell (csh), where the command is preceded by the percent (`%`) symbol, and the bourne shell (sh) where the dollar (`$`) is used[4]:

```
% ls

$ ls
```

Text in **boldface** is used to denote UNIX environment variables.

[4] on a rare occasion you may even find a VMS DCL example in this case

Part I

General Introduction and Concepts

Chapter 1

Introduction

NEMO is a collection of programs, running under a standard UNIX shell[1], 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. In addition, a common user interface is defined with which the user communicates with a program. User interfaces will be described in much more detail in the next chapter and Appendix B.

In order to run NEMO programs, your UNIX environment has to be modified slightly. This is normally done in the form of a few additions to a startup file (`.cshrc` or `.login` if you use the C-shell). Appendix A gives a full description.

Let us first give you an overview of the various "groups" of programs, as they clearly show the structure of NEMO to a first time user:

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, 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, and extract an orbit from it.

The *image group* is defined by a common file structure of "images" , *i.e.* two

[1] We will assume some basic knowledge of the UNIX operating system

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 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/package. 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 in NEMO.

More groups and intermediate 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 Chapter 6 we will detail some 'rules' how to incorporate/add new software into the package, and extend your NEMO environment.

The remaining chapters of this first part of the manual outline various concepts that you will find necessary to work with NEMO. Chapter 2 outlines the user interface (commandline, shells etc.), details are deferred to Appendix B. Chapter 3 explains how data is stored on disk and can be manipulated, including the concept of function descriptors in NEMO. Chapter 4 details how data can be graphically displayed, either using NEMO itself or external programs.

The second part of the manual is a cookbook: Chapter 5 gives a variety of examples of use.

The third part of the manual is the programmers manual: Chapter 6 is for the more adventurous user who wants to modify or extend NEMO.

The last part of the manual are Appendices with a large variety of reference information.

Chapter 2

The User Interface

A NEMO program is invoked just as any other application program under the operating system. Of course you must have modified your shell environment (see Appendix A on how to modify your account).

In the first section the keyword interface is explained. Subsequently, we will explain some of the more advanced concepts of this user interface, which can be skipped on first reading without loosing any essentials. Some of these advanced features may not even be available in your local implementation. The last section discusses the overall documentation system in NEMO, and how to get different types of help. Appendix B serves as a reference guide to the various user interfaces.

2.1 Keywords

2.1.1 Program Keywords

The most basic user interface is formed by the command line interface. Every NEMO program accepts input through a list of so-called **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[1]:

```
1% hackcode1 out=r001.dat
```

[1] from here on the % will denote the prompt of the operating system, anything after that on that line you type - subsequent lines without a prompt are output from the program. The number preceding is an identification number.

```
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

            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 very slow, abort the program with `<control>-C` and re-run it with fewer particles:

```
....
<Control>-C
REVIEW called, enter your command:
REVIEW|hackcode1> quit
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. First of all, an interrupt might have thrown you into the REVIEW section: the `quit` command is needed to get you back to the parent shell. This REVIEW section may not have been compiled into your user interface, in which case not to worry. The second peculiarity is shown by the line starting with “`###`”. It

Figure 2.1: Initial conditions for hackcode1

is generated by the fatal error routine, which immediately[2] aborts the program with a message to the terminal, even if the normal output was diverted[3] 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 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 2.1.

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:[4]

[2] There are ways around this, see the `error=` keyword described later in this section

[3] Technically, it was written to the standard error output channel, commonly called *stderr* in UNIX

[4] Your local VERSION will probably look a little different.

```
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. In summary, the system keywords are:

- 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.
- 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.
- The `yapp=` keyword lets you (re)define the graphics output device. Usually no default.
- 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[5] Default is 0.
- 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).

For a more detailed description of the system keywords and all their options see Appendix B. 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)*.

[5] bypassing existence of an output file is a very common use

2.2 Interrupt to the REVIEW section

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).

Depending on how the user interface on your system has been compiled, NEMO programs can be interrupted[6] to go into the REVIEW section during, or even optionally at the start of the execution of the program. The program pauses here for user interaction.

The `REVIEW>>` prompt appears and the user can interact with the program and reset keywords. The program can also be continued or gracefully aborted, and other programs can be run in the mean time. In Appendix B.3 an overview of all the commands and their options are given in more detail.

It should be remarked though that the program must be written in a certain way that resetting the value of the keyword also affects the actual flow of the program. Although this is always true for the system keywords (`help, yapp, debug` etc.), it is not guaranteed[7] for the program defined keywords (the ones you see when the `help=` keyword is used). The documentation should explain how to handle such situations, however in most current situations modifying a program keyword will not affect the flow of the program. A good example would be a program that iterates, and is given a new tolerance criterion or new initial conditions.

The REVIEW section is mostly useful to interrupt a quiet program that seems to take to long, and increase the `debug` level.

2.3 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

[6] UNIX programs can be interrupted with (control-backslash)

[7] In fact, this is hardly anywhere the case

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 **??** and B.4 There was also a Khoros (cantata, under khoros V1) interface[8] 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.[9]

2.4 Help

The HELP system in NEMO is manyfold.

- Inline help, using the `help=` system keyword, is available for each program. Since this is compiled into the program, you can copy a program to another system, without all the NEMO ssystem support, and still have a little bit of help.

- 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:

  - good old UNIX *man(1)*, but make sure the **MANPATH** environment variable includes the `$NEMO/man` directory. The `manlaser` script can print out the manual pages in a decent form.
  - The X-windows utility *xman(1)* provides a point-and-click interface, and also has a decent *whatis* interface. No hypertext though, but very fast since it directly interprets the *cat* files.
  - 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.
  - Under GNOME the `gman` formats tool has nice browsing capabilities.
  - The html formatted manual pages. Has limited form of hypertext, but contains the links to general UNIX manual pages, if properly addressed. Since installation is a bit tricky[10], the home base (`http://www.astro.umd.edu/nemo/ is your best bet to start surfing).`

[8] See also `http://www.khoral.com` for their new release

[9] It is envisioned NEMO will - perhaps via a SWIG, or-like, environment - support such an environment

[10] really: not documented

- This manual, the *The NEMO User and Programmers Guide*, contains information on a wide level, aimed at beginners as well as advanced users. The manual is also available in html.

Chapter 3

File structure

This chapter gives an overview of the file structure of persistent data[1]. Most of the data handled by NEMO is in the form of a specially designed[2] XML-like binary format, although exceptions like ASCII files/tables will also be discussed. Ample examples illustrate creation, manipulation and data transfer. We close this chapter with a few examples of function descriptions, a dataformat that make use of the native object file format of your operating system (*a.out(5)* and *dlopen(3)*).

3.1 Binary Structured Files

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, or ASF)

[1]Programmers are always free to choose any format they like in memory - this is usally hidden from the users. What we mean here is the disk format. The popular memory (object) models, and how they interact with persistent data on disk, are discussed in Chapter 6

[2]note this was back in 1986, well before XML was conceived

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 5.6). [3]

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.

[3] Note however that currently NEMO's binary files have some limited support for machine-independancies, e.g. simple endian swap. Some portability notes about this can be found in Appendix J.1.2

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 remark about online help: NEMO also uses the *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]
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 of course. 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 5.6.1.

3.3 History of Data Reduction

Most programs[4] 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`. [5]

[4] notable exceptions are basic programs like `tsf`, `rsf`, `csf` and `hisf`

[5] HISTORIC NOTE: To prevent data files having a history written into them an environment variable **HISTORY** must be set to 0. This dates back from older times when not all programs could properly handle data files with embedded history items properly. Also note there is no associated system keyword with **HISTORY**. It is expected that this feature will disappear, *i.e.* history is always forcefully written into the data files, unless the user interface (`getparam.o` in `libnemo.a`) was explicitly compiled with the HISTORY disabled.

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. *Note: the* `tab=` *keyword hints at the existence of such features.*

A usefull (public domain) program *redir(1NEMO)* has been included in NEMO[6] to be able split the two standard UNIX output channels *stdout* and *stderr* to separate files.

```
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 shell (`/bin/sh`) 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.

[6] see also the `tpipe` tool

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[7]). 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. See Section 5.4 for examples.

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 5.1.3, a table of the precompiled ones are in Table 5.1.

3.5.3 Nonlinear Least Squares Fitting Functions

The program `tabnllsqfit` can fit (linear or non-linear, depending on the parameters) a function to a set of datapoints from an ASCII table. The keyword

[7] Normally called `potname=, potpars=` and `potfile=`, but see also `rotcurves`

`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` is the only program that uses these functions, the manual page has a lot more details.

Chapter 4

Graphics and Image Display

NEMO programs also need to display their data of course. We shall 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. To display *image* data we rely mostly (but see `ccdplot`) on external software. Often images would need to be copied to a FITS file for this (but see `nds9`).

4.1 The YAPP graphics interface

The programs in NEMO which use graphics are rather simple and 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 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. 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 `_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 `snapplot` (general Sun screen within suntools), `snapplot_ps` (postscript), `snapplot_cg` (color Sun screen) and `snapplot_sv` (variable size sunview pixwindow for storing small images for movies). Again: actual names may differ on your system.

If programs are linked with the multiplexing libraries *yapp_mongo* or *yapp_pgplot* interface, several device drivers are transparently present through mongo, and the hidden system keyword `yapp=` is then used to select a device (a default can be set by using the **YAPP** environment variable). See also Appendix B.

However, despite these grim sounding words, we currently almost exclusively use the PGPLOT implementation of yapp.

4.2 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(1)*, *tabhist(1)*, and other packages such as *mongo(1L)*, *sm(1L)* and *gnuplot(1L)*, but even by completely foreign packages such as xgobi, grace, and xgraphic. Binary tables need to be converted to ASCII of course.

4.3 Image Display Interface

Data in *image(5NEMO)* format can be transferred in *fits(5NEMO)* format and subsequently displayed and analyzed within 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(1L)*, 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):

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

Part II

Cookbook

Chapter 5

Examples

Now that we have a reasonable idea how NEMO is structured and used, we should be ready to go through some real exercises. Some of the examples below are short versions of shell scripts[1] available online in one of the directories (check `$NEMO/csh` and perhaps `$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.

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 Chapter 3 we already used `mkplummer` to create a Plummer model; 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 reasonable time. Adjust it to whatever you can afford on your CPU and test your patience (see Appendix E benchmarks).

```
1% mkommod in=$NEMODAT/k7isot.dat out=tmp1 nbody=100 seed=280158
```

[1] where applicable, the examples in this chapter are written in the C-shell language

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 (E=–1/4)[2] the models have to be rescaled using `snapscale`:

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

In the case that your user interface was not compiled with the **NEMOINP**[3] directive, the `vscale` expression has to be calculated by you, *i.e.* `vscale=0.707107`. Also 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 of course:

```
3% mkommod in=$NEMODAT/k7isot.dat out=tmp2 nbody=100 seed=130159
```

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. 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 5.1.

[2]Virial units are the preferred units, see also: Heggie & Mathieu, in: *The use of supercomputers in stellar dynamics* ed. Hut & McMillan, Springer 1987, pp.233

[3]This can be found out by using the program nemoinp(1NEMO) or `help=?`.

Figure 5.1: Initial conditions for the encounter as set up in this section

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 & Hut "hierarchical tree" algorithm[4]:

```
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` $= 1/\Delta t$, the softening length `eps` $= \epsilon$, 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

[4] see also their paper in: Nature, Vol. 324, pp 446 (1986).

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;

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

Depending on the actual graphics (*yapp*) interface of snapplot, 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 ($-0.2 < z\&\&z < 0.2$). 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[5]. The expressions must contain legal C expressions and depending on their nature must return a value in the context of the program. *E.g.*

[5] loadobj, the dynamic object loader, does not works on all UNIX implementations

xvar and yvar must return a real value, whereas 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 *bodytrans(3NEMO)* routine. The real variables x, y, z, vx, vy, vz are the cartesian phase-space coordinates, t the time, m the mass, phi the potential, ax,ay,az the cartesian acceleration and aux some auxiliary information. The integer variables are i, the index of the particle in the snapshot (0 being the first one in the usual C tradition) and key, another spare slot.

For convenience a number of expressions have already been pre-compiled (see also Table 5.1), *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 r and 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 bodytrans it is possible to save such compiled expressions object files under a new name.

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, btr_jz.o which can in future programs be referenced as expr=jz (whenever a real-valued bodytrans expression is required), *e.g.*

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

Alternatively, one can handcode a *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 btr_r2.c:

```
#include <bodytrans.h>

real btr_r2(b,t,i)
Body *b;
real t;
int  i;
{
```

Table 5.1: 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
ar	real	(x*ax+y*ay+z*az)/sqrt(x*x+y*y+z*z) or: $(\bar{\mathbf{r}}\cdot\bar{\mathbf{a}})/\|\bar{\mathbf{r}}\|$
aux	real	aux
ax	real	ax
ay	real	ay
az	real	az
etot	real	phi+0.5*(vx*vx+vy*vy+vz*vz) or: $\phi + \bar{\mathbf{v}}^2/2$
i	real	i
jtot	real	sqrt(sqr(x*vy-y*vx)+sqr(y*vz-z*vy)+sqr(z*vx-x*vz)) or: $\|\bar{\mathbf{r}}\times\bar{\mathbf{v}}\|$
key	real	key (see also *int* version above)
m	real	m
phi	real	phi
r	real	sqrt(x*x+y*y+z*z) or: $\|\bar{\mathbf{r}}\|$
t	real	t
v	real	sqrt(vx*vx+vy*vy+vz*vz) or: $\|\bar{\mathbf{v}}\|$
vr	real	(x*vx+y*vy+z*vz)/sqrt(x*x+y*y+z*z) or: $(\bar{\mathbf{r}}\cdot\bar{\mathbf{v}})/\|\bar{\mathbf{r}}\|$
vt	real	sqrt((vx*vx+vy*vy+vz*vz)-sqr(x*vx+y*vy+z*vz)/(x*x+y*y+z*z)) or: $\sqrt{(\bar{\mathbf{v}}^2\text{-}(\bar{\mathbf{r}}\cdot\bar{\mathbf{v}})^2/\|\bar{\mathbf{r}}\|^2)}$
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.

```
        return sqrt(x*x + y*y);
    }
```

By compiling this:

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

an object file `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 **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 `$NEMOOBJ/bodytrans`).

5.1.4 Movies

In the previous subsection we showed that the program `snapplot` displays snapshots of the system at selected times. When displaying such snapshots in rapid succession, the illusion of a movie can be obtained. Normally however, the time to search and read the data from disk, calculate the required *bodytrans* projection and display it is much too long to give this impression, except possibly for very small number of particles. Still one lacks basic frame manipulations.

Two solutions are offered:

snapplot, movie

First, the snapplot program has a `frame=` keyword, whose value is the filename (more properly the base of the filename) of the saved bitmap of the current image on the screen. The format of this bitmap is system and *yapp* interface dependent, *i.e.* it depends on which version of snapplot is used. In other words: make sure you use the right compiled version of snapplot, and check your local documentation. It's name may even be as obscure as `sp`, `snapplot_sv` or so, but it may also be hidden under the official `snapplot` name itself.

There should then exist a program which manipulates and plays the `frame` (-raster) files back at a high enough rate to call it a movie. This is normally achieved by putting the frames into memory, with preferably a memory bitmapped display device or a fast screen-loader. On a SUN workstation the local format used is *rasterfile(5)*, and public domain programs such as *movie(1NEMO)* and *movietool(1NEMO)* can be used to re-display and manipulate frame files. A disadvantage with `movie` is that one has to be outside of suntools, though this may well be worthwile, because of the large number of options in the menu of `movie`. With the in-house written clone of movie, called `movie_sv`, it is possible to display `frame` files from within suntools. The menu of `movie_sv` is not as sophisticated, and the number of frames which fit in memory is not as large, but the advantage of doing it from within suntools is sometimes more important. By choosing a smaller size of the rasterfile this last disadvantage can even be circumvented. `yapp=256` is the default. The normal procedure[6] to create frames and display them within suntools is:

```
17% sp snap.out frame=movie1
18% movie_sv frame=movie1
```

Combining the *bodytrans(3NEMO)* capabilities of `snapplot` with `movie`, it is also possible to look at the 3D structure of an N-body system. The program `makepath` and shell script utility `3dmovie` generate various 'flyby's or 'fly-around's to get an idea of the 3D structure of a system, though this is mainly intended for batch use, is cumbersome in interactive mode and better methods are available though (see below).

The current `movie` programs for SUN workstations can only handle black and white rasterfiles. *check if this is a bug or a feature. ** also some bug when running within openwindows ***

Another making movies: one can also use a fancy `snapplot_cg` version, and record image by image on a high quality recording device, or interface with a *genlock*[7] device, or make slides.

snap3dv, snapxyz, xyzview, xgobi

Another approach is to store the computed 2D or 3D coordinates (that you would normally view with `snapplot` in a special format, and use an interactive 3D viewer for this.

The first step would be to create that intermediate format. Depending on the viewer, and different format (or program) needs to be selected to do this. We have currently two programs available: `snap3dv`: writes a number of formats, all intended for exported outside of the NEMO environment (typically they are all ascii files), and `snapxyz`: writes an **xyzc** file (a regular NEMO structured file which you can view with `tsf`).

[6] This assumes during installation the `snapplot` program was compiled with **$YAPP_SV** and renamed as `sp`

[7] interface device between the video output of a image display device and a VCR recorder which allows frame-by-frame recording

As for the different ascii format, see the comments in the manual page of `snap3dv` which programs can be used.

As for the **xyzc** format: the program `xyzview` can be used to interactively view any 3D coordinates of a snapshot at different times. In addition, one can also store a 3D vector information to the 3D coordinates (could be velocity, or force field or magnetic field direction) of a point, and display these. `xyzview` can keep a limited number of point data in memory, and display them in rapid succession, in the view/zoom/velocity mode that can be changed dynamically. The viewing conditions can be stored in a file, which could be used to create offline movies in a mode described above.

`xgobi` is is versatile multi-variate data browsing and analysis tool which suits itself quite well to some interactive 3D viewing of N-body data.

```
snapprint in=snap.out times=10 | xgobi
```

You will need to obtain `xgobi` via independant means, it does not come with NEMO. A new version of xgobi is under development, see `http://www.ggobi.org/`.

5.1.5 Advanced Analysis

Determine position of the proper "dynamical center of mass" of a system, *e.g.* using the 50% most bound particles. For this, it's best to save the potentials of the particles during the integration. Most N-body integrators will have an option to let you save the potentials of the particles. In `hackcode1` one has to use the `options=mass,phase,phi` keyword to save this relevant information.

Here is an example of analyzing the resulting output dataset of the merger we ran earlier. Using `snapcenter` the snapshot is centered weighing each particle by the third power of the potential, followed by a computation of the lagrangian radii using `snapmradii`. The resulting table is piped into `tabmath` and the logarithm of all radii is computed (the first columns contains the time) after which the converted table is piped into tabplot.

```
set rad=0.01,0.05:0.95:0.05,0.99
set nrad=(‘nemoinp $rad | wc -w‘)
@ nrad++

snapcenter r001.dat - ’-phi*phi*phi’ report=f |\
    snapmradii - $rad |\
    tabmath - - %1,‘nemoinp 2:$nrad format="log(%%%.f)" separ=,‘ all |\
    tabplot - 1 2:$nrad line=1,1 yapp=2/xs
```

5.1.6 Generating models

Besides a variety of programs of the kind of `mkplummer`, `mkommod`, `mkexpdisk` etc., models can also be generated by calculating appropriate tables (containing a run of density, potential and radius) and feeding them into programs which translate such tables into a snapshot. An example of a program in the making is `anisot`.

An alternative way is to use a package such as `Mathematica` to integrate the differential equations. With the following example we leave this as an exercise to the reader:

```
<< NumericalMath/RungeKutta.m

rpsipsiprime = RungeKutta[{psiprime, -(2/r)psiprime + Exp[-psi]},
  {r, psi, psiprime}, {10^-5, 10^-10/6, 10^-5/3}, 50, 10^-8,
  MaximumStepSize->0.5];

rpsi = rpsipsiprime[[Table[n,{n,Length[rpsipsiprime]}],{1,2}]];

r = rpsi[[Table[n,{n,Length[rpsi]}],1]];

psi = rpsi[[Table[n,{n,Length[rpsi]}],2]];

rho = Exp[-psi];

rrhotranspose={r,rho};

rrho = Transpose[rrhotranspose];

logrlogrho = Map[Log, rrho, {2}];

c = Log[10.];

log10rlog10rho = logrlogrho / c;

shortrrho = Take[rrho,30];

PlotODESolution[shortrrho, 1, 2,
      AxesLabel-> {"r", "rho"},
      PlotLabel -> "Isothermal Sphere"]

PlotODESolution[log10rlog10rho, 1, 2,
      AxesLabel-> {"log r", "log rho"}]
```

5.1.7 Handling large datasets

On 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 default[8] a particle takes up about 100 bytes, which limits the size of a snapshot to about 50,000 on most current 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
```

5.2 Images

In this section examples are given how to create an image (from a snapshot), show a contour diagram, and export the image as a FITS file and use it within another package. Specific examples are also given how to read in that FITS file in AIPS and IRAF without having to deal with tapes. In case one really has to do the FITS I/O via tape, the standard UNIX program *dd(1)* can be used. See also *ccdfits(1NEMO)* for details how to do this.

Images in NEMO are currently completely stored (in memory as well as disk) as double precision floating point numbers, which limits programs to of order 1024 square maps, depending how much memory your machine has. There are also the *xyio(3NEMO)* routines which allow more atomic (line oriented) access to the image data, but programs generally don't use them.

[8]one can recompile NEMO in single precision and define `body.h` with less wastefull members

5.2.1 Making an image from a snapshot

The simplest way to grid a snapshot into an image is `snapccd`, but this program has been superseded by the much more powerful tool `snapgrid`, which is based on `snapplot`. An example:

```
19% snapgrid snap_in image_out xrange=-10:10 yrange=-10:10 \
    nx=128 ny=128 xvar=x yvar=z evar="m*m"
```

grids the X and Z coordinates of the snapshot `snap_in` to an $128 * 128$ image `image_out`. The range in gridded coordinates is from -10 to 10 in both X and Z, with pixel coordinates defined in the center of a cell. Note that the emissivity (`evar=m*m`) is given as the square of the mass, which could be applicable for ionized hydrogen gas when the mass in the snapshot would have been indicative of the gas density. In the case of stars or neutral hydrogen gas `evar=m` would have been more appropriate (which is actually the default).

Often it is then desirable to smooth the image to improve the signal to noise ratio; although the degree of smoothing depends on the average number of 'objects' per pixel. Example:

```
20% ccdsmooth image_in image_out gauss=0.3
```

smoothes the image to a circular beam with FWHM (Full Width Half Maximum) of 0.3 physical units. In the above gridding example this amounts to about 2 pixels.

5.2.2 Galactic and Extragalactic objects

`snapgrid` has a specific choice of defaults which would make observers of extragalactic objects, *i.e.* external observers from the positive Z axis, happy (*i.e.* `xvar=x yvar=y zvar=-vz`). However with the help of a few other tools in NEMO one can also make galactic observers happy.

First the extragalactic objects: A total intensity map is generated with the defaults arguments of snapgrid. Channel maps at specific velocities can be generated using `snapgrid zrange=zmin:zmax` or `zrange=zmean,zsig`, depending on the required velocity beam. A velocity map is also easy to generate: The raw zeroth and first order moment maps are saved (`snapgrid moment=0,1`) and smoothed (`ccdsmooth`) after which they can be divided (`ccdmath`) resulting in a velocity map. Position-velocity diagrams can be directly generated using `snapplot/grid xvar=x yvar=-vz`. If position-velocity maps need to be smoothed, remember that it may have to be done in two steps (independently in 'position' and 'velocity'), because the current version of ccdsmooth can only

Figure 5.2: Position-Velocity diagram of a galactic disk as seen by an internal observer

do circular gridded beams if the beam is two dimensional. A more detailed example is given in the next subsection.

`snaprotate, snapshift, snapscale` are helpful tools to project a galaxy before all these operations are performed.

For galactic objects: one must choose an internal viewpoint (x,y,z) and (vx,vy,vz) somewhere "inside" the object, and make this the new origin using `snapspin`.

A rotation (`snaprotate`) may also be necessary. Then use `snapplot/grid xvar=vr yvar='atan2(y,x)' evar='m/(x*x+y*y+z*z)'`, where the Y axis of the plot will be a longitude between $-\pi$ and π, and the X axis the radial velocity, to create the familiar position-velocity diagram. The `evar=` keyword is only needed in gridding the snapshot.

Pretty pictures can be obtained using `ccdplot`, which can combine a contour map, overlayed on a greyscale image.

5.2.3 Extragalactic velocity field

In this example we shall make a velocity field of a a particle representation of a disk galaxy, with stars on circular orbits in centrifugal balance with a fixed background potential.

For disk-stars on circular orbits the program `mkdisk` is useful[9]:

```
21% mkdisk out=disk1 potname=expdisk potpars=0,1,0.5 rmax=2 mass=1
```

In this the mass of the disk was set to non-zero, in order to assign a finite emission to each star later on. If you would plot it's configuration with a program like `snapplot`, you would see not only a more-or-less constant surface density but also that the disk is infinitely thin (`snapplot yvar=z`). Also the particles are indeed on circular orbits (`snapplot xvar=r yvar=vt`), and there is no velocity dispersion (`snapplot xvar=r yvar=vr`).

Viewed from the positive Z axis (the default with `snapplot xvar=x yvar=y`) we would see no radial velocities in the disk; in order to get a realistic looking velocity field, we would have to rotate the model around a line of nodes (say the X axis) using `snaprotate`:

```
22% snaprotate in=disk1 out=disk1.r theta=60 order=x
```

[9] Only relevant keywords are shown, remaining take their default value, use `help=` to see them

Figure 5.3: Velocity field of a galactic disk

To plot the radial velocity the program `snapplot` is used by assigning a different symbol (`psize=`) to different radial velocities:

```
23% snapplot in=disk1.r psize=0.1*vz
```

To simulate a true observation we shall use the program `snapgrid` to grid the discrete snapshot data (`x,y,z,vx,vy,vz`) from the file `disk1.r` onto a CCD-like device: a rectangular pixel array (matrix), with a value (brightness, velocity etc.) associated with each pixel. Since we are interested in the radial velocity field the zero-th and first order moment maps need to be obtained, and divided to get a radial velocity field:

$$\langle v \rangle = -\frac{\int I(z) V_z(z) dz}{\int I(z) dz} \tag{1}$$

Here $I(z)$ and $V_z(z)$ are the intensity and radial velocity along the line of sight. Note the extra $-$ sign, to conform to the astronomical convention that positive velocity means negative `vz` if viewed from the positive Z axis. In NEMO the denominator and numerator in eq. (1) are evaluated as follows:

```
24% snapgrid in=disk1.r out=mom0 moment=0 zvar=-vz evar=m
25% snapgrid in=disk1.r out=mom1 moment=1 zvar=-vz evar=m
```

Since the data will be noisy, it is best to smooth the data a bit. Smoothing must however be done before the maps are divided (*why?*). Since the default pixel size is 4/64=0.0625 a Gaussian beam with a FWHM of 0.15 is used to convolve the data. We would use the programs `ccdsmooth` and `ccdmath` in the following order:

```
26% ccdsmooth in=mom0 out=mom0s gauss=0.15
27% ccdsmooth in=mom1 out=mom1s gauss=0.15
28% ccdmath in=mom0s,mom1s out=disk1.vel fie=%2/%1
```

The final output file, `disk1.vel`, now contains the radial velocity field map at an inclination of 60^o. It can be displayed with programs like `ccdplot` and `ds`. `ccdplot` is a NEMO program, capable of plotting contours as well as greyscale (if given the right graphics device driver). `ds` is general purpose image display program and displays a map in color on a sun workstation (for this, `ds` must have been installed to understand the NEMO file format).

You can also convert the NEMO image file to a FITS file. A FITS file is a true astronomical standard, which can be read into any other image processing package (`ds` can also read FITS files) (`AIPS, IRAF, MIDAS, Miriad`). Creating it can be done as follows:

```
29% ccdfits in=velfie out=fits1
```

5.2.4 Integrated Color Maps

A true color map can be created from a snapshot by assigning a color to the particles in the snapshot. This can be done by gridding the snapshots twice, each one assigning the particles with a different emissivity. In the example below a snapshot is assigned an artificial radial color gradient. The particles are given a linearly increasing emissivity, according to their ranking in radius. First, we must make sure the snapshot is sorted in radius properly:

```
30% snapsort in=snap.dat out=tmp1 rank=r
```

and next the snapshot is gridded twice:

```
31% snapgrid in=tmp1 out=tmp1_1 evar=m
32% snapgrid in=tmp1 out=tmp1_2 evar='m*(i+1)'
```

This means the color at the center would be $-2.5log(I_1/I_2) = 0$, whereas at the edge the color would be $2.5log(nbody)$. The images are best viewed when smoothed, and then divided and taken the `log` of. The factor 2.5 is left out here, because the scaling is arbitrary:

```
33% ccdsmooth in=tmp1_1 out=tmp1_1s gauss=0.3
34% ccdsmooth in=tmp1_2 out=tmp1_2s gauss=0.3
35% ccdmath in=tmp1_1s,tmp1_2s out=color_ccd.dat fie='log(%2/%1)'
```

.... *more to come*

5.3 Tables

NEMO has a few programs that manipulate table files, although much more can be done with standard UNIX programs such as *awk(1)*. A few examples are given how they can be used together. Programs such as *mongo(1L)* can be used to display results. If you're in the possession of `sm`, most of the functionality of the table programs can be reproduced with `sm`. A quick and dirty plot can also be made with *tabplot(1NEMO)*.

The following example shows how *tabmath(1NEMO)* and *awk(1)* can do the same thing:

```
1% tabmath tab_in tab_out %1+%2
2% awk '{ print $0,$1+$2 }' tab_in > tab_out
```

One can also use the table programs in UNIX pipes, and use NEMO's feature of denoting a '-' (dash) as filename for standard input/output files:

```
    3% awk '{print $1,$3,$5}' tab_in | tabmath - tab_out \
                                "ifgt(%1,%2,sin(%3),cos(%3))"
```

Don't ask why such a complicated `ifgt` construct, it's just an example. See *nemofie(3NEMO)* for the syntax options of the third (`newcol=`) keyword of `tabmath`.

5.3.1 Making an image from a table

Any scatterdiagram can now easily be turned into an image by using the snapshot interface! This would be temporary solution until the need for this would turn into a program *tabccd(1NEMO)*. The example below also demonstrates how existing tools can be effectively combined to create a new tool!

```
#! /bin/csh -f
#         - transform table into image -
#       DEMO version: no bells and whistles
set in=$1               # infile (table)
set out=$2              # outfile (image)
set xcol=$3             # columns from table to use
set ycol=$4
set xrange=$5           # gridding area
set yrange=$6
set nx=$7               # number of pixels to use
set ny=$8
set sx=$9               # some smoothing
set sy=$10

set tmp=tmp$$           # temp name for intermediate results

#     convert table to ASCII "205" snapshot (see atos(1NEMO))
awk  "END {print NR}"                $in     > $tmp.1
echo "3"                                    >> $tmp.1
echo "0.0"                                  >> $tmp.1
awk '{print 1.0}'                    $in    >> $tmp.1
awk '{print $'$xcol',$'$ycol',0.0}'  $in    >> $tmp.1
awk '{print 0.0,0.0,0.0}'            $in    >> $tmp.1
#                       convert to snapshot
atos $tmp.1 $tmp.2
#                       convert to image
snapgrid $tmp.2 $tmp.3 xrange=$xrange yrange=$yrange nx=$nx ny=$ny zvar=vz
#                       smooth image a bit
ccdsmooth $tmp.3 $tmp.4 gauss=$sx dir=x
```

```
ccdsmooth $tmp.4 $out  gauss=$sy dir=y
#                       write a FITS file
ccdfits $out $out.fits
#                       clean up mess
rm -f $tmp.*
```

5.4 Potential

Programs which need an external potential (*e.g.* orbit integrators) can obtain these via the so-called potential descriptors. They are implemented in NEMO as loadable object files[10]. To the user interface this commonly appears as a set of three program keywords `potname=`, `potpars=` and `potfile=`; they signify the identifying name of the potential, its parameters and an associated filename. The last two are optional, since the potential may not need parameters or an associated file(s).

For example, the program `potlist` lists the value of the potential and forces at selected gridpoints:

```
% potlist potname=harmonic potpars=0,3,2 x=0:3:1 y=0:3:1 z=0:6:2 dr=0.001
x y z ax ay az phi phixx phiyy phizz rho dr time
0 0 0 -0 -0 -0 0 9 4 1 1.11408 0.001 0
1 1 2 -9 -4 -2 8.5 9 4 1 1.11408 0.001 0
2 2 4 -18 -8 -4 34 9 4 1 1.11408 0.001 0
3 3 6 -27 -12 -6 76.5 9 4 1 1.11408 0.001 0
```

The usage of the colon separated implied do-loop in the `x=`, `y=` and `z=` keywords assumes that *herinp(3NEMO)* has been implemented. `potlist` will also take first order derivatives of the force, to test Poissons equation. This specific potential, `harmonic`, happens to have 4 parameters, although only 3 were given in the example. The fourth one will take some default present in the descriptor. The first parameter of all potentials should be the pattern speed[11]. The second through fourth parameters are the ω_X, ω_Y and ω_Z harmonic coefficients resp., where the potential is given as:

$$\Phi(x,y,z) = \frac{1}{2}\omega_X^2 x^2 + \frac{1}{2}\omega_Y^2 y^2 + \frac{1}{2}\omega_Z^2 z^2$$

Although NEMO comes supplied with a small number of standard potential descriptors, it is relatively easy to make your own ones. In Section **??** we will

[10] Not all operating systems allow the programmer to use this feature - see your local loadobj implementation

[11] Although if you supply your own potential you could cheat and bypass this

describe how to add your own potential descriptors. Next we shall present a few examples from the standard list of available potentials, the full listing can be found in Appendix F.

5.4.1 A few potentials

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

Remember that most NEMO program use the keywords `potname=` for the identifying name, `potpars=` for a list of parameters as listed below and an optional `potfile=` for the few that have an associated datafile for the potential. 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`.

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

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

Ω_p Pattern Speed

M Total mass

r_c Core radius

5.4.2 How to build your own potential descriptors

Although this subject really is one that should be deferred to Chapter 6, we will now present a simple prototype "definition" for a potential descriptor in C and Fortran[12]

```
void inipotential (int *npar, double *par, char *name);
void potential (int *ndim, double *pos, double *acc, double *pot, double *time);

SUBROUTINE INIPOTENTIAL(NPAR, PAR, NAME)
SUBROUTINE POTENTIAL(NDIM, POS, ACC, POT, TIME)
```

As you can and will see more of, a potential descriptor is in origin really a C or FORTRAN source code file, that needs two (FORTRAN) subroutines or

[12] FORTRAN is not supported on all architectures

Figure 5.4: Plummer potential properties

(C) functions with the callable names `inipotential` and `potential`. Their arguments must conform to the specification given above. Because we do want to allow Fortran source code as well, all arguments are called by reference in C.

Programs which need a potential descriptor will automatically compile your source code (if needed) and load the object code into the program for usage. The repository of standard NEMO potential descriptors (as object files) lives in `$NEMOOBJ/potential`, and is automatically searched when the environment variable **POTPATH** is appropriately set. Note that the two subroutines themselves are not called directly by the user, but by a workhorse routine from the standard NEMO library. This hides much of the interface for the programmer. More details on this technique can be found in Chapter 6 (*still to come*).

Below is a fully commented listing of the `harmonic` potential, as an example of such a potential descriptor given in the C language. It, and other potentials, can be found in source code form in the directory `$NEMO/src/orbit/potential/data`

```
/*
 * harmonic.c: procedures for initializing and calculating
 *             the forces and potential of a harmonic potential
 */
#include <stdinc.h>                 /* formal NEMO include file */
local double omega = 0.0;       /* defined but not used in here */
local double h[3] = {1.0,1.0,1.0};        /* default parameters */
/*-----------------------------------------------------------------
 * INIPOTENTIAL: initializes the potential.
 *      input: npar, the number of parameters
 *             par[] an array of npar parameters
 *      If npar=0 defaults are taken (remember to initialize them
 *      as static (local) variables in this file)
 *-----------------------------------------------------------------
 */
void inipotential (int *npar, double *par, string name)
{
    int i;

    if (*npar>0)
        omega = par[0];
    for (i=1; i<(*npar); i++)
       h[i-1] = sqr(par[i]);
    if (*npar > 4)
        warning("Only 4 parameters used in Harmonic Potential");

    dprintf (1,"INI_POTENTIAL Harmonic Potential\n");
    dprintf (1,"  Parameters : Pattern Speed = %f\n",omega);
    dprintf (1,"  wx^2,wy^2,wz^2= %f %f %f\n",h[0],h[1],h[2]);
}
/*-----------------------------------------------------------------
 *  POTENTIAL: the worker routine. Determines at any given point
 *      (x,y,z) the forces and potential.
 *      Note that this routine is good for 1, 2 as well as 3D
 *-----------------------------------------------------------------
 */
void potential (int ndim,double *pos,double *acc,double *pot,double *time)
```

```
{
    int    i;

    *pot = 0.0;
    for (i=0; i<*ndim; i++) {
        (*pot) += h[i]*sqr(pos[i]);
        acc[i] = -h[i]*pos[i];
    }
    *pot *= 0.5;
}
```

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 *orbit(5NEMO)*), and a close conceptual relationship exists between a (single-particle type) **snapshot** and an **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 5.4. Many examples of the tricky `potpars=` keyword are given in Appendix F.

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 `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 orbits[13]

  ```
  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, `otos` turns an orbit into a snapshot, and may come in handy since the snapshot package has far more advanced analysis programs.

[13]For the careful reader: `mkplummer` and `potname=plummer` actually have different units, and as such this experiment is not properly set up.

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 `potname=, potpars=, potfile=` keywords can be given, if the input orbit contains...

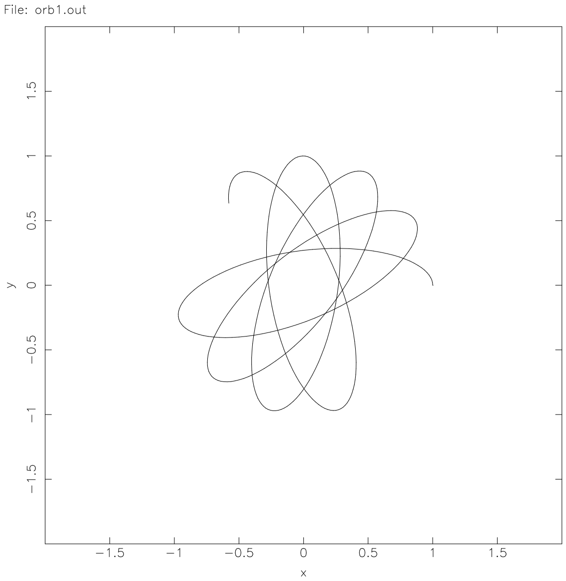

Figure 5.5: Sample orbit 1 (`orb1.out`)

- **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 method[14]. This program has however not been released to the public version of NEMO.

5.5.3 Display

- **orbplot** is the only orbit plotting program we currently have. For more sophisticated display `tabplot` and/or `snapplot` would have to be used after transforming the data. Also `snapplot` uses the powerful *bodytrans* expression parser to plot arbitrary body related expressions, although `orbplot`

[14]see also van Albada & Sanders, (1982, MNRAS, 201, 303)

can handle both `x, y, z` and `vx, vy, vz` for the `xvar=` and `yvar=` keywords. An example of the output of `orbplot` is given in Figure 5.5.

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

```
% 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

% 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.

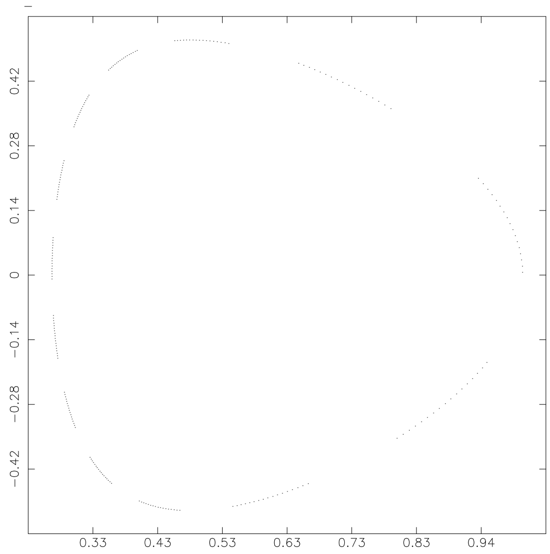

Figure 5.6: Surface of Section for sample orbit 1 (`orb1.long`)

- **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[15].

- **otos** transforms an orbit back into a snapshot, thereby giving you the much richer set of analysis tools that are available for *snapshot*'s.

[15] Carnevali, P. & Santangelo, P., 1984. ApJ 281 473-476

5.6 Exchanging data

The exchange of (binary) data between machines of different architecture is often a painful process. For NEMO binary structured files, we have devised a general portable way to port files between machines, even if both have different low level file formats (*e.g.* SUN OS and the Cray UNICOS OS). A different solution has been used by the MIRIAD package, which writes it's data always in the same (IEEE) format. This means a different layer of translation routines is needed for certain architectures, notably VMS and Unicos. A similar mechanism is expected to be used in NEMO in some future release.

In case the other machine has a totally different file format, it's handy to have the data in simple ASCII table format. NEMO also allows import and export of N-body data through an ASCII format described in `atos` It can be used directly for multiple-snapshot data, but example shell script are available to transport data.

In the case of N-body data there is no standard format to store the particle information, and we are subject to someone's favorite format. We will encounter a few, and show examples how to convert them under NEMO. In the case of images, there happens to be an astronomical standard: FITS[16], we will discuss a few applications here too. Tables can also be transferred in an extended form of the FITS format[17], although here the ASCII format may do equally well. Even N-body snapshots can be written in FITS format, for an example see the toy program `snapfits` which uses the now deprecated FITS Random Group Format.

5.6.1 NEMO data files in general

Here is a neat trick to exchange NEMO data files between systems of different binary file format. On machine 1 the data is saved in (UNIX) compressed ASCII format:

```
m1% tsf r.dat maxprec=t allline=t | compress > r.data.Z
```

The data can then be transported to machine 2 (in binary mode of course if data was compressed), and saved in the local binary structured file format:

```
m2% zcat r.data.Z | rsf in=- out=r.dat
```

It turns out that for most data files the compressed ASCII file in full precision is about as large as the original binary file. The example above also shows that,

[16] See: Wells et al. (1981), A&A Suppl. 44, 363.
[17] See: Harten et al. (1988), A&A Suppl. 73, 365.

by using pipes, machine 1 and 2 never need to store the full ASCII version of the file, which will in general be about 4 times as large as the binary file(s). Note again that a dash filename is interpreted as standard input/output in the NEMO environment (see also *stropen(3NEMO)*), but one should be warned here that some older versions of structured files could not be used in pipes.

For machines which support I/O redirection in the ftp program, an even more efficient solution is possible by redirecting the (compressed) data from the other machine into local binary structured format:

```
m2% ftp m1
ftp> binary
ftp> get r.data.Z "| zcat | rsf - r.dat"
```

The compressed ASCII data never needs to be stored on the local disk directly. The data is uncompressed and passed to `rsf` through a pipe.

5.6.2 Snapshot Data

To import a snapshot into NEMO format one can use `atos` or write the data in this ASCII (also referred to as the "205") format. In particular a snapshot which is already in table format with masses, positions and velocities in columns 1,2-4,5-7, can be converted to snapshot format using a simple C-shell. For example,

```
% table_to_snapshot tab_file snap_file
```

with the following simplified version of the `table_to_snapshot` C-shell script (without any bells and whistles) using `awk` (`tabmath` could have been used similarly):

```
#! /bin/csh -f
#       table_to_snapshot: demo version
set infile=$1           # input table (m,x,y,z,vx,vy,vz)
set outfile=$2          # output snapshot
set tmpfile=tmp$$       # a temporary scratch name

awk "END {print NR}"   $infile   > $tmpfile
echo "3"                        >> $tmpfile
echo "0.0"                      >> $tmpfile
awk '{print $1}'       $infile  >> $tmpfile
awk '{print $2,$3,$4}' $infile  >> $tmpfile
awk '{print $5,$6,$7}' $infile  >> $tmpfile

atos $tmpfile $outfile
rm $tmpfile
```

The full version of this script can be found in in `$NEMO/csh`.

5.6.3 Image Data

For images the situation is a little better because there exists a standard in the astronomical community: the FITS format (see also *fits(5NEMO)*). `ccdfits` convert a NEMO image to standard FITS diskfile, which can be read and manipulated by various image processing packages. The reverse program, `fitsccd`, is also available, and can convert most FITS images into NEMO's *image(5NEMO)* format. When importing FITS images into NEMO, always be concerned with the units, since NEMO insists that the origin be at (0,0,0).

AIPS

In **AIPS** the following can be done:

Suppose your fits file is stored in a directory, which we will call **$dir** (*e.g.* `set dir=/usr/nemo/fits`) and the filename is FITS.DAT (most filenames MUST be in upper case in AIPS), then the AIPS task `IMLOD` can be used to read the fits file (or from tape, see below):

```
% setenv XX $dir        # make sure this is set for AIPS
% aips                  # login/start up AIPS
...                     # (some more login stuff here)
> TASK 'IMLOD'          # set task name
> INFILE 'XX:FITS.DAT'  # set up input FITS filename
> INNAME 'TEST'         # set some name for output AIPS file
> GO                    # run the task
> MCAT                  # check if file TEST there
> GETN ...              # get map number for file TEST
> TVLOD                 # load it on tv
> TVFIDDLE              # change contrast
> EXIT                  # quit AIPS
%
```

Using the tape-interface is a bit more cumbersome: dump the FITS.DAT file to a tape, using *dd(1)* with a block size of 2880 bytes[18], and have `IMLOD` read the data from tape. This in case the disk interface will not work. Even on DEC-VMS machines the direct disk fits file may be used (this has done been successfully in the GIPSY package - see *ccdfits(1NEMO))*.

The reverse process can also be used to write AIPS files to disk in FITS format using the task `FITTP`, as show in the following example:

[18] Newer versions of AIPS now allow you to use a blocking factor which writes blocks in multiples of 2880 bytes; e.g. a blocking factor of 10 needs block size 28800 bytes.

```
% setenv XX $dir        # make sure this is set for AIPS
% aips                  # login/start up AIPS
...                     # (some more stuff here)
> MCAT                  # check directory
> TASK 'FITTP'          # set task name
> INNAME 'TEST'         # set name of input AIPS file
> OUTFILE 'XX:FITS.DAT' # filename for output FITS file
> GO                    # run the task
> EXIT                  # quit AIPS
%
```

The file will then be in `$dir/FITS.DAT`, make sure that the program `aips` has write permission in that directory. Again, if the disk interface does not work, the file has to be dumped to tape, and read to disk using *dd(1)*. For an example see *ccdfits(1NEMO)*.

IRAF

IRAF is normally started up by issuing the `cl` command (you may need a `login.cl` startup file in your current or home directory). Converting an existing file in IRAF format into a FITS is very simple, as is illustrated in the following example:

```
% cl                            # startup IRAF
cl> dataio                      # go into the dataio area
da> wfits iraf_file fits_file   # and convert it
da> logout                      # quit IRAF
%
```

The parameter `bitpix` may have to be set to 16 or 32, if you don't like it's default. The complementary IRAF program `rfits` converts a FITS file into IRAF format.

MIRIAD

MIRIAD can currently not read or write FITS images with `BITPIX=8`.

IDL

MIDAS

In this section we will some examples of NEMO's image format. The manual pages *image(5NEMO)*) describes the data format, whereas *image(3NEMO)*) introduces image I/O library routines.

5.6.4 Initializing

There are a few programs with which orbits 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]
```

```
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
```

5.6.5 Galactic Velocity Fields

As an example, a special section is devoted here to the analysis of galactic velocity fields. 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
```

```
% set r=$(B!F(Bnemoinp 0:60$(B!F(B
% set v=$(B!F(Bnemoinp 0:60 | tabmath - - "100*%1/(20+%1)" all$(B!F(B
% ccdvel out=map1.vel rad="$r" vrot="$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=map.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
```

Part III

Programmers Guide

Chapter 6

Introduction

In this chapter an introduction[1] is given how to write programs within the NEMO environment.

To the application programmer NEMO consists of a set of 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 the better.

After reviewing how the NEMO environment should be present, Section 6.2 describes some of the available macro packages. How an example program is written, compiled and used in NEMO is shown in Section 6.3. In Section 6.5 we will show how you can write programs in C++, and still use the NEMO libraries. Finally, Section 6.6 deals with those people who insist on using FORTRAN,

6.1 The NEMO Programming Environment

The modifications necessary to your UNIX environment in order to access NEMO are extensively described in Appendix A. This not only applies to a user, but also to the application programmer, although for the latter the **static** environment is to be preferred here. Relevant environment variables are described in Appendix H.2

In summary, the essential changes to your environment consist of three additions to your local `.cshrc` startup file (or its equivalent if you don't use the `csh`-shell):

[1] Based on an original report *"NEMO: Elementary Mechanics Observatory"* by Joshua Barnes

1. set the environment variable **NEMO** to the location of the root directory of NEMO, and the **NEMOHOST** environment variable.

2. source the startup file `$NEMO/NEMORC`.

3. add `$NEMOBIN` to your search path in a convenient location (but before the system directories where the C compiler is located).

Although the environment variable **NEMOHOST** is used to allow the package to be shared across a number of different architectures (*e.g.* Sun3's and Sun4's), it is recommended to use this environment variable in a single-architecture environment

Once the NEMO environment is merged into your UNIX environment, most programs can be compiled as follows:

```
cc -o try try.c -lnemo -lm
```

6.2 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 <stdinc.h>
#include <snapshot/body.h>
#include <snapshot/get_snap.c>
```

Some of the macro packages are merely function prototypes, to facilitate modern C compilers, and have associated object code in libraries in `$NEMOLIB` and programs need to be linked with the appropriate ones.

6.2.1 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 `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`. There is potential trouble when NULL has been set to `(void *)0`, `'\0'` is OK though. Example on IBM's AIX operating system.

- `bool:` typedef for `short int` or `char`, used to specify boolean data. See also next item. [2].

- `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.

- `local, permanent:` macros for `static`. Use `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 `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, $\pi/2$ and $4\pi/3$, respectively.

- `ABS(x), SGN(x):` macros for absolute value and sign of `x`, irrespective of the type of `x`.. Beware of side effects.

- `MAX(x,y), MIN(x,y):` macros for the maximum and minimum of `x,y`, irrespective of the type of `x,y`. 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 quirks. (see section 6.2.5 below)

[2] The *curses* library also defines `bool`, and this made us change from `short int` to `char`

6.2.2 getparam.h

The command line syntax described earlier in Chapter 2 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 in the next subsection.

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 <stdinc.h>      /*   every NEMO module needs this  */
#include <getparam.h>    /* needed when user interface used */

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 earlier releases, no help string is available.[3] 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`[4] as the first step in your MAIN program:

```
main (argc, argv)
int argc;
string argv[];
```

[3] ZENO uses a different technique: ...

[4] It secretly assumes that `argv[]` is NULL terminated, which is not guaranteed on all UNIX implementations

```
{
    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:

```
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. *stdio* etc.). Within NEMO we have taken this one step further. The example given later in Section 6.3.3 also uses the technique of calling the main entry point `nemo_main()`.

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[5]:

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

There is a whole family of `getXparam()` functions which parse[6] 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>     /* defines ‘‘int getiparam()’’ */
. . .
int nbody;
```

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

[6]Depending on compiler switches at installation the getXparam parsing includes full expressions

```
. . .
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)*.

6.2.3 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);
```

In the first call the number of elements to be parsed from an input keyword `y=` is limited to `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 `y[]` is `nyret`.

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

Both routines return negative error return codes, see *nemoinp(3NEMO)*.

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

6.2.4 Alternatives to nemo_main

It is not required for your program to define with `nemo_main()`. There are cases where the user needs more control. An example of this is the `falcON` N-body code in `$NEMO/use/dehnen/falcON`. A header file (see e.g. `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
  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()
```

6.2.5 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. 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 *items* and *sets*. This definition is recursive, so fully hierarchical file structures are allowed, and indeed encouraged. Every set or item has a name *tag* associated with it, used to label the contents of a file and to retrieve objects from a set. Data items have a *type* and array *dimension* 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 `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 *stdio(3)* streams to access input and output files; hence the first step in using filestruct is to open the file streams. For this job we use the NEMO library routine `stropen()`, which itself is not part of filestruct. `stropen(`*name,mode*`)` is much like `fopen()` of *stdio*, but slightly more clever; it will not open an existing file for output, unless the *mode* string is `"w!"`. An additional oddity to `stropen` is that it treats the dash filename `"-"`, as standard in/output,[7] and `"s"` as a scratch file. Since *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 `strclose()`. This 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. Scratch files are automatically deleted from disk when they are closed.

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 `stdinc.h` conventions). And now suppose that, after some computation, results have been stored in the first `nbody` components of the `mass` array, and a descriptive message has been placed in `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 `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 `mass.dat` would already have existed, `error()` would have been called inside `stropen()` and aborted the program. Names of tags are arbitrary, but we encourage you to use descriptive names,

[7] Older versions of *filestruct* cannot handle binary files in pipes, since filestruct uses fseek(3)

although an arbitrary maximum of 64 is enforced by chopping any incoming string.

The resulting contents of `mass.dat` can be viewed with the `tsf` utility:

```
% 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 `010=8` of `Nbody`.

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");

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 `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 `get_data_coerced()`, which is called exactly like `get_data()`, but interconverts `float` and `double` values[8].

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);
```

[8]The implementors of NEMO will not be held responsible for any loss of precision resulting from the use of this feature.

Note that each `put_set()` must be matched by an equivalent `put_tes()`. For input, corresponding routines `get_set()` and `get_tes()` are used. These also introduce a significant additional functionality: between a `get_set()` and `get_tes()`, the data items of the set may be read in any order[9], or not even read at all. For example, the following is also a legal way to access the `NotASnapShot`[10]:

```
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 `bool` routine `get_tag_ok()` can be used to control the flow of the program, as `get_set()` would call `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");

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);
```

[9] tagnames must now be unique within an item, as in a C `struct`

[10] This is a toy model, shown for its simplicity. The full `SnapShot` format is discussed in Section 6.2.7

```
        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 `instr` is not a `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 `skip_item()` is only effective, or for that matter required, when doing input at the top level, *i.e.* not between a `get_set()` and matching `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.

!Some lyrics on new random access features in filestruct will go in here!

6.2.6 vectmath.h

The `vectmath.h` macro package 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 **NDIM**. If this is not supplied, the presence of macros **TWODIM** and **THREEDIM** will be checked, in which case **NDIM** is set to 2 or 3 respectively. The default of **NDIM** when all of the above are absent, is 3. Of course, the macro **NDIM** must be provided before `vectmath.h` is included to have any effect. Resetting the value of **NDIM** after that, if your compiler would allow it anyhow without an explicit `#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, *e.g.* `absv()` for the length of a vector.

Operations such as **SETV** (copying a vector) are properly defined for every dimension, but **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. *Numerical Recipes in C* by *Press et al.* (MIT Press, 1988).

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);
}
```

some more examples here - taken from some snap code

6.2.7 snapshots: get_snap.c and put_snap.c

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 `Body` structures; A `Body` has components accessed by macros such as `Mass` for the mass, `Pos` and `Vel` for the position and velocity vectors, *etc.*. Since `get_snap.c` and `put_snap.c` use only these macros to declare and access bodies, they can be used with any suitable `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 `snapshot/body.h` and `snaphot/snapshot.h`.

A program which should handle a large number of particles, may decide to include a more simple `Body` structure, as is *e.g.* provided by the `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 `Body` from the standard `snapshot/body.h` macro file. This last one contains `Mass, PhaseSpace, Phi, Acc, Aux` and `Key`.

In the example listed under Table 6.1 the first snapshot of an input file is copied to an output file.

Notice that the first argument of `stropen()`, the filename, is directly obtained from the user interface. The input file is opened for reading, and the output file for writing. Some history[11] is obtained from the input file (would we not

[11] See next section for more details on history processing

have done this, and the input file would have contained history, a subsequent `get_snap()` call would have failed to find the snapshot), and the first snapshot is read into an array of bodies, pointed to by `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.

6.2.8 history.h

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

When a NEMO program is invoked, the program name and command line arguments are saved by the `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 `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);
```

Private comments should be added with the `app_history()` [12] 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 `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 `put_history()` was only called before the for-loop. It is only a defensive call: `get_snap()` would fail since it expects only pure `SnapShot` sets (in effect, it

[12] The old name, `add_history` was already used by the GNU *readline* library

calls `get_set(instr,"SnapShot")` first, and would call `error()` if no snapshot encountered).

6.3 Building NEMO programs

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 writing the documentation and possibly a Makefile, the former one preferably in the form of standard UNIX manual pages (*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
or:
    % cc -g -o snapprint snapprint -lnemo -lm
```

The second form would only work if your *cc* compiler understands the **-L** switch, and the `$NEMOBIN/cc` has installed this feature. See Appendix G how to properly install this script.

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

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

The `bake` and `mknemo` 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:[13]

```
% mknemo snapprint
```

6.3.1 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.

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:

NAME the name of the beast.

SYNOPSIS command line format or function prototype, include files needed etc.

DESCRIPTION maybe a few lines of what it does, or not does.

PARAMETERS description of parameters, their meaning and default values. This usually applies to programs only.

EXAMPLES (*) in case non-trivial, but recommended anyhow

DEBUG (*) at what `debug` levels what output appears.

SEE ALSO (*) references to similar functions, more info

BUGS (*) one prefers not to have this of course

TIMING (*) performance, dependence on parameters if non-trivial

STORAGE (*) storage requirements - mostly of importance when programs allocate memory dynamically, or when applicable for the programmer.

LIMITATIONS (*) does it have any obvious limitations?

AUTHOR who wrote it (a little credit is in its place) and/or who is responsible.

FILES (*) in case non-trivial

HISTORY date, version numbers, why updated, by whom (when created)

[13] this assumes you have some appropriate NEMO permissions

6.3.2 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 basically three types of Makefiles in NEMO:

1. The first (top) level Makefile. It lives in NEMO's root directory (normally referred to as `$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.

2. Second level Makefiles, currently in `$NEMO/src` and `$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.

3. Third level Makefiles live in source or user directories `$NEMO/src/topic` and `$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 `$NEMO`. The second kind of entry point are the 'nemo' targets and never called by you, the user; they are only called by Makefiles one directory level up from within `$NEMO` below during the rebuilding process of NEMO, *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.

> This means that user Makefiles **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 6.3.3.

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

6.3.3 An example NEMO program

Under Table 6.2 below you can find a listing of a very minimal NEMO program, "`hello.c`":

and a corresponding example Makefile to install by *user* and *nemo* could look like the one shown under Table 6.3

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

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 GNU make facility.

6.4 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 `nbody0` program to NEMO, and we assume him to have his original stuff in a directory ~`nbody0`.

1: Create a new directory, `"cd $NEMO/usr ; mkdir aarseth"` and inform the system manager of NEMO that a new user should be added to the user list in `$NEMO/usr/Makefile`. You can also do it yourself if the file is writable by you.

2: Create working subdirectories in your new user directory, `"cd aarseth ; mkdir nbody0"`.

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 (`'nbody0'` in this case): `"cp ../pjt/Makefile .  ; emacs Makefile"`.

4: Go 'home' and install, `"cd ~nbody0 ; make install"`, assuming the Makefile there has the proper install targets. Check the target Makefile in the directory `$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. `$NEMO/usr/aarseth/Makefile`) contains the nemo_lib, nemo_bin and nemo_src install targets. An example of adding a foreign package that way is the `GRAVSIM` package , which has it's own internal structure. In the directory tree starting at `$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.

6.5 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 `main()` be in C++. For this you have to link with the NEMO++ library **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:

```
% cppc -g -o test test.cc -lnemo++ -lnemo -lm
```

6.6 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 the `nemo_main()` function (see Section 6.3.3). 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 `nbody0` and `nbody2`. In both cases data file I/O is done in C in NEMO's *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 `$NEMOINC/fortran`.

Again this remark: the *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 `snapwork(n,m,pos,vel,...)`.

More description will follow

6.6.1 Calling NEMO C routines from FORTRAN

The NEMO user interface, with limited capabilities, is also available to FORTRAN programmers. First of all, the keywords, their defaults and a help string must be made available (see Section **??**). This can be done by supplying them as comments in the FORTRAN source code, as is show in the following example listed under Table 6.4

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

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

after which the new `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 `f77` command, instead of `cc`:

```
% f77 -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:

```
% cc -c test_main.c
% f77 -o test test.f test_main.o -L$NEMOLIB -lnemo
```

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

The other alternative would be:

```
% f77 -c test.f
% cc -o test test.o test_main.c -L$NEMOLIB -lnemo \
           -lF77 -lI77 -lU77 -lm
```

with various possible complications with the new FORTRAN 1.3+ compiler on SUN workstations. Browsing with *nm(1)* through UNIX library files to find undefined reference might be the only alternative left to find out where the system has hidden them. See the `FORLIBS` environment variable defined in `NEMORC` startup file.

6.6.2 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.

6.7 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 `dprintf` or `error` function calls:

- During runtime you can set the value for the `debug=` (or use the equivalent `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 `error=` (or use the equivalent `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 `error` by 1.

Table 6.1: $NEMO/src/tutor/snap/snapfirst.c

```
#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[] = {
    "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);
}
```

Table 6.2: $NEMO/src/tutor/hello/hello.c

```
#include <stdinc.h>             /* standard (NEMO) definitions */
#include <getparam.h>                        /* user interface */

string defv[] = {       /* standard keywords and default values */
    "verbose=true\n            Verbosity level (t|f)",  /* key1 */
    "VERSION=1.2\n             25-may-92 PJT",          /* key2 */
    NULL,               /* standard terminator of defv[] vector */
};

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

nemo_main ()              /* standard start of any NEMO program */
{
    bool verbose;                  /* declaration of local var. */

    verbose = getbparam("verbose");         /* get that keyword */
    printf("Hello NEMO!\n");                /* do some work ... */
    if (verbose)                            /* and perhaps more */
        printf("Bye then.\n");
}
```

Table 6.3: Sample makefile - cf. $NEMOLIB/Makefile

```
# 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
```

Table 6.4: $NEMO/src/kernel/fortran/test.f

```
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
```

Chapter 7

References

"The Unix C Shell Field Guide" - (Anderson, G. and Anderson, P., Prentice Hall, 1988).

"A Hierarchical O(N log N) Force-Calculation" - Barnes, J.E. and Hut, P. (Nature, Vol. 324, pp 446 1986).

The FITS tables extension. - Harten R.H., Grosbol, P., Greisen, E.W. and Wells, D.C. (A&A Suppl. 73, 365, 1988)

"Hierarchical N-body Methods" - L. Hernquist, (Computer Physics Communications, Vol. 48, p. 107, 1988.)

"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.)

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

"The Art of N-body Building" J.A. Sellwood - (Ann. Rev. Astron. Astrophys. Vol. 25, pp. 151 1987).

"FITS – " - (Wells et al., A&A Suppl. 44, 363. 1981)

"Effective Fortran" - Metcalf. Oxford: Clarendon Press (1985).

"Numerical Recepies" - Press et. al. ...

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

Part IV

Appendices

Appendix A

Setting Up Your Account

This Appendix describe how you have to modify your UNIX environment in order to use NEMO. It requires nothing more than a few additions to your standard `.cshrc` startup file. We do however differentiate here between a **static** and **dynamic** setup.

Overall installation of the NEMO package will be discussed separately, in Appendix G and requires some knowledge of the UNIX operating system.

A.1 Static Setup

A **static** setup provides NEMO automatically every time you log in and may consist of up to four modifications to the `.cshrc` file:

(1) The environment variable **NEMO**[1] should be defined to tell NEMO where its root directory is. This environment variable is used to derive many subsequent references to locations of data files, documentation, program binaries, libraries etc. Add the following line to your `.cshrc` file before any other references to NEMO are made, *e.g.*:

```
setenv NEMO /usr/nemo
```

(or whatever your local root directory for NEMO is).

(2) The optional environment variable **NEMOHOST** should be set to refer to the HOSTTYPE you want to run. If not present, NEMO will attempt to resolve it. Some UNIX shells define an environment variable HOSTTYPE, which could also be used.

[1]The environment variable **NEMOPATH** is from V1 is now invalid

(3) Add the statement

```
source $NEMO/NEMORC
```

to your `.cshrc` file.

(3) Your search path should include `$NEMOBIN`, preferably in the beginning of the path definition. The reason for this specific location is that we often use a slightly modified *cc(1)* compiler (script), *e.g.*, you may then have something like

```
set path=(. $NEMOBIN /bin /usr/bin /usr/local/bin )
```

in your `.cshrc` file.

If you do this the very first time, make these modifications permanent for your current terminal session, *e.g.*:

```
% source .cshrc ; rehash ; echo $PATH
```

A.2 Dynamic Setup

A **dynamic** setup provides NEMO only after a startup load command, usually dubbed `nemo`, issued during an interactive terminal session. A script `$NEMO/nemo.rc` is available for this purpose. The following two modification are then necessary to your `.cshrc` file, instead of the above described procedure:

```
setenv NEMO /usr/nemo
alias nemo 'source $NEMO/nemo.rc'
```

Note the single quotes, to allow so-called late evaluation. Whenever the command "`nemo`" is issued, the NEMO environment is loaded with whatever the current value of $NEMO is. After this has been done, the `nemo` alias is replaced by another one to prevent re-entry. The alias `omen` will unload NEMO from the environment.

A.3 Tailoring

If you create a file `.nemorc` in your home directory, it will be read after the standard `$NEMO/NEMORC` (or `nemo.rc`) file, and this is where you would want to add your private NEMO additions, or even override the things you don't want.

Appendix B

User Interface

This Appendix overviews the basic command-line interface of NEMO programs. Front-ends, such as `mirtool` and the `miriad` shell are also described here.

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

B.1 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.

B.2 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), and are:

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.* `"setenv 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 **NEMODEF**[1] is set. The keyword files have the name *"progname"*.**def**, *e.g.* `snapshot.def`[2]. 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 *e.g.* the case in AIPS; you would have to use the `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 **EDITOR** can be used to set any other editor than good old *vi(1)*. It is not possible to combine this level with UNIX I/O redirection.

8,16,... although not processed, it is reserved for the next levels of menu interface.

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 `AIPS` 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 of the `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.

a list all arguments in the form *keyword=value*.

[1] mirtool also uses this environment variable

[2] This may result in long filenames, Unix SYS5 allows only 14 characters - a different solution is needed here

p,k list parameters (keywords) of all arguments in the form *keyword*.

d,v list defaults (values) of all arguments in the form *value*.

n add a newline to every *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 of `miriad`[3]. Is mainly intended to be used by scripts such as `mktool`. The procedure in NEMO to update a `.doc` file would be:

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

if the script `mktool` has been installed[4]

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.

host= Runs the program on a remote host. It depends on the implementation of software on local as well as remote host if and how this option works. Among SUN systems the `rsh` command is used, and assumes a shared disk with the same absolute pathname (NFS). Future implementations will have to use more sophisticated RPC (Remote Procedure Call) or X11 interfaces for distributed networking. No environment variable is used here.

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 for system experts usage only. 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 *dbx(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.

---

[3] Both `mirtool` and `miriad` need such a doc-file to lookup keywords and supply help

[4] Obviously this is priviliges to NEMO superusers only

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.

B.2.1 Yapp_mongo

Valid devices in *yapp_mongo* are (see also MONGO-87[5] users manual) numbers `1..6, -6..-1`:

```
1.  Retrographics 640    -1.  Versatec (portrait)
2.  DEC VT125            -2.  Versatec (landscape)
3.  Tektronix 4010       -3.  Printronix (portrait)
4.  Grinell 270          -4.  Printronix (landscape)
5.  HP 2648A             -5.  Postscript (portrait)
6.  Sun Windows          -6.  Postscript (landscape)
                         -7.  Postscript (square portrait) ***
```

Make sure you have set:

```
setenv MONGOPATH /usr/local/lib/mongo
setenv MONGOFILES $MONGOPATH/mongofiles.dat
```

or appropriate private ones, or use whatever your system manager has provided for.

B.2.2 Yapp_sunview

For the *yapp_sv* (Sunview) interface the absolute value of the `yapp=` keyword gives the (square) size in pixels of the window created. The default is 128. For positive value the window will disappear when *plclose(3NEMO)* is called at the termination of the graphics operations (essential for movie production). For negative values it remains displayed until a Carriage Return is given.

B.2.3 Yapp_pgplot

A graphics device in PGPLOT[6] is defined by preceding it with a slash, in the same way as is done in PGPLOT itself. Optional parameters 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):

[5] MONGO is a copyrighted program by John Tonry

[6] PGPLOT is a copyrighted public domain graphics library written by Tim Pierson

```
?            Get a list of all currently defined graphics devices
x,y/sun      Sunview, (x,y) = sizes in inches [8,8]
file/ps      Landscape Postscript, file=filename [pgplot.ps]
file/vps     Portrait Postscript, file=filename [pgplot.ps]
/null        Null device
/xwindow     X-windows
/xdisp       pgdisp or figdisp server
/tek4010     Tektronix
/tk4100
/gterm       IRAF Gterm window within sunview
/retro       Retrographics
/file        PGPLOT Metafile
/gf          Graphon
/vt125
/printronix
/peritek
/ws4         Landscape
/tfile
/pk
```

BUGS: When a non-zero help level is used, one cannot specify system keywords, other than by specifying them through environment variables,

Also consult manual pages such as *getparam(3NEMO)* and *yapp(5NEMO)*.

B.2.4 Yapp_sm

A graphics device in *sm*[7] is defined by the same text string as you would issue it in interactive mode. The *sm* command `list devices` will list all currently compiled device drivers. You need a file `.sm` in your home directory, or take what your system manager has provided for.

[7] SM is a copyrighted program and subroutine library by Robert Lupton and Patricia Monger

B.3 The REVIEW section

By setting the **REVIEW** environment variable a NEMO program is always put into the REVIEW section just before the start of the actual execution of the program (the end of the *initparam(3NEMO)* routine). This functionality is quite similar to using the helplevel `help=4` (see previous Section).

A NEMO program can also be interrupted, using the quit signal (see *signal(2)*), into the REVIEW section, although the program must be adapted to get keyword information through *getparam(3NEMO)* and not through it's own local database, in order for modified keywords to take effect. This does not hold for the system keywords, whose new value is always correctly interpreted by the program.

In the REVIEW section the prompt is **"REVIEW"** and the following commands are understood:

exit, quit, end Exit the program (ungracefully).

stop Gracefully end the program, but first goes through `finiparam()` (see *getparam(3NEMO)*) to update the keyword file if the helplevel includes 1.

set [**key**=[**value**]] Set a new value for a program keyword (`set key=value`), where `value` may also be blank, or display the contents of a program keyword (`set key`).

show key Show the value of a program keyword.

keys Show the values of all program keywords.

syskeys Show the values of all system keywords.

set syskey[=**value**] Set a new value for a system keyword `set syskey=value` or display its current contents `set syskey`.

time Show the cputime (in minutes) used so far.

!cmd Pass a command `cmd` to the shell.

go,continue Continue execution of the program.

version Display version of `initparam()` compiled into program.

?, help Displays all commands and their format.

When the system keyword `debug` is non-zero, the "REVIEW" prompt also includes the process identification number of the process.

B.4 Miriad

The `miriad` front-end works on any simple terminal, and is not restricted to a particular operating system or computer type. To invoke[1] `miriad`, type:

```
> miriad
```

The usual system prompt will be replaced by the `Miriad%` prompt:

```
Miriad%
```

and `miriad` will read a file, `lastexit`, with the values of all parameters saved when you last exited `miriad` (see EXIT below). This file will be created/read in/from your current working directory.

We shall now describe all `miriad` commands in more detail. A summary of the commands, and their syntax, is given in Table B.1 at the end of this section. Commands not known to `miriad` are simply passed to your host operating system. This means on VMS the command `DIR`, and on UNIX the command `ls`, would still be usable and valid commands[2].

INP, GO, and TASK

To inspect the inputs of a task, as well as to select the task, *e.g.* `histo` type

```
Miriad% inp histo
```

`miriad` will show the parameters of the task along with the values, if any, previously set. For example, if, the first time you run `miriad`, you type:

```
Miriad% inp histo
```

`miriad` will reply by writing:

```
Task:   histo
in       =
region   =
range    =
nbin     =
```

and will replace the `Miriad%` prompt with a task prompt

[1] Optional command line switches are summarized in Table B.2
[2] UNIX aliases are not supported in `miriad`

```
histo%
```

indicating that you have chosen the task `histo`. You can also choose the task `histo` without using `inp` by typing:

```
Miriad% task histo
```

`miriad` will then replace the `Miriad%` prompt with the `histo%` prompt, but the inputs will not be printed out. In either way a parameter can be set by typing any of:

```
histo% in=gauss
histo% set in gauss
```

(the first form being preferred). In either case retyping:

```
histo% inp
```

will result in `miriad` replying:

```
Task:   histo
in       = gauss
region   =
range    =
nbin     =
```

Tasks are run either by typing `go` *taskname* at the `Miriad%` prompt (advisable only if you know you like the inputs) or by typing `go` at the *taskname%* prompt. Thus, in the above example, typing:

```
histo% go
```

would result in the task `histo` running with `in=gauss` as the only assigned paramater (all the other parameters were set to their default values).

Any task can be run regardless of the chosen task, merely by typing:

```
Miriad% go itemize
```

`miriad` then executes the task `itemize` (using whatever input parameters it finds from the `lastexit` file or a previous run) and changes the default task and prompt to:

```
itemize%
```

GOB

The `gob` command does exactly the same as the `go` command, except it runs the task in the background (UNIX), or spawns the task (VMS). In either way, `miriad` is immediately ready for new commands, although the output of the task just started may still return output to the terminal.

HELP and ?

When typed at the `Miriad%` prompt, the `help` command gives a list of all the available tasks. This feature is useful when you use `miriad` for the first time. When typed at the *taskname%* prompt, the help command gives information on the chosen task. As with other commands, typing

```
histo% help itemize
```

displays the help file for `itemize` but does not change the default taskname (*i.e.* `histo`). To display the list of available tasks from a *taskname %* prompt, type `help ?`.

The `?` command lists general help about the `miriad` interface, including most of the information contained in this section.

EXIT, END, QUIT

`exit` and `end` both exit `miriad`, and, if any parameter values have been changed from the previous time you ran `miriad`, all parameter values are saved in a file `lastexit`. The parameter values from this file are read in automatically when you next run `miriad`. `quit` exits `miriad` without saving the present parameter values in `lastexit`. This command is useful if you have changed parameter values you do not wish to save.

UNSET and RESET

It is often convenient to assign a parameter value to the default. This assignment is accomplished with the command `unset`. For example, if the task `histo` had been run with inputs:

```
in       = gauss
region   =
range    = 0,1
nbin     = 10
```

but you wanted to run histo with the default for nbin, you would type:

```
histo% unset nbin
```

Multiple parameters can be unset on the same line. Thus, to assign both `range` and `nbin` to their default values, you would type:

```
histo% unset range nbin
```

Then, typing:

```
histo% inp
```

will result in `miriad` replying

```
Task:   histo
in       = gauss
region   =
range    =
nbin     =
```

The command reset can be used to unset all parameter values (not just those of the present task selected); use with caution.

SAVE and LOAD

As noted above, when you exit `miriad`, all your inputs will be saved in the file `lastexit`. However, many parameters are used by more than one program and, at present, only one value of each parameter is allowed throughout `miriad` (global parameters). The command `save` writes the current parameter values to a default file (*taskname*`.def` if the task *taskname* has been selected) or to a user specified file. This default file (or a user specified file) can be read in using the command load. Note that `save` writes out **all** parameter values, not just those for the specific task chosen. Example:

```
histo% save
histo% save histo1.pars
```

In the first case the parameters are saved in a file `histo.def`, whereas the second case saves them in a user specified file `histo1.pars`.

VIEW

Rather than type in parameter assignments one at a time, it is possible to invoke a text editor (EDT in VMS, vi or the editor given by the **EDITOR** environment variable in UNIX) to edit the `taskname.def` file. As usual, if no taskname is given, view will edit the `taskname.def` file indicated by the `taskname%` prompt. If a taskname is specified, *e.g.*:

```
histo% view itemize
```

the existing file `itemize.def` is edited, and the selected task is changed to `itemize`. Note that if the `itemize.def` file did not exists, one is created.

CD

The `cd` (change directory) command[1] allows you to change your working directory from within `miriad`. Note that the file `lastexit` is written to your current working directory, but that you will exit `miriad` to the directory you started it from. You can always find out your current working directory with a local host command, *i.e.* `pwd` on UNIX and `SHO DEF` on VMS. *perhaps cd with no args should say current working dir*

VERSION

The `version` command displays the which version of `miriad` you are running, and how it was compiled[2] If you have a problem with `miriad`, especially one you have never seen before, check if the version of `miriad` has changed. If so, you may want to tell you local MIRIAD manager about the problem, if you suspect that it is related to the new version of `miriad`.

B.4.1 Command and command line switches overview

The two tables summarize all of `miriad`'s commands (Table B.1) and the optional command line switches (Table B.2). As an example if you want to keep all default files located globally accross the filesystem, independant of the project you're working on, you could create a subdirectory in you home directory, and use that when you startup miriad. A UNIX example:

```
% mkdir ~/def
```

[1] Note that the `miriad cd` command does not understand special UNIX symbols

[2] This latter information is useful only for the advanced user, so if you don't understand it, don't worry.

Table B.1: Miriad shell command overview

Command	Syntax	Comments
=	par = value	assignment (see also deprected SET)
!	!cmd par1 par2 ...	run a cmd in parent shell
?		help
ˆ C		attempt to interrupt current running task
alias	alias [name] [value]	set or show aliases
cd	cd directory	change directory
exit	exit	exit miriad, and save parameters in lastexit
go	go [taskname]	start up task
gob	gob [taskname]	start up task in background
help	help [taskname]	online help on task
inp	inp [taskname]	overview inputs
input	input cmdfile	read commands from a command file
load	load [keyfile]	load task parameters from keyfile
quit	quit	quit miriad, and does not save lastexit
reset	reset	reset all variables
save	save [keyfile]	save task parameters to keyfile
set	set par value	assigment (*deprecated*)
task	task taskname	set new default taskname
unalias	unalias al1 al2	unset aliases
unset	unset var1 var2	unset variables (blank them)
version	version	current version of miriad and capabilities
view	view [taskname]	edit task parameters in a keyfile

```
% miriad -d ~/def
```

Table B.2: Miriad shell command line switches

Switch	Environment Variable	Comments
-f lastexit_file	-	global keyword file [lastexit]
-b mirbin_dir	MIRBIN	executables
-d mirdef_dir	MIRDEF	default files
-p mirpdoc_dir	MIRPDOC	documentation files
-g		turn debugging on
	PAGER	pager for reading documents [more]
	MIRPAGER	document pre-formatter [doc]
?		help

Appendix C

Shell Scripts

Shell scripts can make life a lot easier. If you have never written them, and have to do a lot of repetitive work, this appendix is for you. It will give you a small tour through creating and using them, but for a more complete coverage proper literature is recommended.[1] In the second part of this appendix we will introduce you to an advanced usage of shell script, to make them look more like real NEMO programs with a "keyword=value" type interface.

C.1 C Shell Scripts

A shell script is a plain ASCII text file and can hence be created with any text editor. It is a list of commands which you would normally have issued yourself on the commandline (quite similar in concept to command files under VMS or batch files under MSDOS). In addition, most shells have the capability of command flow logic (`goto`, `if/then/else/endif` etc.) retrieving the command line parameters, defining and using (shell) variables etc.

Under the Unix environment one can also choose which shell to use, although we shall only give examples in the most commonly used shell, the C-shell (`csh`). As an example, we will show a shell script which copies all files from one directory to a new one, also creating that new directory. The new directory must not exist yet, otherwise the script will fail with an error message. It's usage would look like:

```
% csh -f scriptname dir1 dir2
```

[1] recommended: Anderson and Anderson, *The Unix C Shell Field Guide.* (Prentice Hall,1986)

or shorter:

```
% scriptname dir1 dir2
```

The second form, in which the script is not told through which shell it should process its commands, is the recommended practice. The first line of this script file must then contain the line

```
#! /bin/csh -f
```

to denote the script is to be run via the C-shell, which on standard Unix systems is located in `/bin/csh`. In this case the script needs to be made executable, *i.e.*

```
% chmod +x scriptname
```

In effect your operating system will then issue the first form of the command, The second form has the advantage that you don't have to remember which shell to use, and in the end saves a few keystrokes, always considered a big issue in Unix!

Now lets look at the full text of the script first:

```
#! /bin/csh -f
#
# Example of a shell script to copy all files from one directory
# to another. The input directory must not contain any subdirectories,
# and it will not copy any so-called (hidden) dot-files.
#
##                   check if called properly
if ($#argv != 2) then
    echo "Usage: $0 dir1 dir2"
    echo "copies all files from one directory to another"
    goto done
endif
##                  save command line args in variables
set dir1=$1
set dir2=$2
##                  check if dir1 indeed is an existing dir
if (! -d $dir1) then
    echo "$dir1 is not a directory" ; exit 1
endif
##                  check if dir2 does not exist
if (-e $dir2) then
    echo "$dir2 already exists" ; exit 1
endif
##                  create new dir2
mkdir $dir2
if ($status != 0) goto error
##                  loop through all files in dir1
foreach file ($dir1/*)
    if (-d $file) then
```

```
        echo "Skipping $file (is a directory)"
    else
        echo "Copying $file"
        cp $file $dir2
    endif
end
##          Labels to jump to exit OK (done) or not OK (error)
done:
 exit 0
error:
 exit 1
```

A few things can be noted:

- Comments are lines that start with a `#`, but the first line of the script must contain this strange construction `"#! /bin/csh"`[2] to tell it how to execute itself. By default, if the first line would not contain such a directive, the script would be executed by the (more primitive) Bourne shell (`/bin/sh`). Any options that would normally be supplied to the shell must then be given in the first-line shell directive, as with the `-f` option in the example above.
- Shell variables can be created using the `set name=value` construct; they are henceforth referenced by prepending the shell variable name with a dollar: `echo $name` would simply display the `value` of the `name` shell variable (`echo` is a UNIX command).
- Command line arguments are in special shell variables `0, 1, 2` etc. `$0` is the name of the script, `$1` the first argument (if present), etc. All command line arguments `$1, $2,..` are also pre-defined in an shell variable `argv`, which is actually a list (like an array). It can be referenced as `$argv[1], $argv[2], ...` etc. This form has the advantage that the construct `$#argv` (the same as `$*`) can be used to find the number of elements in the list. In our example we want exactly two, hence the first check. A shell variable list can be initialized using brackets, *e.g.*: `set name=(val1 val2 val3)`. In this case `$#name` is 3, `$name[2]` has the value `val2` etc.
- The `if (test) then/else/endif` is used to test a condition and control command flow. The `!` is used to negate a condition, and multiple conditions can be used together in the `or` (`!!`) and `and` (`&&`) boolean operators. The `-e` tests if the next argument is a file or directory that exists, and the `-d` if the next argument is a directory.
- The `foreach name (list)` construct takes all elements from the `list` that follows, and executes all commands until a matching `end` is encountered. Inside the loop the current shell variable is defined in `name`.

[2] Some older versions of UNIX do not understand this technique

- Labels are defined by a name, followed by a colon, `label:` jumping to a label is be done by using `goto label`.
- Scripts should by terminated properly by an `exit` command, optionally returning an error code (0=no error) to the caller. This gives the caller the opportunity to catch faulty behavior. After each command the `status` shell variable contains the `exit` value (`$status`) of that command.
- After each shell command the special shell variable `status` contains the exit status of the previous command. We have used it once to check of the `mkdir` command completed successfully, if not, a direct exit by jumping to the label `error` was provided for.

More examples of shell scripts in NEMO can be found in the directory `$NEMO/csh` (public scripts) and for the more adventurous in `$NEMO/src/scripts` (maintenance scripts).

C.2 Parseargs C-shell scripts

Writing user friendly shell scripts often requires a fair amount of administrative work, especially if the number of variables to the script is variable.

If the public domain program `parseargs` is available, that approach may used. Shell scripts created with this option have the syntax (using the example `plummer.csh` C-shell script below):

```
% plummer.csh [-n <nbody>] [-s <seed>] <file>
```

in the short notation[3] using single character options, or in the longer form:

```
% plummer.csh [+nbody=<nbody>] [+seed=seed>] <file>
```

In this longer form the difference between the normal NEMO user interface is that keywords have to be preceded with a `+` symbol, plus it is possible to create scripts with "parameters" without an associated keyword. Perhaps this last option should not be used (this example uses it in the `<file>` filename parameter) in order to keep in pace with core NEMO programs.

[3] like the UNIX command-line syntax

Example script `plummer.csh`:

```
#!/bin/csh -f
#
#  Example C-shell script, using parseargs, to create an Nbody Plummer model
#
set NAME="`basename $0`"

setenv ARGUMENTS "\
  '?', ARGHIDDEN, argUsage, NULL,    'Help {print usage and exit}', \
  'n', ARGOPT,    argInt,   nbody,   'Nbody {# bodies in disk}', \
  's', ARGOPT,    argInt,   seed,    'Seed {Random number seed}', \
  ' ', ARGREQ,    argStr,   file,    'File {Output Filename}', \
  ' ', ARGLIST,   argStr,   argv,    'argv {any remaining arguments}', \
  ENDOFARGS \
"
## set defaults ##
set nbody='1000'    ## default number of bodies in disk
set seed='0'        ## default inityial seed
set other=()        ## remaining parameter (ignored in here)

## parse command-line ##
parseargs -s csh -e ARGUMENTS -u -- "$NAME" $argv:q >/tmp/tmp$$
if ( $status != 0 ) then  ## improper syntax (or just wanted usage)
        rm -f /tmp/tmp$$
        exit 2
endif

## evaluate output from parseargs & remove temporary file
source /tmp/tmp$$
rm -f /tmp/tmp$$

## echo arguments ##
echo "ARGUMENTS:"
echo "=========="
echo file=$file:q
echo nbody=$nbody:q
echo seed=$seed:q
if ($#argv > 0) echo Additional Positional Parameters=$argv:q

## get down to the work
#
echo " ### Making initial conditions"
mkplummer out=$file nbody=$nbody seed=$seed
```

Appendix D

Directory Structure

In this Appendix we show the directory tree structure as you should find it under NEMO. Directory names preceded with a (*) are being fully exported by the export procedures. The "`src`" tree contains all of the official source code, and is expanded a bit on following pages. The "`usr`" (or "`contrib`") tree is what used to be the `$NEMO/src` tree in the first releases of NEMO (Version 1.x). Some of the code in `$NEMO/usr` is now outdated, and superseded by more recent versions somewhere in `$NEMO/src` tree. Any 'personal' (most public domain also) code should now be placed in the appropriate `$NEMO/usr` tree before it can be accepted (moved over) into the `$NEMO/src` tree. We advice programmers to stick as closely as possible to the install procedures in the `$NEMO/src` tree, if at all possible. See the Makefiles and the discussion on these in the Programmers Guide (see 6.3.2).

```
   adm/                       local administrativia
       export/                - exported tar files
       import/                - imported tar files from other sites
   BIN/                       executables (NEMOHOST)
 * bugs/                      ... the known ones (optional)
 * csh/                       example shell scripts
 * data/                      NEMO data files
 * demo/                      ... demo scripts
   etc/                       various /etc type files
 * inc/                       standard include dir for NEMO
       fortran/               -- fortran I/O routines
       multicode/             -- josh' multicode
       snapshot/              -- snapshot
   LIB/                       object libraries (NEMOHOST)
   local/                     things local to your site
 * man/                       manual pages
       man1/                  -- programs
       man3/                  -- library routines
       man5/                  -- file structure
       man8/                  -- miscellaneous/maintenance
       doc/                   -- inline .doc for miriad/nemotool
 * news/                      news on new/modified software
   OBJ/                       dynamic object loader files (NEMOHOST)
       bodytrans/
       potential/
 * src/                       The whole source tree: (see also next page)
       kernel/                 Basic general stuff (i/o, utils)
       nbody/                  Nbody (snapshot)
       image/                  Images
       orbit/                  Orbit calulations, potentials
       tools/                  Various borrowed tools
       scripts/                Various (maintenance) scripts
       tutorial/               Tutorial material - toy programs
       ...                    ==See next page for expanded view==
(*)usr/                       Public donation programs (old NEMO/src)
       josh/
       pjt/
       piet/
       ...
 * text/                      alternate doc, mostly (la)tex stuff
       manuals/               latex documents (e.g. this manual)
   tmp/                       temporary storage
```

Table D.1: Overview of the $NEMO/ tree

```
* src/                  THE SOURCE TREE
      kernel/            All of these needed for full NEMO implementation
          cores/          independent but otherwise essential utilities for IO
          io/             getparam & filestruct (user interface, database i/o)
          misc/           miscellaneous library utilities
          yapp/           YAPP plotting device drivers
          loadobj/        Dynamic object loader
          tab/            Table manipulation stuff
          fortran/        Fortran to C interface
      nbody/             Nbody
          cores/          essential utilities for Nbody
          io/             I/O get_snap etc.
          init/           create models
          evolve/         Evolve (integrate) models
             hackcode/     Treecode
                export/     --export version - no bells and whistles
                source1/   Barnes and Hut (official c version)
                treecode/  Lars Hernquist fortran version of treecode
             multicode/    Simon White's multipole expansion (Barnes version)
             aarseth/      Sverre Aarseth
                nbody0/
                nbody1/
             potcode/      Non-selfconsistent particle integrator
             silly/        Silly demo version of N^2 code
          trans/          Transformation utilities for snapshots
          reduc/          Analysis utilities
      image/             2- and 3D image analysis
          cores/
          io/
          fits/
          misc/
      orbit/             Orbit analysis
          io/
          cores/
          potential/       Potentials
             data/          repository of potential source code files
      tools/             Tools, independent and mostly (PD) borroware
          movie/           movie for sun raster files (PD)
          movietool/       movietool for sun raster files (PD)
          ds/              image display program (Sebok)
      tutor/             Example programs and Makefile to toy with
```

Table D.2: Some pieces of the `$NEMO/src/` tree (this branch is officially exported)

```
(*)usr/                        User contributed software tree
       aarseth/                --Sverre Aarseth
       josh/                   --Josh Barnes
           clib/
           hackcode/
               export/
               source1/
           multicode/
           nbody/
               bodytrans/
               snapshot/
               tools/
       lars/                   --Lars Hernquist
           treecode/
       makino/                 --Jun Makino
           nbody/
       mbellon/                --Mark Bellon
           gravsim/
       nemo/                   --NEMO
           demo/
           maint/
           news/
           util/
               shar/
               f2c/
           yapp/
       piet/                   --Piet Hut
           clib/
           newton/
       pjt/                    --Peter Teuben
           clib/
           image/
           nbody/
           orbit/
       pswisnov/               --Peter Wisnovksi
           ccddisplay/
           snapsmooth/
       unemo/                  -- Micro Nemo (old version)
           lib/
           nbody/
```

Table D.3: Some pieces of the `$NEMO/usr/` tree (this branch is officially not exported)

Appendix E

Benchmarks

In this section we'll discuss some of the benchmarks. How fast does the N-body treecode run? To what degree does optimization/vectorizing help? When do programs become I/O dominated? Some of the numbers quoted below should be taken with great care, since a lot of other factors can go into the timing result.

E.1 N-body integration

E.1.1 Treecode

The standard NEMO benchmark of the treecode integration is to run `hackcode1` without any parameters. It will generate a spherical stellar system in virial equilibrium with 128 particles, and integrate it for 64 timesteps (`tol=1 eps=0.05`). In Table E.1 the amount of CPU (in seconds) needed for **one** timestep is listed in column 2. When not otherwise mentioned, the code used is the standard NEMO `hackcode1` with default compilation on the machine quoted. Note that one can often obtain significant performance increase (factor of 2 on some sparc architectures) by studying the native compiler and in particular its optimization options.

The `gravsim` code[1] is better suited for a multiprocessor machine. Its user interface and database format are however different from NEMO's and interface scripts can be defined which make working with this code a little easier. Both these C versions of the treecode (`hackcode1` and `gravsim`) are inherently slower because they are recursive and spend most of their CPU time in treewalking (with a lot of integer arithmetic). The modified (vectorized) Hernquist fortran

[1] C version code of the treecode written by Mark Bellon - Urbana, IL

Table E.1: Treecode Benchmarks

Machine	cpu-sec/step	code	comments
Dec-alpha	0.0042	hackcode1	-O4 -fast
Dec-alpha	0.0048	hackcode1	default
CRAY X/YMP48	0.0060	TREECODE V3	estimate (1989)
Onyx-2	0.0088	hackcode1	default (1996)
ETA-10	0.010	TREECODE V2	estimate (1987)
Sun Ultra-140	0.012	hackcode1	-xO4 -xcg92 -dalign -xlibmil
Sun 20/62	0.013	hackcode1	default (1994)
Cyber 205	0.018	TREECODE V2	estimate (1986)
Sun 20/61	0.020	hackcode1	
HP/UX 700	0.020	hackcode1	
Sun Ultra-140	0.024	hackcode1	default
Sun 20/??	0.024	hackcode1	-xO4 -xcg92 -dalign -xlibmil
G3 PowerPC 250Mhz	0.026	hackcode1	-O Sun 10/51
0.029	hackcode1	-O -fast -fsingle	
Cray-2	0.029	TREECODE2	REAL - Pitt, oct 91
DEC DS3000/400 alpha	0.036	hackcode1	default compilation
Pentium-100	0.038	hackcode1	default
SGI Indigo	0.045	hackcode1	default compilation
CRAY YMP	0.059	hackcode1	default compilation
Sparc-10	0.063	hackcode1	using `acc -cg92`
486DX4-100 (linux)	0.068	hackcode1	default
486DX2-66 (linux)	0.093	hackcode1	-DSINGLEPREC
Sparc-2	0.099	gravsim V1	
IBM R/6000	0.109	hackcode1	default cc compiler
Dec 5000/200	0.116	hackcode1	
Sparc-2	0.130	hackcode1	-DSINGLEPREC -fsingle
Sparc-2	0.180	hackcode1	
Multiflow 14/300	0.190	hackcode1	
Convex C220	0.290		
NeXT	0.240		[ganymede 68040, nov 91]
Sparcstation1+	0.340		
Sun-4/60 Sparcstation 1	0.420		
Alliant FX??	0.430	gravsim V1	
Alliant FX4/w 3 proc's	0.590		
VAX workstation 3500	0.970		
Sun-4/60 Sparcstation 1	1.040	treecode2	cf. C-code @ 0.420
Sun-3/110	1.660	hackcode1	fpa.il
Sun-3/60	2.280		
Sun-3/60	5.400		-fswitch
3b1 (10Mhz 68010)	49.000		
386SX (16Mhz)	87.000		(linux) software floating point

Table E.2: N-Body0 Benchmarks

machine(name)	time1 (sec)	time2 (sec)	speed
Sun-4/110(Pele - 8Mb)	21,753		0.41
Vaxstation 3100(Miffy - M48, 24Meg)		1302	0.65
Sparc 1		1023	0.83
Sparc IPC(Courage - 16 Mb)	9,015	850	1.000
Sparc 2	4,483		2.01
Sparc 2'		417	2.04
Dec 5000/200		318	2.67
Stardent(ism)		211	4.03
IBM Risc (Juno)	2,117	198	4.27
IBM Risc (wibm01)	2,115		4.26
Convex		172	4.94
HP/UX 700		26.2	
Cray YMP		19.1	44.5

code (referred to as `TREECODE V3`) has an approximate speedup of about a factor 200-400 over the original VAX/Sun-3 speed on a CRAY supercomputer.

It is also perhaps interesting to quote that replacing the `sqrt` function by a very fast machine dependant one will increase the speed of the C version of the treecode by about 20%. Some recent HP computers have a special hardware floating point operation to perform `1/sqrt()`.

E.1.2 Nbody0

The program is Aarseth's simplest nbody code (contained in Binney and Tremaine, 1987, no regularization or nearest neighbors). The input is a Hubble expanding cartesian lattice, w/ 925 pts, GMtot=1, expansion factor = 6 (omega = 1.2). Long version followed for 60 time units, short version for 5. Results are summarized in Table E.2[2]

E.2 Orbit integration

Benchmark is taking 100,000 leapfrog steps. For 2D optimized potentials[3] the timing on a Sparc-1 station is about 12" for `log` or `plummer`, and 23" for `teusan85` in the core region (orbit remaining within the body of the bar).

[2]Table E.2 compiled by D. Richstone
[3]Compiled with -DTWODIM

more coming

Appendix F

Potentials

This Appendix[1] lists a number of potentials that are available in NEMO. Most NEMO programs that deal with potentials have three program keywords associated with potentials: **potname=, potpars=** and **potfile=**, describing their name, optional parameters and optional associates filename (or other textual information). Each section below details a potential, and explains the usage of the **potpars=** and **potfile=** keywords. Mostly $G = 1$, unless otherwise mentioned.

F.1 athan92

potname=athan92 potpars=Ω,*amode*,*n*, $\frac{a}{b}, r_l, Q_m, \rho_c, f_{disk}$

F.2 bar83

potname=bar83 potpars=$\Omega, f_m, f_x, \frac{c}{a}$

F.3 bulge1

potname=bulge1 potpars=$\Omega, M, R, b/a$

[1] Automatically generated from CTEX comments in the `$NEMO/src/orbit/potential/data` source code

F.4 ccd

potname=ccd potpars=Ω, X_o, Y_o **potfile=***image(5NEMO)*

F.5 cp80

potname=cp80 potpars=Ω, ϵ

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, and an axis ratio is not well defined. 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

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

F.6 dehnen

potname=dehnen potpars=Ω, M, a, γ

Walter Dehnen (1993, MN **265**, 250-256) introduced the following family of potential-density pairs for spherical systems:

The potential is given by:

$$\frac{\Phi = -\frac{M}{a} \times -\frac{1}{2-\gamma} * 1 - (r}{r+a)^{2-\gamma}}$$

cumulative mass by

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

and density by

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

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.

F.7 expdisk

potname=expdisk potpars=Ω, M, a

Exponential disk (BT, pp.77)

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

F.8 halo

potname=halo potpars=Ω, v_0, r_c

F.9 harmonic

potname=harmonic potpars=$\Omega, \omega_x^2, \omega_z^2, \omega_z^2$

Harmonic potential

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

F.10 hernquist

potname=hernquist potpars=Ω, M, r_c

Potential is given by:

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

and mass

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

and density

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

F.11 hom

potname=hom potpars=Ω, M, R, τ

F.12 hubble

potname=hubble potpars=Ω, M, R, b, c

F.13 isochrone

potname=isochrone potpars=Ω, M, R

F.14 jaffe

potname=jaffe potpars=Ω, M, r_c

Jaffe potential (BT, pp.237)

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

F.15 log

potname=log potpars=Ω, M_c, r_c, q

Logarithmic Potential (BT, pp.45, eq. 2.54)

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

F.16 mestel

potname=mestel potpars=Ω, M, R

F.17 miyamoto

potname=miyamoto potpars=Ω, a, b, M

F.18 null

F.19 op73

potname=op73 potpars=Ω, M_H, r_c, r_h

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

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

F.20 pappet83

potname=pappet83 potpars=Ω, M, k, b

F.21 phubble

potname=phubble potpars=Ω, M, R, b

F.22 plog

potname=phubble

F.23 plummer

potname=plummer potpars=Ω, M, R

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

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

F.24 rh84

potname=rh84 potpars=$\Omega, B, a, A, r_0, i_0, j$

F.25 rh84k

F.26 rotcur

potname=rotcur potpars=Ω **potfile=***table(5NEMO)*

F.27 selwil93

potname=selwil93 potpars=$\Omega, M_b, \frac{a}{c}, M_s, M_h, r_s, r_h$

F.28 test85

F.29 teusan85

potname=teusan85

F.30 triax

potname=triax potpars=$\Omega, \Phi_0, R, \epsilon_{min}, \tau$

A growing triaxial potential. Does not conserve energy.

Appendix G

Installation, updates and exporting

This Appendix describes installation and maintenance of NEMO. NEMO currently consists of about 10Mb of ASCII (source, documentation etc.) files, mostly obtained as a (compressed) tar file. For some machines a bootstrap loader is present, which makes the initial installation extremely straightforward (however, with frequently some subsequent tweaking, e.g. choosing the right graphics drivers). In NEMO V3 the installation has been simplified using autoconf.

The first Section G.2 walks you through an installation using the new configure script. For the most it is likely to be close to the procedure on any standard UNIX system, although a number of known deviations are described in Section G.3. Exporting a working NEMO system is described in Section G.4. Incremental updates (import as well as export) are described in Section G.5. Finally, various maintenance issues are discussed in Section G.6.

The installation examples shown below assume you are using something like the `csh` shell; you would likely have to replace commands such as `setenv`, `source`, `rehash`, `set` etc. by the appropriate ones for another shell. We mereley leave this as an excersize for the reader (:-).

G.1 Bootstrap

G.2 Linux workstations - NEMO V3.x

We are going to install NEMO in the directory designated by the environment variable `$NEMO` on a Linux workstation. The **NEMO** environment variable will be needed (see also Appendix A), and will generally be set in your `.cshrc` file. We will do this a little later. Even if your are not on a Sun workstation, please read this section carefully and the next section on installation procedures on non-Sun UNIX systems.

If you haven't done so yet, create NEMO's root directory now (or make at least sure that the directory is empty):

```
    % mkdir $NEMO                (or whatever $NEMO will be)
    % cd $NEMO
```

It is perhaps convenient to install a special "`nemo`" user. In case your site has active users that are donating software to NEMO (the ultimate goal of this project) the sub-directories need to have write permission for them. The use of UNIX 'groups' may come in handy here. To begin with, `$NEMO/usr` is intended for this purpose.

We will now take you through the installation step by step[1].

G.2.1 tar

A tar "image" of the source code will be either on tape, or is already available[2] on disk. Extract it from tape (the name of the tapedrive in this example is `/dev/rmt0`):

```
    % tar xf /dev/rmt0
```

or extract it from a disk tarfile (called `nemo.tar` here):

```
    % tar xf nemo.tar
or:
    % zcat nemo.tar.Z | tar xf -
or:
```

[1] Remember there is also the `bootstrap` method, which automates large parts of the following steps

[2] See Section G.4 how to create such tar files from an existing NEMO installation

```
    % gzip -dc nemo.tar.gz | tar xf -
or:
    ftp> get nemo.tar.gz "|gzip -dc | tar xf -"
```

depending on the nature and location of the (compressed) tar file. It may have been compressed (`.Z`) with the standard UNIX *compress(1)* program, or the GNU utility `gzip`. Instead of using *uncompress(1)*, the utility *zcat(1)* can be piped to *tar(1)* to extract the tar file much more efficiently. You may even transfer the file accross the (inter)net via the `ftp`[3] utility and piping it's output straight into `tar` to avoid having to store the tar file on your local disk.

BE SURE to be in the `$NEMO` directory while doing this! On some UNIX systems it may happen that `tar` complains about not being able to create directories or such. You can probably ignore it, since SYSV `tar` wants to preserve ownership (stored by a user id, which may be unknown or even invalid on your target machine). The `tar` program may have a flag for this.

If you haven't set the **NEMO** environment variable (cf. Appendix A) now is really a good time to do it.

First add the following to your `.cshrc` file, somewhere before the **PATH** environment is defined:

```
setenv NEMO /usr/nemo                        (or whatever)
source $NEMO/NEMORC
```

After having read the `NEMORC` startup file, a number of environment variables and alias will have been added to your current environment. Include the `$NEMOBIN` directory in your search path before any major system areas, *e.g.*:

```
set path = ( . $NEMOBIN $path)
```

and make it current:

```
% rehash
```

Now that all source and documentation is in place, the installation can start. As said before, the top level `$NEMO/Makefile` can steer the whole installation process heirarchically, calling `Makefile`'s in lower level directories and so on and so on.[4]

[3] not all versions of `ftp` allow this feature though

[4] At least, this is the goal of this game

G.2.2 make dirs

First we need to complete the creation of the NEMO directory structure[5] as needed for this particular `$NEMOHOST`. Also a few files within the system need to have write permission by all users:

```
% make dirs
```

You will likely see comments that certain directories and files exist, which is fine. This install command can be rerun without any danger when updates come in. If you get some kind of error message like:

```
Badly placed ()'s.
*** Error code 1 (ignored)
```

it is most likely that you are within a strict SYSV-like environment, that directly imports the SHELL variable into `make`, and since our makefile's are written with the assumption that `/bin/sh` is the make-shell, one has to use something like:

```
% make dirs SHELL=/bin/sh
```

By copying the appropriate `make.*` script from `$NEMO/src/scripts/` into `$NEMOBIN`, this problem is generally solved.

Also a few files are created that are needed when running NEMO. These are summarized in Table G.1. If you expect to run the import/export scripts and be in regular contact with the central source archive, you need to give them your **NEMOSITE** name alias, and edit the `NEMORC.local` file appropriately. The **YAPPLIB** probably needs to be set to one of the available **YAPP_*** variables from the `NEMORC` file, since the default is **YAPP_NULL** (no graphics output!). The sensible thing to do, is going through the installation of the library (as discussed in Section G.2.5), after which some tests in the directory `$NEMO/src/kernel/yapp` will tell you which graphics device is easiest to install on your machine.

G.2.3 make scripts

The next step is to copy a number of important[6] scripts into `$NEMOBIN`:

```
% make scripts
```

[5] A full listing was presented in Appendix D

[6] only the `cc` script is installed now

Table G.1: Installation created files

File	Purpose	Notes
etc/motd	New announcements	You edit to suit your site
adm/TIMESTAMP	installation	do not hand `touch` this!
adm/Usage	usage log	needs `chmod a+w` permission
NEMORC.local	import/export	edit at least NEMOSITE
nemo_start	easier startup	not created with `make`
VERSION	version id	must be `major.minor.patch` numbers

Currently the NEMO environment needs a search path where the system C-compiler is preceded by our own `cc` script. This script then calls the real UNIX C-compiler (often `/bin/cc`). On some UNIX machines, a `make` script needs to be installed too to precede the system *make(1)* utility. This will be discussed later.

If you don't like the choice the installation procedure has given you, pick another one, and copy it yourself instead, for example, to select the gnu compiler, issue:

```
% cd $NEMO/src/scripts
% cp cc.gnu $NEMOBIN/cc
% rehash
```

G.2.4 More on scripts

In the previous section only a few very essential scripts were copied to `$NEMOBIN`, but the directory `$NEMO/src/scripts` contains a number of other useful scripts for users, as summarized in Table G.2.

Table G.2: Useful user scripts for NEMOBIN

Script	Purpose	Notes
bake	make replacement	Template in `$NEMOLIB`
mknemo	Find and install programs	
mkpdoc	check if doc file needs update	Also called by mknemo
manlaser	send manual pages to (laser)printer	Ignores small .so files
rmsf	remove NEMO binary data files	

There are some more scripts, but they are more useful for NEMO maintenance, as summarized in Table G.3, and should perhaps not be copied to `$NEMOBIN` for direct access by users.

Table G.3: Useful maintenance scripts

Script	Purpose	Notes
nmlist	listing of library	
ranlib	fake if none present	needed on SYS5
tardot	gather all hidden . files	see also submit

G.2.5 make install

A full installation starts with installing the essential libraries, after which application programs can be compiled at will.

On a Sun4 workstation the default installation is invoked by:

```
% make -i sun4 >& make.log &
% tail -f make.log
 .....
  <Control-C>                           #  to abort tail
```

We have added an (stdout + stderr) redirection of the screen output to a file here and also put the job in the background for convenience. Also, the extra `-i` flag will force the installation through possible errors. The log file `make.log` is to check for errors afterwards.

Particularly at this stage, before you compile source code, you may wish to review if certain system dependant features compiled correctly, and patch the system. No good facilities are available from the top level, but `Makefile`'s in the appropriate directories usually contain information and are flexible enough to patch the system with a minimum amount of effort. A number of standard patches are described in the next Section.

G.2.6 mknemo

The previous process only installs the necessary libraries. At this stage each program needs to be loaded manually. The script `mknemo` should be installed for this (see section G.2.4). To install the programs `hackcode1, mkplummer` and `snaprstat` you could issue:[7]

```
% mknemo hackcode1 mkplummer snaprstat
```

The `mknemo` script is a little bit smart about certain different structures of compiling programs, *e.g.* if there is no single sourcecode found (as with `hackcode1`),

[7] you may need a `rehash` command here

it does accept a directory with that name, after which the local `Makefile` is allowed to take over. Clones, such as `hackcode1_qp` cannot be installed this way though.

Don't forget to run the `rehash` command if a command was new. Everything should be set up to use NEMO!

G.2.7 Documentation

The documentation of NEMO is manyold: first of all there is a *Users and Programmers Guide* (what you are reading now), which is currently in LaTeX format. Before you format this huge document, be sure to check if the `nemo.dvi` or even the (compressed) `nemo.ps` postscript file are not present in the manuals directory:

```
% cd $NEMO/text/manual
% zcat nemo.ps.Z | lpr
```

or compile it:

```
% cd $NEMO/text/manual
% make nemo
```

and print it:

```
% dvipr nemo    # or whatever your local version is called
```

The program `makeindex` is needed by LaTeX, it is a public domain utility, see the local Makefile for more details. The actual commands may be slightly different on your host. If `makeindex` does not compile or work for you, a dummy zero length file `nemo.ind` must be created, before LaTeX can be run. You will have to live without an index in this case, or ask the distributors for the `nemo.dvi` or proper `nemo.idx` file. Be also aware that the manual (`nemo.tex`) uses include files (`.inc`) using the TEX "`\input`" command, and may also include direct postscript files using the special \ `PSinsert` command. There are a few logical "flags" using the TEX \ `newif` command, *e.g.* in the top of the `nemo.tex` document they are declared as:

```
\newif\ifnemo        % declaration of the logical "variables"
\newif\ifdebug
\newif\ifindex
...
```

```
\nemotrue            %  setting the variables to true or false
\debugtrue
\indextrue
...
\ifindex             % example use of such a conditional
   \printindex
\fi
```

In particular, the \ `ifdebug` variable is used to comment the margins with all the items that go into the index. This makes proofreading the manual very efficient.

The second form of documentation are online manual pages. For all programs there is (or at least should be) a manual page (see *man(1)*). These can be individually printed using a shell-script `manlaser` or:

```
% manlaser program.1
```

or

```
% troff -man -t program.1 | lpr -t
```

The command script `$NEMO/usr/nemo/maint/PrintMan` can be used to print out **all** manual pages 1,3,5 and 8, but also check the `catman` target in the toplevel Makefile. This creates `fmt` files, that can be sent directly to the printer. Be careful about redundant information because of the `.SO` troff command. The `fmt` are normally logical links in this case, the script should take care of this, and only print out proper files and not links or zero length files.

Third, there are the *doc* files. They are needed by various front-end shell (as described in Appendix B). The script `mkpdoc` will check if an update of this *doc* file is needed; this script is called by `mknemo`, the quick and dirty procedure to install programs. In principle the *doc* files don't have to be saved, since they can be generated from the source code.

Fourth, there are the *ctex* files. In a number of places the source code contains mathematical details in TeX format. A small utility, `ctex`, extracts the these from the source code, and standard `tex` utilities can be used to format them. For example, most of the *potential(5NEMO)* descriptors have an associated CTEX section, from which Appendix **??**:potential can be generated automatically. Other examples are `anisot.c` and `mkop.c`.

G.2.8 SUN only: mirtool, ds, movie's

A number of window based utilities are available for Sun workstations. To install them, you have to do it after the general NEMO installation, as some of them need the NEMO library.

See `$NEMO/Makefile` for details how to install them.

G.3 Tailoring

NEMO has also fairly successfully been ported to various other UNIX machines, such as a CONVEX supermini computer, several GOULD machines, a VAX workstation running Ultrix 3.0, a Multiflow Trace-14 and Cray Unicos. They all required a few simple modifications, about which this section reports.

The first few (small) steps in the top-level Makefile require some attention, in specific the cc-compiler and the make-utility have been assumed to have too many options which are generally not true on all versions of UNIX. Note that these 'inflexibilities' may disappear with time when BSD and SYSV UNIX merge (expected release SUN OS 5.0 1992?).

G.3.1 Replacement scripts in NEMOBIN

Before NEMO V3 programs like cc, make and ranlib were sometimes replaced with a script in $NEMOBIN.

G.3.2 YAPP graphics device driver

In compiling the major libraries and utilities in NEMO the proper graphics package should be installed too. The `$NEMO/src/kernel/yapp` directory contains a number of yapp implementations and your need to decide which yapp interface(s) to use.

To test a particular graphics device you can use:

```
% cd $NEMO/src/kernel/yapp
% make yapptest_mongo
```

where the trailing `_mongo` part tests, in this case, the mongo device driver.

This usually means that the **YAPPLIB** environment variable in the startup script `$NEMO/NEMORC.local` has to be adjusted accordingly.

G.3.3 Math libraries

There are one or two programs which use *Numerical Recipes in C* routines, and the library `libnumrec.a` should be present in `$NEMOLIB`, or be symbolically/logically linked to the real one if references remain unresolved during linking. Note we might want to use double instead of float. Some other routines use IMSL, and we also supply a few emulated IMSL routines in `$NEMO/src/kernel/misc/imsl.c`

G.3.4 LOADOBJ dynamic object loader

This package is still a major problem on most non-BSD UNIX implementations, and unfortunately many useful programs depend on it's functionality. The BSD version (Sun3, Sun4, Ultrix) as well as SystemV COFF version are fairly stable, but versions for complicated cpu's or non-standard object code loaders, such as Alliant, Convex, Unicos and Multiflow have given problems. See source code documentation in `loadobj.c` must contain an entry for the proper `loadobjXXX.c` file. Good news is that the new SYSV4 revision, as well as Sun OS 4.1 include the *dlopen(3x)* library utilities for dynamic linking.

G.3.5 Tailoring the NEMO kernel

It can be handy to know what a minimum subset of library routines will be in order to run major portions of NEMO. Some of the comments below have never been tried out, so beware! Also some comments apply how to tailor certain routines for less functionality, if your local hostmachine is not as sophisticated. It is our intention to have them defined in a file `options.h`, although this may complicate multi-cpu shared source code. In such cases they may need to be defined via the *cc* script **-D** command line option, just as we currently use the **-I** switch.

The source code of most of the barebone NEMO library routines can be found in `$NEMO/src/kernel/` and below.

loadobj.c: This has been discussed before, a proper symbol must be defined, in order for `loadobj.c` to include the correct loadobj source code type. Note that the trigger may be used for you if no symbol has been supplied. usually *cpp* supplies a trigger, such as *sun, sparc, unicos, mips* etc.

bodytrans.c: This routine is normally compiled with the **-DSAVE_OBJ** compilation switch in which case bodytrans functions, which are not yet found, will be saved to the standard repository directory `$NEMOOBJ/bodytrans`. this option is now standard defined in `options.h`.

getparam.c: This file contains many `#define`'s which allow you to add a number of extensions to the user interface which have been discussed in Appendix B: data history mechanism, interactive keyword and menu prompting, parsing of expressions of *getXparam()* variables, interrupt to the review section, remote machine execution, etc. For more documentation see the source code. none of these run through `options.h` though.

image.c: This routine handles all image i/o (*e.g.* read_image, write_image, create_image), and can be compiled in **-DCDEF** or **-DFORDEF** mode, depending how your matrices in c should be stored in memory. The fits interface may not work properly when the wrong def is used???

G.3.6 isolation

If certain portions of NEMO's subroutine library appeal to you, yet you don't want to drag the whole NEMO library along (this may not always be possible though) you can isolate them, with probably the following modifications:

- • A call to *error* must be replaced with a simple *printf* followed by *exit*, since these are supposed to be fatal errors, and the program should abort.
- • A call to *warning* must be replaced with a simple *printf*. no need to exit the program though.
- • A call to *dprintf* must be replaced with a simple *printf*, by leaving of the first (integer) argument, or by totally leaving it out of the code. It is merely a debug-type printf, and could also be used with an `#ifdef debug`.

These are all small printf-like routines.

G.4 Exporting

There are a few automated ways to make full and partial (incremental) export versions of NEMO. Full exports are described here, partial and incremental backups in the next section. Again, it's best to be in NEMO's root directory:

```
% cd $NEMO
```

The first possibility is to make a tar-file of the barebone source:[8]

```
% make tarfile [FILE=nemo.tar] [SRC="src usr"]
```

With the optional FILE keyword (note the upper case!) you can save the tar-file in a directory in which you have write permission, just in case your name is not `nemo`. The default for `FILE` is `nemo.tar` in the current directory. It is currently about 10 Mb, if both the `src` and `usr` tree are exported.

It is adviced to compress the tar file, since in this case it saves a considerable fraction:

[8]To exclude hidden dot-files, issue a "`make purge`" before creating exporting tarfiles

```
% compress nemo.tar
```

The second possibility:

```
% make tarbackup [BACKUP=nemo-fullbck.tar]
```

creates a complete tar backup image on a diskfile. this diskfile can be any device (file/tape) using the optional backup keyword. Note that this tar image will be very large, and is only handy to copy NEMO between identical systems, or make a backup of the whole system, since also binary files are backed up.

G.5 Small updates: tar import and export

The directory `$NEMO/adm` is the working directory from where incremental updates are processed. A nightly script can be set up by the NEMO system manager to export new files to all other relevant sites which run NEMO on a regular basis. All new files are bundled together in a tar file, to preserve directory structure and time stamps. These tar files are then sent to the central site or other sites. Similarly, other sites may export their tar files with updated files to your local site. These imported tarfiles are normally located in `$NEMO/adm/import`. Scripts `import.csh` and `export.csh` handle the traffic between tar files and the local directory. See also *import(8NEMO)* and *export(8NEMO)*. Note that inter-site collisions are not always handled properly.

G.5.1 make tarfile

If you know what to export, you can also use the `tarfile` target in the root install `Makefile`. For example, in the case where NEMO has to be ported to a system with not that much filespace, you could start by copying the kernel first on the originator (as shown by the `1%` prompt below) into a tarfile, the tarfile is then copied to the destination (as shown by the `2%` prompt below) and extracted out there, after which installation can start:

```
 1% cd $NEMO
 1% make tarfile ASCIIDIRS="inc src/kernel"
....
 2% cd $NEMO
 2% tar xv nemo.tar
 2% cd $NEMO/src/kernel
 2% make install
```

This would copy all default ASCIIFILES, as specified in the root `Makefile`, and your choice of ASCIIDIRS into the tar file (this bare minimum NEMO kernel is currently about 2Mb).

After having gone through installation, as described earlier, the next step could be to add the N-body package (or any other one listed in the `$NEMO/src` directory) on top of it:

```
 1% cd $NEMO
 1% make tarfile ASCIIDIRS=src/nbody ASCIIFILES=
....
 2% cd $NEMO
 2% tar xv nemo.tar
 2% cd $NEMO/src/nbody
 2% make install
```

This tarfile is currently about 1.5Mb.

Typically the installation procedure adds relevant routines to the NEMO library, after which `mknemo` should be able to install requested programs from that package.

Table G.4: Small export tar file

ASCIIDIRS	ASCIIFILES	Description
"inc src/kernel src/scripts"	-	Bootstrap kernel
src/tools	""	Various utilities
src/nbody	""	Nbody integration
src/image	""	Image utilities
src/orbit	""	Orbit utilities

G.5.2 WEB maintenance

G.6 Maintenance

The directory `$NEMO/usr/nemo/maint` contains a number of shell scripts and C-programs which may be useful to do a few consistency checks, cross correlate two versions of nemo etc. see local documentation, as this is still part of the old NEMO V1.x release.

In the directory `$NEMO/adm` a few administrative files will grow over time. They may have to be cleaned up on regular intervals, or their features need to be shut down. see also Section G.5.

- the file `NEMO.LOG` is the accumulated log of the nightly script, which exports new files to remote sites.
- the file `Usage` is a usage list of all NEMO programs. It can be used to create various statistics about usage by user, program, machine, time etc. Use at your own discretion. the `Usage` file is created by the user interface, *getparam(3NEMO)*, and a logical defined flag **USAGELOG** in `getparam.c` needs to turned off. All of NEMO's programs would need to be recompiled of course.

Appendix H

Troubleshooting

Fatal errors are caught by most NEMO programs by calling the function `error` (see *error(3NEMO)*); 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 *adb(1)* or *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 H.2 below all of the environment variables used by NEMO are listed and their functionality. Sometimes they interfere if used in conjunction with other packages.

H.1 List of Run Time Errors

This section presents an alphabetical list of fatal error messages, as generated by *error(3NEMO)*. Although this list is not meant to be complete, it hopes to report on the most common bizarre 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 B 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 *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 `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 `#define`d in `filesecret.h` by the macro `StrTabLen`. Recompile the `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 *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 *filestruct(5NEMO)* format. Apparently it was not, file may also have been in new filestruct format. **Hints:** Try *tsf(1NEMO)*, *hd(1NEMO)*, *od(1)* and as last resort *more(1)*.

```
gethdr:         bad magic: 0164
```

Input was attempted on a file assumed to be in *filestruct(5NEMO)* format. Apparently it was not. **Hints:** Try *tsf(1NEMO)*, *hd(1NEMO)*, *od(1)* and as last resort *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. **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. *E.g.* in the case of *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 *orbit(1/3NEMO)* library/commands. The standard UNIX utility *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 `nm(1)` to find out, or use `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.

```
Malformed or non-terminated attribute-value list.
```

This error is actually from one of the Sunview routines, and is a result of a too large buffer to display (in header or 'below' a mouse button). It will happen when the number of files in a directory is too large to display below the **Run** button in `mirtool`.

```
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 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). The SUN OS does, as well as BSD 4.3 (?). Perhaps a special man-script/alias with an extra `-M` flag needs to be installed in this case if MANPATH is not supported. See also the lpath variable on some systems.

```
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(1NEMO)*. Recompile the program with a more recent version of ⟨snapshot/put_snap.c⟩ and possibly ⟨snapshot/body.h⟩.

```
readparam:      No interactive input allowed
```

The keyword **help=** or the equivalent environment variable **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 `rsh` program may not exist on your host, in which case the *getparam(3NEMO)* might as well have been compiled without the **REMOTE** flag. The other possibility is that the `.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 *execvp(3)* normally fails. A third possibility is that the remote machine did not have the executable present.

```
Badly placed ()'s
```

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\(aux\)
```

H.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 *bodytrans(3NEMO)* functions can be stored for retrieval. The default is `/usr/nemo/obj/bodytrans`. Normally set to `".:$NEMOOBJ/bodytrans"` in `NEMORC`.

CFLAGS Options for the C compiler for on-the fly compilations under NEMO V2. Not used for NEMO V3. See e.g. *bodytrans(1NEMO)*. When not set, no compilation options used. When set, some make(1) implementations will also use it when the environment is imported.

DEBUG Debug level, must be between 0 and 9. The higher the number, the more debug output appear on *stderr*. The default is 0. See *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 `vi` (see *vi(1)*). See also *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 *getparam(3NEMO)*.

FLOAT_OPTION Used by Sun3s native *cc(1)* compiler which floating point unit to use. Options are, amongst others, `ffpa, f68881`. This environment variable has to be turned off on Sun4s.

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 *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 *getparam(3NEMO)*). A few old programs may use the keyword `amnesia=` for this.

HOSTTYPE In case of multiCPU environment, which has to be served from the same `NEMORC` and/or `.cshrc` file, this variable will have the CPU type in it, *e.g.* `sun3` or `sun4`, which are used to break up the `bin`, `lib` and `obj` directories. It is also used in some Makefiles.

INCLUDE List of directories from where to include NEMO header files when compiling. Used by the *mycc(1NEMO)* preprocessor and some cc-scripts. This environment variable is not actively used anymore. *–deprecated–*

LIBRARY List of directories used to resolve NEMO library references in the compile/link command. Used by the *mycc(1NEMO)* preprocessor and some cc-scripts. This environment variable is not actively used anymore. *–deprecated–*

MANPATH Used by UNIX to be able to address more than one area of manual pages. Normally set to `$NEMO/man:/usr/man` by the `NEMORC` file. Does not work in Ultrix 3.0, but perhaps the `-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 `.cshrc` startup file. No default.

NEMOBIN Directory where nemo's binaries live, defined in `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 `mirtool/miriad` are stored/retrieved. The default is the current directory.

NEMOLIB Directory where nemo's libraries live. Normally set by . No default. `NEMORC`.

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 A

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 *yapp(3NEMO)* the program was linked with. See also *getparam(3NEMO)* and *yapp(5NEMO)*. YAPP is also used as system keyword, in which case the environment variable is ignored.

YAPPLIB Libraries needed to link a program which the default YAPP graphics device. No default.

See also the manual pages of *files(1NEMO)*.

H.3 Known Bugs and Features

A few known bugs and features are listed here, some of which are hard to "fix". Some of them are also of little interest to users, but programmers have to be aware of the pittfalls.

- Of items with the same name only the first one can be accessed if the item is **within** a hierarchical set. On the top level items are accessed sequentially. In particular, a C-programming construction like

  ```
  get_set(instream,tagname)
  while(get_tag_ok(instream,myitem)) {
      get_data(...)
  }
  get_tes(instream,tagname)
  ```

  would run into an infinite loop. Programmers should be careful not to construct such datasets, in particular it means one cannot portably wrap an existing dataset inside an item. Multiple history items are an example of this behavior.

- a file with multiple snapshots which vary in size doesn't get it's memory reallocated, and hence programs will likely to crash if any snapshot is larger than the first one[1] The "problem" is in the file `$NEMOINC/snapshot/get_snap.c`. A possible solution is to free the snapshot, or just reset the snapshot to a NULL pointer, and force it to be allocated again. One surely relies on a well designed swap space management system. Here's an example of this terrible coding kludge:

```
    Body *btab;
```

[1]currently the output of `unbind` may do this for you

```
for(;;) {
    get_history(instr);
    if (!get_tag_ok(instr, SnapShotTag))
        break;
    get_snap(instr, &btab, &nbody, &tsnap, &bits);

    ...

    btab = NULL;
}   /* for(;;) */
```

Appendix I

Glossary

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`".

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.

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 (`miriad` and `mirtool`) 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`".

potential Dataformat in NEMO used to represent a potential; most programs in NEMO which handle potentials start or end with "`pot`".

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. Most programs that handle *snapshot*'s in NEMO start or end with "`snap`".

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

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).

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.

Appendix J

Future, Present and Past

This Appendix contains a number of fairly random remarks on some remaining difficulties in NEMO, and what we may do to resolve them, and other things which may happen to NEMO over the next years[1]. A proper history is given of its past, and can be found in itemized form in J.4

J.1 Some present problems

J.1.1 Graphics

Yapp can only be linked in once, there's no dispatcher yet, and different versions of the program exist for different graphics output (printers also) devices.

It would be nice to have more general dispatcher, which fully utilizes the `yapp=` system keyword, may be used such that only one compiled version of the program is needed.

E.g. on the SUN we could use:

yapp_core: generic yapp using suntools (deprecated)

yapp_cg: extended suntools version with color support (deprecated)

yapp_ps: output to a postscript file, for Postscript device. (color not yet implemented)

yapp_mongo: interface which connects to mongo-87. (deprecated)

[1] The rumors of my death have been greatly exaggerated...

yapp_pgplot: interface which connects to pgplot package. This implementation also has the great advantage of being able to handle a variety of terminals and printers fairly transparent. This is the one we use mostly.

yapp_plplot: interface which connects to plplot package.

We use the environment variable YAPPLIB in Makefile's. This environment variable is also set by your local NEMO guru in NEMORC.local file (normally initialized by reading it by a users `.cshrc` file) or you have to add them specifically to your `.cshrc` file. This makes it easier to install a new version of NEMO where a different graphics package will been used.

In some older versions of NEMO some of the Makefile's may have not been modified to have these flexible setups, or the variable.

Sometimes, the need occurs for a specific YAPP interface, besides default one. Programs which specifically address a graphics device have the base name (*e.g.* `snapplot`, appended with an underscore and the device ID, *e.g.* `snapplot_ps` or `snapplot_cg`. Best is to check the $NEMOBIN directory.

The installer should carefully read the `Makefile` and/or `README` file in the directory `$NEMO/src/kernel/yapp` for instructions regarding specific installations of yapp interfaces.

Programs would have to be recompiled manually, as in the following example, because most Makefile's have hardwired graphics library names in them:

```
% make snapplot YAPPLIB="$YAPP_PS"
% mv snapplot $NEMOBIN/snapplot_ps
```

J.1.2 System independent file structure

Currently the file structure is tied in with the operating system routines *fread(3)* and *fwrite(3)*, and binary files cannot be guarenteed[2] to be used across machines with different data types (size/floating point conventions etc.). This problem is only partially solved by using programs such as *tsf(1NEMO)* and *rsf(1NEMO)*: it still requires physical data modification, transport and again modification. (UNICOS) cannot be read on a SUN, however datafiles on e.g. an Alliant, Multiflow and SUN are binary compatible because of IEEE floating point numbers and the proper size twos complement integers. Support exists now for byte-swapping, such that files on Sun and Dec can be read and written either way. However, machines like the CRAY supercomputer with it's deviating size and floating point format will have to convert their data as is exemplified in Section 5.6.

[2] for IEEE and twos-complement data automatic byte-swap detection this problem has been solved

A possible solution is the way data is written to disk in a package as `miriad`: the layer just before the *fread/fwrite* packs the data in some predefined standard format (IEEE floating point and twos complement integers seem an obvious choice at this time). This causes a small overhead on some machines, and on other machines it is nothing more than a copy operation or even passing of pointers.

J.1.3 File size - float vs. double

For a really large number of bodies (to take the example of particle pushers) file-size becomes important for many analysis programs which become I/O bound:

- It's not always necessary to keep information in double precision. Images are also stored in doubles, in good faith with old C, where all math is intrinsically done in double anyhow. This would save a factor of 2 in space. There exist data i/o routines with force float/double conversions (*get_data_coerced()*).
- It's not always needed to keep all 6 phase space coordinates, besides the data structure of snapshots has phase coordinates rather well tied in, it's not easy yet to separate positions and velocities, and only store positions in a datafile. This would save another factor of 2.
- Images: totally unneeded to have them in double precision. Should become `'float'`. Since we have used the `real` type

J.1.4 Some shortcuts and hints

- You can set **$NEMO** to a 'universal' path as "/usr/nemo" and make a symbolic link of this file to the actual physical location of nemo. In a file cluster system the actual location of NEMO might be on a common fileserver. Normally you have to become superuser to make the link

```
# cd /usr
# ln -s /usr/guinness/nemo
```

  This means that your `.cshrc` can always keep the same

```
setenv NEMO /usr/nemo
```

  and have the symbolic link take over the work for you.

- The use of FLOAT_OPTION is encouraged on the SUN3, because it allows a flexible change from a system with different floating point hardware. Perhaps the CFLAGS in most Makefiles should have the inclusion CFLAGS=$(FLOAT_OPTION) for compatibility reasons with other-than-SUN systems??? Beware that only the NEMO variable is exported to a Makefile (but see below)

- When, while trying to compile, the cc compiler does not seem to find the NEMO include file, it is probably an indication that the `cc` in `$NEMOBIN` has not the proper flags. In particular, some versions of the `cc`-compiler do not support the -L flag (e.g. Ultrix, Sun UNIX 4.2 Release 3.1FCS) Make sure the '`cc`' and '`make`' are properly placed in the `$NEMOBIN`.

- On a SUN3 floating point intensive programs will run a lot faster when the inline floating point libraries are used. Instead linking with the standard math library (`-lm`), link it with `/usr/lib/f68881.il` or `/usr/lib/ffpa.il`. The fortran TREECODE only gains about 5% in speed, but floating point intensive programs can gain up to 30% in speed.

- The best run-time performance from SUN4 compilers for compute-bound applications is usually obtained from some combination of the following compile-time options:

```
Fortran 1.3.1:
        -O4 -cg89 -libmil -dalign -fnonstd -Bstatic
C 1.0:
        -O4 -cg89 -libmil -dalign -fnonstd -Bstatic -fsingle
```

  These are discussed in the Numerical Computation Guide which accompanies C 1.0 and Fortran 1.3.1. Also the default `swap` and `/tmp` partitions supplied by SunOS are often insufficient to fully optimize some large programs. Use *swapon(8)* in the first instance and `-temp=...` compile option, described in *cc(1)* and *f77(1)*, in the second instance.

J.2 Future

A wishlist and what may be forthcoming in some future release of NEMO:

- graphics: yapp_server to work across machines. Not necessarily X11 server - but likely so. This finally means a full implementation of the yapp= system keyword and also would make executables a lot smaller. Also a *plot(3)* interface.

- graphics: yapp_x: An honest X11 graphics server.

- image display: more support for display facilities, e.g. *ds(1L)*.

- loadobj should understand constant expressions
- SPH : utilities. Makes Lars happy.
- The everlasting expansion of the manual: more examples, tutorials for course work, Figures, tables etc etc.
- loadobj for COFF (SUN386i, most SYS5 implementations, Convex). (partially done, 3b1 version works)
- ???shared libraries for the latest SUN OS 4.1 system??? Is a rather laborious thing, and difficult to maintain in a environment where the library is frequently upgraded.
- dynamic object loader for N-body diagnostics
- options.h through stdinc.h or nemo.h?
- handle multi-snapshot files more efficiently
- loadobj for MF, UNICOS, Alliant and Convex do not work!! The only reliable implementation we have is BSD (SUN OS) and Ultrix and a SYSV (3b1). SPARC also seems to be stable, and MIPS COFF is not quite done yet.
- loadobj in yacc?, nemoinp in yacc? (cf. sm)
- Utility for chaining programs in (nsh?) shell scripts, automatic passing of in/out files from on to the next. See `pipe` shell script for example.
- Install the official NBODY1 and NBODY2 programs from Aarseth with NEMO's user interface and file structure, as has been done for NBODY0
- Adaptation of all C programs to the ANSI standard. Usage of the GNU gcc compiler recommended for portability. This process is now underway (march 90 - PJT). It is also likely that the GNU make program will be used on the long run, and making small updates to the package will be a lot easier. (more portable `make`?)

J.3 New Features

J.3.1 Release 3.0

Was released April 1, 2001. Certainly not a joke. Source code has been released within CVS for subsquent development with the partiview and starlab modules, also under CVS. Installation has now completely been done using configure, an autoconf product. 3.0.0 was never released, 3.0.1 was the first official release that worked under linux. Solaris and SGI.

- configure support
- loadobj support now using .so files, not .o files.

J.3.2 Release MD-2.5

Was released December 1999, with initial support for configure. This series had 5 subreleased though 2.5.5, but 2.5.6 was never released in favor of 3.0.0. No major changes through this release.

J.3.3 Release MD-2.4

Was released April 1, 1997.

J.3.4 Release MD-2.3

- We are now using an official versioning scheme, with major, minor and patch-level.
- The dreaded dynamic object loader has been made to work on the DEC Alpha (OSF1 V3.2) and SGI (IRIS 5.2) using a new system utility `ldso`. This has increased the portability, but not taken away the fundamental difficulty in installing NEMO on a new operating system.

J.3.5 Release MD-2.2

- Full support for Solaris 2.x. (tested on 2.3) The default graphics `yapp` device should now be generic X and Postscript. loadobj method is now supported by the operating system.
- Compiling with **-DSINGLEPREC** actually caused lots of programs, include `hackcode1`, not to run properly. Most of the bugs associated with this have

been removed, but for example LINUX is very sensitive to such errors and still a lot of programs will crash in this mode.

J.3.6 Release MD-2.1

- *Literate Programming:* trying CTEX embedded comments in e.g. the potential descriptors, and `anisot.c`
- Properly documented and advertised use of using binary structured files in pipes.
- Some support for Starlab in the form of NEMO++. Translation programs to convert from a `dyn` to a `snapshot` have been written. Apart from properly ANSI-coding the NEMO kernel, the routine `nemomain.c` needs to be present in C++ format too: `nemomain.C`.
- Potentials are now supposed to return a pattern speed (even if it was not changed) into the first argument of the potpars array. This to deal with rotating potentials.

J.3.7 Release MD-2.0

With this major release upgrade the directory structure has been modified away from 'user' oriented to 'topic' oriented. The 'user' oriented stuff is now under `$NEMO/usr`, whereas the more stable 'topic' oriented under `$NEMO/src`. Some files live in both, in which case the `$NEMO/src` version should take preference.

- *dlopen()* version of loadobj, as well as *ldl* ("gnu"). Still some problems, but they are Sun bugs, not ours. Certain complicated expressions fail. *dlopen* may also work on NeXT.
- *yapp* and *loadobj* separated out and cleaned up
- small subset of numerical recipes maintained
- stories in snapshots
- random access additions to filestruct, and support for little and big-endian machines using auto-byte-swapping. Hence no Macro's used. Slow?
- *bodytrans* using *fie*.

J.3.8 Release MD-1.4

Yet another non-release - summer 1990.

- Miriad shell implemented as `nemo` through an alias. The old `nemo` program renamed to `nemoshow`. The `mirtool` can also be compiled to `nemotool`.
- Sault's FITS I/O routine replace Werong. No real need for a Fortran-C interface anymore. New FITS routines in image.
- Experimental Micro-NEMO in `src/nemo/micro`.
- Manual now reaches about 100 pages.

J.3.9 Release MD-1.3

Newly released option until February 1990.

- potential(5NEMO) has extra time parameter, relevant programs have been updated.
- hackcode3 is an experimental version which allows an extra external potential through the potential(5) format. It also has the option of keeping the first `nrigid` particles rigid, and more silly options are expected. It should be noted that hackcode2, an experimental toy, has never been released.
- The user interface `getparam()` now supports reading the value(s) of a keyword from a file using the `key=@file` construct.

J.3.10 Release MD-1.2

Newly released option until November 1989

- Introduction of the HOSTTYPE environment variable in the NEMORC file. This meant that new environment variables such as NEMOBIN, NEMOLIB and NEMOOBJ are derived from NEMO and HOSTTYPE. See the NEMORC file. It also meant that basically all Makefile's had to be updated.
- Libraries are a bit screwed up now, and best is to include both libT.a and libJ.a in the minimum list of libraries. Makefiles are being updated for this. One can also use libNEMO.a, and use utilities such as mklib, addlib and mkbin.

- bodytrans enhanced, has a proper database of expressions, which is dynamically updated when bodytrans(3) is used. The file `$NEMOOBJ/bodytrans/BTNAMES` contains a list of extra expressions currently understood. Make sure those directories are write permissible by the world.
- Loadobj is now also functional for Sparc (sun4) and COFF (System V UNIX). The SPARC version requires loading with **-Dsun4 -Bstatic** since often required symbols, such as integer multiplication, are in shared libraries. This unfortunately makes those binaries larger than they could be.

J.3.11 Release MD-1.1

A description of the major differences between the existing IAS version and the newly released MD version 1.1 (summer 1989):

- Improved user interface: suntools menu interface at help=8. The sophistication of the user interface is determined at compile time through a number of compile switches in `getparam.c`
- Standardized usage of some standard interactive facilities using `setparam` instead of and with help of `getparam`.
- Installation through Makefiles is more flexible. The installation and various administrative utilities are more streamlined to make porting to non-SUN's easier.
- Aarseth's straight N-body code - as published in BT87 is in nbody0 and an Ahmad-Cohen version in nbody2 (this last one has not been tested out well enough) These programs have a NEMO interface, which also handles some Fortran-C interface questions.
- filestruct_old is now obscure: all of Piet's clib programs have been converted (snaplist, snapenter, snapdist, mkplummer)
- various new yapp's, the $NEMO/src/nemo/yapp directory has been cleaned up and documentation has been updated.
- bodytrans saving: the program bodytrans can be tested and also save files (**-DSAVE_OBJ** compile switch) - same for library routines.
- snapplot and movies work better together; snapplotedit.
- potcode
- ds, image display on image(5) or fits(5) files.
- fortran interface
- GRAVSIM added to nemo/src tree

- lars/treecode has been added, but needs a good recursive fortran compiler. NEMO interfaces built in. Not been able to test well - Does not work on Suns..
- The NEMO startup file is now called `NEMORC`.
- This expanded manual.

J.4 HISTORY of NEMO

Oct 86 - Jun 87 Initial development on a network of SUN3 workstations at the Institute for Advanced Study, Princeton, NJ by Barnes, Hut and Teuben.

Jun 87 export version, for easy installing on BSD4.2, we call it alpha version 1.0a.

Jul 87 test phase for installing on an VAX 8300 running Ultrix 1.2: mods: Makefile, new cc and make (cc has no -L flag on ultrix) tsf.c casting of pointers and advancing pointers fixed

-Mar 88 various installations at MIT, Drexel U. in Philadelphia, U.of Illinois at Urbana (Convex, Alliant, Gould) tested and done. New programs and updated programs keep coming in at a steady rate.

Mar 88 Many UIUC changes: getparam() has a few new system keywords, (debug, yapp, host). History mechanism in data file I/O means that now get_hist() and put_hist() must be called, optionally add_hist(). dprintf() added to getparam.c for user debugging aid, can now be called in user programs. Yapp_mongo used. Environment variables YAPPLINT and YAPPLINP are now encouraged in user Makefiles. Documentation into one big TeX file for users as well as programmers.

Jun 88 IAS and UIUC versions have been merged again. Improved filestructure (Josh), user interface, data history mechanism, yapp interface (Peter). Documentation end manuals significantly improved. Still a number of items in the 'problems' area not resolved.

Nov 88 IAS and UIUC versions have been merged again.

Spring 89 Manual updated - working on class/course problems - lots of working examples added to manual. MD version is now slowly diverging from IAS version.

Summer 89 Groningen version installed on a combined SUN/Alliant network - shows difficulty of maintaining a shared disk environment with different versions of the binaries (binX, libX, objX, datX) -

August 1989 Version prepared for official beta release 1.1. (Maryland)

December 1989 Experimental multi-CPU release (1.2) (Maryland)

February 1990 Minor upgrades, mainly SUN4 and multi-CPU bugs. (1.3). Toronto's version has been dubbed ZENO now.

May 1990 Slightly expanded manual for the Pittsburgh Workshop and some minor upgrades every here and there.

Summer/Fall 1990 An attempt to merge Starlab and Nemo - added some extra functionality to filestruct and merged story concept. Total directory structure overhaul: all code related to a particular topic is in its own directory in `$NEMO/src`. The tree starting at `$NEMO/usr` will now be used for user contributed software. In particular, `$NEMO/src/kernel` contains a small core of NEMO which can be used without any of the parallel branches.

Fall 1991 Slowly progressing the `$NEMO/src` tree. A few new sites for export maintained.

Summer 1993 Some support for C++ and Starlab.

January 1994 Solaris 2.x support, started WWW.

Winter 1995 Linux, Dec Alpha and SGI support added, since they allow dynamic object loading. Manual in html (latex2html)

Spring 2001 Installation converted to configure/autoconf.

Index