NEMO: A Stellar Dynamics Toolbox

Peter Teuben (e-mail: `teuben@astro.umd.edu`)
University of Maryland, College Park, MD, USA.

Abstract

We describe NEMO, a collection of software with which a wide range of experimental situations can be constructed, dynamically evolved and analyzed. Great importance has also been given to make the package extendible and to flexibly import and export data. NEMO has been implemented within and integrated into the UNIX[1] environment. A unified command syntax provides users with a simple help facility, and programmers with a ready-made user interface. A general method for hierarchical structured binary data files is used to communicate between programs.

The Appendix describes a proposal to adopt FITS, a widely used data format in the astronomical community, as a basis to interchange and archive simulation results. It would be a specific implementation of the now standard FITS BINTABLE extension.

Keywords: Stellar dynamics – Data interchange – Software – Numerical methods

1 Introduction

With the advent of powerful workstations in the mid 80s the use of N-body simulations has become increasingly widespread. It has been realized (e.g. Hut and Sussman 1986, Hut 1993) that it would benefit this part of the astronomical community to have a programming environment similar to what many observers have had for years in packages like AIPS and IRAF. Most researchers in the field have their own private packages, some of them with a large degree of sophistication. A first attempt to structure these efforts into a package were reported by Barnes et al (1988), and became known under the name NEMO.

2 System Components

The following components can be identified within NEMO:

- A common **user interface** is shared by all programs; all program are executed from the UNIX (or equivalent) shell, and accept a *keyword=value* based command line parameter list. There are **program** and **system** parameters, the latter one shared by all programs, and control global properties such as error handling, debug output, graphics devices etc.

  This user-interface allows you to switch to a different look-and-feel user-interface using it's plug-compatible property. Some nice examples have been constructed in interfacing NEMO with the MIRIAD (an AIPS-like interface) and KHOROS/CANTATA (a visual programming language) packages. In practice however, we find shell scripts ("batch mode") the most powerful, and self-documenting, mode of operation.

[1] this paper assumes some basic knowledge of UNIX

- A common **file-structure** is shared by all programs. Data, used to communicate between programs, is stored in files in a hierarchical structure of name- and type-tagged items in binary format. Items can be single scalars, multi-dimensional arrays and even arbitrary (C-type) structures. In addition each program can have an input and output channel which is connected to a pipe instead of a disk-file. Since many programs transform one dataset into another, it allows for efficient chaining to build useful new tools.

- On top of this file-structure, a number of **packages** have been defined. The most important one is the *snapshot*, on which NEMO was originally based. Subsequently *images*, *tables* and *orbits* were introduced, with a number of programs that convert data between them.

- A common **graphics interface** is defined on the Applications Programming Interface (API) level. One simply needs to link with any graphics library (e.g. mongo, pgplot, X11, postscript, suntools, GL) to create a working program. This hides the complexity of different graphics programs, but is of course not as flexible since a fairly low common denominator has to be chosen for the API.

- Programs can use a **dynamic loader**, in order to compile, load and run user-specific code on the fly. This supports very flexible and powerful analysis tools. For example, in orbit integration a user-defined potential can be used to integrate and analyze orbits, and for snapshots arbitrary user-defined expressions are used for many analysis- and plotting programs.

3 Documentation

Extensive on-line help is present on three levels. First of all, each running program comes with dynamic inline help, reminding the user of the keywords, their default values and a small one line help with the meaning of a keyword. It is part of the user interface, for which the programmer is responsible. The advantage of this lowest help level is that it is much less likely to be out of date than the ones described next and does not need the whole NEMO environment.

The second form of help is a standard UNIX manual page for each program. This is an example how NEMO is integrated into the UNIX environment, and where it immediately opens up interactive hypertext/browsing of documentation with "standard" tools like `xman` and `tkman`. The third form of help is an extensive "Users and Programmers Guide". It contains numerous examples how to use, program and extend the toolbox. Recently, NEMO has also entered the World Wide Web and now distributes information from a central site.

4 Future

After the on-going conversion of supporting Solaris 2.x, NEMO will have formal ANSI C and C++ support, possibly to allow a tight connection with AIPS++. Also, support for other packages, such as Starlab (Hut, Makino, and McMillan, priv.comm.), and TIPSY (Katz, priv.comm.) is being worked on.

5 Example

To illustrates some of the features in NEMO we show the following C-shell script that analyzes the results of a contrived encounter between two spherical Plummer models:

```
# make two plummer spheres, the second one not in virial equilibrium
mkplummer out=p1 nbody=1000
mkplummer out=- nbody=1000 | snapvirial in=- out=p2 'virial=1/sqrt(3)'
snapstack in1=p1 in2=p2 out=p.in deltar=10,0,0 deltav=-1.5,0,0
# integrate
hackcode1 in=p.in out=p.out tstop=10 options=phi,acc freqout=1 > p.log
# analyze last 3 dynamical times
foreach t (8 9 10)
  snaptrim in=p.out out=- times=$t |\
    snapcenter in=- out=- |\
    unbind in=- out=- |\
    snaprect in=- out=p.out.$t weight="-phi*phi*phi" > p.rect.$t
  # freeze potential, and classify 50 orbits in the rectified potential
  foreach i (`nemoinp 1:1000:20`)
    stoo in=p.out.$t out=- ibody=$i |\
      orbint in=- out=- nsteps=10000 dt=0.01 potname=hackcode1 potfile=p.out.$t |\
      orbname in=- >> p.orb.$t
  end
end
```

Acknowledgements: NEMO was initially developed by Joshua Barnes, Piet Hut and myself during the 1986-87 academic year in Princeton, I wish to thank them for the stimulating environment in those initial years. I would also like to thank the Monbusho International Scientific Research Program : Joint Research 03044032, for a visit to Tokyo, where some of the work described here has been done.

References

Barnes, J.E., Hernquist, L., Hut, P., and Teuben, P. 1988, *Bull. Americ. Astro. Soc.*, **20**, 706.
Cotton, W.B., and Tody, D. 1992, "Binary Table Extensions to FITS: A Proposal". NRAO technical report.
Hut, P. 1989. in: IAU Colloq. 109. "Applications of Computer Technology to Dynamical Astronomy"
Hut, P. and Sussman, G.J., 1986. In: "The Use of Supercomputers in Stellar Dynamics", ed. P. Hut and S. McMillan (Springer Verlag, Berlin) p 193.
Wells, D. C., Greisen, E. W., and Harten R. H. 1981, "FITS: A Flexible Image Transport System", *Astron. Astrophys. Suppl*, **44**, 363.

A APPENDIX: Data Exchange Proposal

This is a summary of a draft proposal to define an interchange and archive format for particle (NBODY) data. It will be fully described in an upcoming paper by Teuben (in preparation). A prototype implementation is available within NEMO under the name `snapfits`.

A.1 Why a new format, and which?

It it important that there is a common data format with which particle data (N-body, SPH, etc.) can be exchanged and archived. There will always be different analysis packages around, and as time progresses, so will their formats. With this in mind, the following the requirements for such an interchange/archive format are readily defined.

First of all, the data must be stored in a machine independent format, well-documented and to some extend be self-describing. The dataformat must be flexible in order to store a wide variety of data structures, in other words it must be extendible. And lastly, the data must be stored efficiently. This eliminates virtually any ASCII based format, particularly if precision is important.

Within the various branches of observational 2- and 3D-imaging astronomy this had been realized a long time ago, and resulted in the creation of the FITS (Wells et al. 1981) format that is now an accepted standard (not only astronomical). It can store data in binary as well as ascii format, but is mostly known for interchanging and archiving images. Recently binary tables have been added as a FITS extension (FITS is an extendible data format description), which has greatly improved its efficiency. The rows in binary tables may be a mixture of a number of numerical, logical and character data entries. In addition, each entry is allowed to be a single dimensioned array (with obvious extensions to higher order arrays). Numeric data are kept in IEEE formats, and hence guarantee machine independency. These binary tables are an excellent way to store N-body data.

We propose to use this BINTABLE extension to implement our particular NBODY format. There are several reasons why we want to use the BINTABLE format. First of all, the BINTABLE extension has now been properly defined (Cotton and Tody, 1992) and ratified by all three regional FITS committees, although it still needs to be ratified by the IAU. Secondly, the FITSIO library (Pence, 1992) is available for a wide number of machines. It is implemented in FORTRAN, but an experimental C calling interface is also available. Thirdly, various databanks are becoming available to the astronomical community. CD-ROM's already exist with collections of standard astronomical images and tabular material from the literature (e.g. ADC, AAS). Theory should not lag behind. Important simulations should be made available to the astronomical community in a well described format. Certain NASA divisions even require their public data to be stored in FITS format.

A.2 Implementation decisions

The BINTABLE format is merely a prescription to store tabular data, but there are a number of implementation details that have to be decided on, before we all start to write our data in some kind of FITS format. Although any FITS reader should be able to read (and/or skip) our

NBODY data, only specialized readers would know what to do with the data. General table browsers, such as the `FTB` program, should however be able to manipulate such datasets. We note the following implementaton decisions:

- Naming conventions of the column names (the `TTYPE`'s). This will be the most visible part of the implementation. Apart from registering the extension name (NBODY) with the FITS office, our community needs to agree on things like the column names, units, coordinate systems etc. See the example below.
- Datasets with different kinds of particles, which have different attributes. For example, SPH particles have extra attributes (such as temperature and density) which pure N-body particles formally don't have. It would not always be very efficient to tag along such unneeded data-attributes. The "variable length array" construct, proposed within the BINTABLE proposal could solve this problem, but this has to be worked out. Another possible solution is to store them in separate so-called FITS extensions. This is clearly a much simpler and easier to understand implementation.
- Some variables may be global variables, and result into more efficient storage. For example, if all the masses (`MASS`) are the same, they may be stored in the header (which however is in ASCII). Another example is `TIME`; although in principle each particle may have it's own variable time, a snapshot is commonly defined with a unique time for all particles.
- For certain complex situations extra data structures may be needed to fully describe an experimental situation; for example, describing hierarchical double and triple stars could be stored in an accompanying table, since FITS allows for this. The names and descriptions of these extensions would need to be part of the standard too, but they can always be added to the proposal at a later date.

A.3 Software

Followup documents and discussion are available through a World Wide Web server using the URL `http://astro.umd.edu/nemo`. A sample fortran program, that has to be linked with the Pence FITSIO-library, is also available.

A.4 Example Binary Table Header

Finally we show an example of a binary table header with 6 columns of number of different data types and dimensions. Columns labeled `POS`, `VEL`, and `ACC` are arrays of dimension 1 and length `NDIM`. The column labeled `CODE` is a character string of length 4. The nonstandard keywords `TIME`, `COORDSYS`, etc. also appear at the end of the header, and identify global parameters of the dataset. Although copied by any FITS reading program, their meaning is only understood in the context of an "`NBODY BINTABLE`". Note the units are not SI, which is what the IAU recommends to store data in FITS format. The (binary) data followed at a multiple of 2880 bytes after the last keyword of the header (always `END`) is not displayed.

NOTE: The first two lines of numbers are only present to show card columns and are not part of the table header.

```
         1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
XTENSION= 'BINTABLE'           / Extension type
BITPIX  =                    8 / Binary data
NAXIS   =                    2 / Table is a matrix
NAXIS1  =                   76 / Width of table row in bytes
NAXIS2  =                 1024 / Number of rows in table  (number of Nbodies)
PCOUNT  =                    0 / Random parameter count
GCOUNT  =                    1 / Group count
TFIELDS =                    6 / Number of columns in each row
EXTNAME = 'NBODY   '           / Standardized name of this extension
EXTVER  =                    1 / Version number of table
TFORM1  = '1D      '           / Double Precision value in col 1
TTYPE1  = 'MASS            '   / Type (label) of column  1
TUNIT1  = 'SOLARM  '           / Physical units of column  1
TFORM2  = '3D      '           / Array of NDIM Double Precision values in col 2
TTYPE2  = 'POS             '   / Type (label) of column  2
TUNIT2  = 'KPC     '           / Physical units of column  2
TFORM3  = '3D      '           / Array of NDIM Double Precision values in col 3
TTYPE3  = 'VEL             '   / Type (label) of column  3
TUNIT3  = 'KM/S    '           / Physical units of column  3
TFORM4  = '1F      '           / Floating point number in col 3
TTYPE4  = 'PHI             '   / Type (label) of column  4
TUNIT4  = 'KM**2/S**2'         / Physical units of column  4
TFORM5  = '3F      '           / Array of NDIM Floating point values in col 5
TTYPE5  = 'ACC             '   / Type (label) of column  5
TUNIT5  = 'KM**2/S**2/KPC'     / Physical units of column  5
TFORM6  = '4A      '           / Character string (4) in col 6
TTYPE6  = 'CODE            '   / Type (label) of column  6
COMMENT   Now follow some 'global' parameters to this dataset
TIME    =                 10.0 / Model time of this snapshot
COORDSYS= 'CARTESIAN'          / Coordinate system
NDIM    =                    3 / Number of dimensions
AUTHOR  = 'Teuben'             / Person who is responsible for the data
ORIGIN  = 'NEMO'               / Originator
DATE    = '05/12/93'           / dd/mm/yy when file created
DATEOBS = '28/08/93'           / dd/mm/yy when simulation done
HISTORY  NEMO mkplummer out=run301.in nbody=1024 massexpr=pow(m,p) \
HISTORY  NEMO  masspars=p,-2.0 massrange=0.1,10 VERSION=2.5a
HISTORY  NEMO nbody0 in=run301.in out=run301.out eta=0.01 tcrit=20 \
HISTORY  NEMO  eps=0.025 VERSION=1.2b
HISTORY  NEMO snaptrim run301.out tmp3421 times=10 VERSION=1.5a
HISTORY  NEMO snapfits tmp3421 run301.t20.fits
END
```