manuscript, version of 18 May 1993

Starlab Primer

Piet Hut

Institute for Advanced Study, Princeton, NJ 08540, U.S.A.

Steve McMillan

Department of Physics and Atmospheric Science
Drexel University, Philadelphia, PA 19104, U.S.A.

Junichiro Makino

Department of Earth Science and Astronomy, University of Tokyo
3-8-1 Komaba, Meguro-ku, Tokyo 153, Japan

Abstract

This primer contains a brief introduction to Starlab (version 1.0), a software environment for simulations of dense stellar systems. Starlab offers a flexible way to combine stellar dynamics, hydrodynamics and stellar evolution modules, in order to set up, execute and analyze simulations interactively.

1. Introduction

This document is a very preliminary version of what will someday become a real primer for Starlab. At present, it is not much more than a brief summary of the main ideas, together with some pointers to particular implementation aspects. This version specifically applies to Starlab 1.0, the first "frozen" Starlab version, dating from mid-May 1993. This version supersedes other more experimental versions that have briefly existed after the start of the C++ implementation of Starlab in November 1992. After mid-May 1993, Starlab 1.0 will be left unchanged, and will thereby provide a reference point for any question about compatibility. Bugs found in version 1.0 will be dealt with in future Starlab versions.

In this primer, we provide some motiviation for using Starlab, and present a brief user overview of the Starlab package. In §2, we desribe our reasons for developing the Starlab environment, and in §3, we outline its structure. The dynamical centerpiece of the project, the recursive hermite integration scheme, is introduced and described in §4. In §5, we turn briefly to our reasons for choosing C++ as our programming language, and in §6 we describe the flexible data structure used by all the programs in the package. Finally, in §7, we give a brief tour of the tools and libraries that make up Starlab 1.0.

2. The Starlab Manifesto

Starlab is a software package for simulating the evolution of dense stellar systems, and analyzing the resultant data. It is a collection of loosely coupled programs, linked at the level of the UNIX operating system, that share a common data structure, and can be combined in arbitrarily complex ways to study the dynamics of star clusters and galactic nuclei. Current improvements in both the quality and the quantity of observational data, together with ongoing and anticipated increases in available computational power, combine to make this project both necessary and feasible.

All stellar-dynamical N-body simulations rely on sophisticated integration schemes to follow the motion of all particles in the system under study. It is the job of the N-body program to deliver a faithful representation of the dynamical evolution of the system, along with information on all stellar interactions of interest, to the user, subject only to the fundamental limitations imposed by the chaotic equations of motion and the laws of physics. It is becoming increasingly clear that, in order to make detailed comparisons between simulations and the high-quality data now available, N-body (and perhaps also detailed Monte-Carlo) simulations are really the only viable option.

Performing N-body simulations is already is a complex and demanding task. However, generating data is only half the job. The other half of the work of a computational theorist parallels that of the observer, and lies in the job of data reduction. As in the observational case, a good set of tools is essential, and unless the tools can be used in a flexible and coherent software environment, their usefulness will be severely limited.

Three requirements are central in handling the data flow from a full-scale globular cluster simulation: modularity, flexibility, and compatibility. Starlab incorporates these three requirements.

To some extent, Starlab is modeled on NEMO, a stellar dynamics software environment developed six years ago at the Institute for Advanced Study, in large part by Josh Barnes, with input from Peter Teuben and Piet Hut (and subsequently maintained and extended by Peter Teuben). Starlab differs from NEMO mainly in the following areas: it emphasizes the use of UNIX pipes, rather than temporary files; its use of tree structures rather than arrays to represent N-body systems; and its guarantee of data conservation—data which are not understood by a given module are simply passed on rather than filtered out.

Modularity: A Toolbox Approach

Starlab consists of a large number of separate tools ("applications") and libraries, allowing it to be used by both the novice and the expert programmed with equal ease. The structure of the package will be discussed in more detail below. We have followed the UNIX model of combining a large number of small and relatively simple tools through pipes. This allows a quick and compact way of running small test simulations. For example, a study of relaxation effects in a cold collapse could be done as follows:

```
mkplummer -n 100 | freeze | leapfrog -t 2 -a 0.02 -e 0.05 | lagrad
```

Here `mkplummer` creates initial conditions for a 100-body system, according to a Plummer model distribution. The resulting data are piped into the next module, `freeze`, which simply sets all velocities to zero, while preserving the positions. Following that, the data are read in by the leapfrog integrator, which is asked to evolve the system for a period of 2 time units, with a (constant) step size of 0.02 time units, and a softening length of 0.05 length units. Finally, the resulting data are piped into a module that makes a plot of the Lagrangian radii of various percentiles of the system.

Flexibility: Structured Data Representation and Unfiltered Piping

A persistent problem in N-body (and other) simulations is the question of how to handle data outside the scope of a particular program. Often, this problem is "solved" simply by throwing away all unrecognized information, but this is clearly not workable when UNIX pipes are the primary means whereby data flows around the system.

In Starlab, each snapshot of a N-body simulation can be stored in a file in a standard format, with a header indicating the nature of the snapshot. In addition, a list of all the commands used to create the data is stored at the top of the file, together with the time at which the commands were issued, so as to minimize the uncertainty about the exact procedures used. Each individual body is presented as a node in a tree, constructed so as to reflect the presence of closely interacting subsystems and their internal structure.

Within this structure, each body has several unstructured "scratch pads," in which application programs can write diagnostics or other comments describing particular occurrences during the integration. This has proved to be extremely useful, by allowing various forms of data reduction to take place already during the run. Especially during complicated interactions involving stellar dynamics, stellar hydrodynamics, and stellar evolution effects, a free-format reporting system, tied to the individual interacting objects, is very helpful in allowing a reconstruction of episodes of greatest interest.

The internal data representation of each module is such that unrecognized quantities or comments are stored internally, in the form of character strings. They are reproduced at the time of output, at the correct position, preserving their correspondence with the initial bodies they were associated with (or their descendents, since some of them may have collided and merged). This allows the use of an arbitrary combination of pipes with the guarantee that no data or comments will be lost.

For example, in the command

```
evolve | mark_core | HR_plot | evolve
```

the first module evolves the system, integrating the equations of motion, while also following the way the individual stars age and interact hydrodynamically. The second program computes the location and size of the core of the star cluster, and marks those particles that are within one core radius from the center. The third module plots a Hertzsprung-Russell diagram of the star cluster (perhaps using special symbols for the core stars), before passing on the data once more to the module that evolves the whole system. For this to work, the `mark_core` program needs to preserve the stellar evolution information, even though it only "knows" about the stellar dynamical part of the data. Similarly, the `HR_plot` program needs to preserve the dynamical data.

3. A Bird's-Eye View of Starlab

Now let us turn to the overall structure of Starlab. The package contains three groups of programs, for use in studies of stellar dynamics, hydrodynamics, and stellar evolution. The separation between these three groups is not rigorous, since some of the programs act as bridges between these different parts. For example, a simulation of stars with finite radii and finite lifetimes can be driven by an integrator that computes the orbits of the bodies in the point-mass limit. Whenever two of these stars come close enough together to exhibit non-point-mass interactions, including tidal distortions and physical collisions, the stellar dynamics module notifies the hydrodynamics module to do the required additional computations. Similarly, when time progresses beyond the scheduled main-sequence lifetime of a star, a stellar evolution module can keep track of the increasing radius of the star as it climbs the giant branch. Nevertheless, it is convenient to maintain this distinction between the three groups (this distinction is also embedded in the structure of the Starlab directory hierarchy).

At present, all three sets of modules are quite rudimentary, although the stellar dynamics modules are close to the point of becoming interesting, and useful as real research tools.

Besides a simple leapfrog integrator, we have a code with a Hermite integration scheme (see §4), a generalization of Aarseth's polynomial integrator that includes a predictor-corrector for both acceleration and jerk (the time derivative of the acceleration). One interesting application of this integration scheme will be the use of a Teraflops speed high-accuracy special-purpose computer (the HARP, for "Hermite AcceleratoR Pipeline;" see Makino, Kokubo & Taiji 1993, and references therein). In addition, we include a preliminary version of a Hermite scheme that includes local coordinate transformations that deal with close encounters of two or more particles (`dynamic_hermite`). Since the dynamical integration lies at the heart of Starlab 1.0, it is discussed in more detail below.

For the hydrodynamics part, Fred Rasio has contributed a simplified version of his SPH (Smooth Particle Hydrodynamics) code, a stand-alone FORTRAN module, integrated into Starlab by connection modules written by Steve McMillan. Currently, this module only handles one two-body encounter at a time. Whenever two stars approach each other closely enough, a separate SPH integration is invoked, in which each star is replaced by a full SPH representation in terms of a few hundred particles. When the encounter has reached completion, as judged automatically by one of the SPH modules, the one or two remaining stars are collapsed into point mass again, and the dynamical integration continues (for more information, see §8). The degree of integration between SPH and the dynamical code will increase in future Starlab releases.

For the stellar evolution part, Simon Portegies Zwart has contributed an implementation of Peter Eggleton's fitting formulas for approximating stellar evolution (see the file `usr/spz/README`).

4. N-body simulations and the Dynamic Hermite Scheme

Over the past three decades, the development of N-body codes has been driven by some rather unpleasant facts of life concerning the range of length and time scales inherent in all collisional stellar systems. The shortest length and time scales of interest in the evolution of globular clusters and galactic nuclei are posed by the closest encounters of the most compact types of stars. In the worst case, this leads us to consider a near-grazing encounter between two neutron stars, an event with a duration of order of a few milliseconds. Compared with the age of a globular cluster, 10^{10} years, we have a discrepancy in time scales of some 20 orders of magnitude!

A similar problem shows up when we compare length scales. The diameter of a neutron star, expressed in units of the tidal radius of a globular cluster, is of order 10^{-15}. At first sight, modeling a globular cluster on a star-by-star basis would seem hopeless. The fact that we are able even to contemplate such an enterprise is due in large part to the ingenuity and persistence of Sverre Aarseth, who has provided a framework (not to mention a series of efficient implementations) within which these seemingly unsurmountable problems can be tackled. The key words for facing up to the length scale and time scale problems are individual time steps and local coordinate transformations.

The Time Scale Problem

During the 1960s, the importance of binaries in determining the dynamics of star systems—and their devastating effect on one's computer budget—was gradually realized. Their dynamical significance makes it essential that binaries be properly included in any simulation; the numerical difficulties they pose created a crisis (the first of many) in the N-body business. The reason was that, even with variable time steps, the early "standard" integration schemes forced all stars to share the same time step size. As a result, a typical time step size of a few hundred years in a loose association of ten stars was reduced to a month or less when the first moderately tight binary arrived on the scene. The integration effectively stopped with the appearance of the first dynamically interesting event!

The answer to this problem, provided by Aarseth (*cf.* Aarseth 1985), was the introduction of individual time steps. Instead of viewing a star cluster as a set of mass points in space, Aarseth replaced each star by an orbit segment. When the time had come for a particular star to move, it could determine the gravitational acceleration due to all its neighbors by asking them to slide along their orbit segments (implemented as polynomial approximations) to the desired point in time, even though each of them in turn had computed its own positions, velocities, and accelerations only at earlier times.

This single algorithmic improvement did more to speed up star cluster calculations than decades worth of hardware speed improvement. Even on today's fastest machines, it would be difficult to reproduce Aarseth's calculations of twenty years ago without using individual time steps.

The Length Scale Problem

The large range in length scales, although somewhat smaller than the range in time scales, has turned out to pose an even more significant problem for simulations of star cluster evolution. In contrast to the time scale problem, which simply slows things down to a crawl, the length-scale problem can easily make a whole calculation meaningless.

The main problem lies in round-off errors. When we consider a binary consisting of two neutron stars, moving in the outskirts of a globular cluster, there is no need to follow its internal orbit so long as other perturbers are far away (another significant software improvement provided by Aarseth). However, in the rare case that a third star would interact with that binary, we would have to follow all three point-particles numerically during the time of the interaction. Now the problem shows up clearly: for the position vectors of the two neutron stars, with respect to the center of mass of the cluster, the first fifteen digits could turn out to be the same, as we saw above. Even a double precision (64-bit) representation of floating point numbers typically carries a mantissa of only fifteen significant digits—so much for direct integration of the equations of motion!

Admittedly, this is an extreme case, but it makes a point: loss of accuracy due to round-off when subtracting large numbers is a very serious worry in N-body calculations. Aarseth's answer was to implement a series of ever more intricate treatments of two-body, three-body and more complex multiple encounters. These treatments are largely based on Kustaanheimo–Stiefel regularization procedures, in which the three-dimensional Kepler singularity is "unfolded" by a coordinate transformation to four dimensions, mapping the three-dimensional Kepler problem into that of a four-dimensional harmonic oscillator through the inverse of one of the Hopf maps from S^3 to S^2 (*cf.* Stiefel & Scheifele 1971), introducing a U(1) gauge symmetry in the process.

Further Code Development

Aarseth has implemented various other algorithmic improvements as well, most notably the Ahmad-Cohen neighbor scheme, in which the individual time step scheme is further refined to a two-time-scale approach (by a more frequent calculation of nearby interactions, compared to remote interactions). The resulting code, NBODY5, that includes all these improvements, has been the tool of choice for any type of detailed star cluster N-body simulation, for well over a decade. The community owes a debt of gratitude to the generosity and dedication of Sverre Aarseth, who has made both his codes—and himself—widely available, for friendly advice, as well as code repair and maintenance.

All these developments notwithstanding, there are still some major problems facing us, if we want to carry out a star-by-star N-body simulation of globular cluster evolution. First of all, if the past can offer any guidance, increasing the number of particles by a significant factor is guaranteed to uncover new algorithmic bottlenecks, requiring new solutions and fine-tuning. The present state of Aarseth's codes is the outcome of an evolutionary process of successive attempts to deal with increasingly harder problems, posed by the increase in complexity of the systems to which it has been applied. This process is likely to continue for quite a while.

Secondly, the large set of heuristic improvements, valuable as they have been in making calculations possible in the first place, pose a formidable problem to the implementation of stellar evolution recipes. Even simple questions such as the value of the distances between binary components at a given time becomes less straightforward as it may seem at first, when one realizes that one has to trace these distances in the guise of their four-dimensional transformations in which time itself is an extra coordinate, given as a function of the independent integration parameter. Add to this the complexities in the Ahmad-Cohen bookkeeping of the two time scales used (separately for each individual particle, introducing a multiplicity of orbit segment representations), and the various treatments of perturbed and unperturbed regularized many-body motion, and the full dimensions of the task begin to appear.

Thirdly, and equally important, it would seem less than ideal if all simulations in an entire field of astrophysics would be continued with a single code, without any independent form of comparison or calibration. In a sense, winning the star cluster N-body space race hands-down, twenty years ago, has been a mixed blessing for Sverre Aarseth. He has gained a lot of friends, but at the same time has lacked any significant form of competition. It is time to explore alternative approaches to N-body code writing. The "`dynamic_hermite`" integrator distributed with Starlab 1.0 marks the beginning of one such alternative.

A Recursive Approach

The central new idea we have introduced is that of recursive coordinate transformations. The underlying notion is that of hierarchical simplicity. Rather than giving a special treatment to each of a number of different closely interacting groups (binaries, triples, binary-binary encounters, etc.), we use a recursive approach to split up an interacting group of stars in subgroups, down to the level of individual stars. The advantage is twofold: we can now handle arbitrary k-body subsystems, for any value of k, and we can treat especially tight subgroups that may appear deep inside an already tight group (or even tighter sub-subgroups).

Most stars are unaffected by these special treatments and are simply represented as leaves of a flat top-level tree. The stars that take part in close encounters or are members of multiple star systems, however, play a crucial role in cluster simulations. The bulk of NBODY5, for example, is concerned with special treatments of closely interacting groups of ≤ 4 stars. In addition, in most simulations so far most of the computer time has been taken up by the integration of the equations of motion of this stellar minority.

In our treatment of such groups of closely interacting stars, the center of mass of the group is represented by a node in the top-level tree. This node in turn serves as the local root node for a binary tree (a tree with two branches per node), in which each star forms a leaf. The structure of this tree is changed dynamically, to guarantee that the tree structure closely reflects, at any time, the configuration of the (sub)groups of the stars. Our approach somewhat resembles that taken by Jernigan and Porter (1989), the major differences being that we limit ourselves to small subsets of stars, and at least for the time being forgo any fancy regularization technique.

Whether we can get away with a bare-bones set of recursive coordinate transformations, rather than four-dimensional regularizations, remains to be seen. We suspect, although we have not yet proved, that the main advantage offered by Kustaanheimo–Stiefel (and other forms of) regularization lies in the introduction of relative coordinates, rather than in the removal of the singularity in the 2-body Hamiltonian. But in any case, our recursive approach to a dynamical maintenance of a tree configuration is likely to alleviate our task of providing a clean interface between the stellar dynamics and the stellar evolution parts of our code.

The Dynamic Hermite Scheme in a Nutshell

The `dynamic_hermite` integrator is based on the Hermite scheme described by Making (19xx), a fourth-order accurate generalization of the familiar leapfrog scheme. Tests conducted by Makino and Aarseth indicate that it is marginally faster than the standard "Aarseth" polynomial scheme for the same accuracy; however, it appears to be substantially more robust. Its greatest advantage, however, is the fact that it is self-starting—i.e. no "historical" information about previous forces is retained. This apparently simple modification translates into a large reduction in programming complexity, as the system can be arbitrarily reconfigured without the need for corrections to the polynomial segments used in the Aarseth scheme.

Synchronization between various levels of the system is accomplished using block timesteps, as described by McMillan and Aarseth (1993). The block scheme effectively constrains all time steps to be a power of two. The use of blocks also increases the efficiency of the integrator, as particle positions and velocities are predicted once for each block step (involving many particles), rather than once per particle step. On vector machines, it also allows the corrector and time-step determination phases of the integration to be vectorized.

At present, no special treatment of the long-range gravitational force calculation is incorporated into the program (so that, in "Aarseth" terms, `dynamic_hermite` is equivalent to NBODY3, rather than NBODY5). The data structure is such that auxiliary force-evaluation schemes may be attached to the system; however, with the impending arrival of the HARP hardware, it is unclear to what extent such schemes will be actually be necessary.

5. Why we Chose C++

Starlab is written in C++, a language chosen for its combination of flexibility and availability. If we had been chiefly concerned with either criterion, we would probably have chosen a different language. There are certainly languages that are more elegant, systematic, leaner, and definitely more predictable! But in the end our decision to go with C++ was most driven by its wide availability, and the sense the language was here to stay, at least for the foreseeable future.

However available C++ has become, this availability so far does not guarantee either compatibility or consistency. As it is, we have had our share of problems with annoying things like incompatibility of compilers. As you will see in the file `inc/stdinc.h`, we have had to include quite a few compiler dependent switches. These switches have to be set when you first install Starlab. For detailed instructions, see the file `README` in the top level of any given Starlab version.

As for a lack of consistency, we have been quite alarmed about some of the statements we have read concerning the state of flux in which the language definition of C++ still finds itself. To quote one warning, ¿from the beginning of Chapter 9 in the book "C++ Programming Style", by Tom Cargill (an interesting book that mostly tells you what NOT to do, rather than what to do!):

> Multiple inheritance is a complicated and poorly understood part of C++. At first sight, multiple inheritance is appealing because ... On closer inspection, multiple inheritance turns out to be difficult to use effectively. The difficulty stems in part from the many subtle interactions hidden in the language semantics; it is hard to make programs behave as intended. The difficulty is also due to the limited situations in which the multiple inheritance mechanism supports useful relationships. ...
>
> ...In general, an inheritance hierarchy involving virtual base classes must be constructed very carefully to ensure that all objects that can be instantiated from any class in the hierarchy behave consistently. As Ellis and Stroustrup understate on page 296, "Such consistency is not easily achieved for, say, user-defined assignment

> operators." ...
>
> ... An effective way to learn the details of a programming language is to observe how experimental programs behave. Examining the output ¿from a small program and studying how the output changes due to small changes in the program often provide insight into how a programming language works. This technique works for most of C++, but cannot be used reliably for learning multiple inheritance. Unfortunately, virtual base classes are implemented incorrectly by many current C++ compilers, which should not be relied on for learning this part of the language. In preparing the material on virtual base classes, it had been my intention to use a simpler example of virtual base class semantics. I abandoned the simpler example when I discovered that none of my three C++ compilers implemented it correctly. (One compiler crashed, and the other two generated incorrect code—which behavior is preferable?) ...

For these reasons and others, we have made a conscious choice to limit ourselves to a subset of "standard" features that we have found especially useful in C++. Specifically, we have confined ourselves mainly to operator and function overloading, and to the use of classes as a way both to hide visibility of internal data, and to control the way in which data can be accessed and modified.

6. The Underlying Data Structure

Each star in a star cluster is represented by a single node in a tree that spans the whole ensemble of stars. This tree is implemented on every level of its hierarchy as a linked list of nodes. Each node has a pointer to its parent node (`parent`), its leading daughter node (`oldest_daughter`), and the nodes immediately preceding and following it (`elder_sister` and `younger_sister`, respectively). The details of our implementation are spelled out (literally) in `inc/node.h`.

A node is instance of the `node` class, a base class for derived classes such as `dyn` (containing positions and velocities, among other things) and `hdyn` (containing additional information for the use of Hermite integration schemes). Each node contains pointers to instances of two other base classes as well: `hydrobase` and `starbase`. The former forms the base class for derived classes such as `hydro`, for hydrodynamics applications. The latter forms the base class for derived classes such as `star`, for applications in stellar evolution. Note that the mass of each object is contained as a variable in the `node` class, and is therefore directly accessible to the `dyn`, `hydro`, and `star` part of the object.

The present data structure is clearly a compromise. It would have been far more elegant to implement everything through multiple inheritance. However, as discussed in the previous section, there are various reasons that made us decide against that option, at least for the time being.

A more abstract description of our data structure would bring out the dual role of the `node` class: that of providing access to the particle tree (its role as a node) and that of providing a base class for the `dyn` structure (its role as dynbase, although that term does not occur specifically in Starlab). A cleaner implementation would therefore have featured a node class, containing pointers to each of the following base classes: dynbase, hydrobase, and starbase. However, in practice this seemed to us to be unnecessarily complicated. Instead, we collapsed the roles of dynbase and node into one class: `node`.

The main reason for singling out the `dyn` class as the more fundamental of the three is that the structure of the tree is dictated by the stellar dynamics requirements. Indeed, it is exactly the problem of dealing with close encounters that drove us to put our particles in a flexible tree structure, rather than in a fixed array.

The I/O structure of Starlab reflects the more abstract view of our data structures. The tree structure is reflected in the lisp-like ordering of the nodes, and each node contains three "stories," one each for the `dyn`-type, `hydro`-type, and `star`-type classes. As briefly mentioned in §3, each story forms a lisp-like tree of character strings. Each string consists of one line of ASCII text containing the name and value of a single variable. Depending on the application program, each string is separately processed: it either is converted to a dynamical variable, or it is simply stored in the internal "scratch pad" containing the story. In this way, "unknown" data can be saved and passed on by any tool.

During the execution of an program, the internal representation of a story is still dynamically accessible. Variables inside a story can be read and written, through a form of internal I/O. The implementation details may be found in the file `src/std/story.C`.

7. A Quick Tour through the Starlab Directory Tree

The main directory for the standard set of source code is `src`, with additional user-provided software residing in a parallel and less structured directory tree under `usr`, followed by the individual user name (such as `usr/spz`). Include files belonging to the main programs in `src` are located in `inc`. Executables can be found in `bin`, while the various libraries reside in `lib`.

The main source code directory, `src`, contains six subdirectories: `std, node, dyn, hydro, star, alt`. The first, `std`, contains standard utilities, including the code for the story mechanism. The directory `node` contains those utilities that operate directly on the tree structure, without additional knowledge of the physics involved in the individual particle model. Specific physics is provided in the directories `dyn` (for stellar dynamics), `hydro` (for hydrodynamics) and `star` (for stellar evolution).

The last directory, `alt`, contains various FORTRAN codes, including both integrators and analysis packages. The subdirectory `nbody1` contains a simple individual (block) timestep Hermite integrator, written by Steve McMillan, similar in capability (but different in structure) to Sverre Aarseth's famous NBODY1 (see "Galactic Dynamics," appendix X). The subdirectory `nbody5` contains McMillan's version of Aarseth's NBODY5. Each can operate standalone in a way that would still be recognizable to Sverre. However, with the "`-starlab`" command-line switch set, each can operate as an element in a Starlab pipe (except that they do not understand "story" format on input—the tool " `dumbp` " was written to provide the input conversion). The symbolic links "`nbody1s`" and "`nbody5s`" automatically invoke the programs in "Starlab" mode.

The following is a list of some standalone tools contained in Starlab 1.0. The executables reside in the `bin` directory. (The other executables there are for test purposes.) Here, we list the location of the source code in the Starlab hierarchy, along with a brief description of each tool's purpose.

`node/basetools/rmq.C`	Remove a quantity from a story
`dyn/init/mkcube.C`	Make a uniform cube of particles
`dyn/init/mkplummer.C`	Make a Plummer model
`dyn/init/mksphere.C`	Make a uniform sphere of particles
`dyn/init/mkwrite.C`	Make an N-body system from a text file
`dyn/evolve/arr_hermite.C`	Array-based Hermite integrator
`dyn/evolve/dynamic_hermite.C`	Dynamic Hermite integrator
`dyn/evolve/hermite.C`	Simple Hermite integrator
`dyn/evolve/leapfrog.C`	Leapfrog integrator
`dyn/trafo/freeze.C`	Set velocities to zero
`dyn/trafo/renumber.C`	Renumber particles in the system
`dyn/trafo/scale.C`	Scale the mass, energy, and virial ratio
`dyn/trafo/to_com.C`	Move to the center-of-mass frame
`dyn/reduce/lagrad.C`	Determine Lagrangian radii
`dyn/reduce/lagradplot.C`	Plot Lagrangian radii
`dyn/reduce/molecules.C`	Detect substructure within a system
`dyn/reduce/starplot.C`	Plot particle positions
`dyn/io/dumbp.C`	Convert from story to "dumb" format
`dyn/diag/energy.C`	Print energy of the system
`dyn/oldkepler/splitp.C`	Split particles into binaries
`star/evolve/starhydro_leapfrog.C`	Leapfrog with "sticky" evolving stars
`star/init/addstar.C`	Add star attributes to a system
`hydro/evolve/chydro_leapfrog.C`	Leapfrog with "sticky" core stars
`hydro/evolve/hydro_leapfrog.C`	Leapfrog with "sticky" stars
`hydro/init/addchydro.C`	Add chydro attributes to a system
`hydro/init/addhydro.C`	Add hydro attributes to a system
`hydro/alt/sph_leapfrog.C`	Leapfrog with true (SPH) hydrodynamics

8. Smooth Particle Hydrodynamics

The Starlab 1.0 distribution includes a preliminary version of a fast, low-resolution SPH solver for calculating stellar collisions on the fly during the N-body simulation of a dense cluster.

The SPH code is written in fortran and the interface with starlab and NBODY5 is through a single subroutine that can be called as a black box:

sphcol(m1,r1,x1,y1,z1,vx1,vy1,vz1,m2,r2,x2,y2,z2,vx2,vy2,vz2)

On entry, the arguments contain the values of the two stars' masses, radii, positions, and velocities just before the collision. These initial values must be such that the two stars are on an approaching trajectory. On return, the arguments contain the corresponding values after the collision. A mass of zero indicates complete disruption. The difference between the sum m1+m2 before and after the collision gives the total mass of gas that escaped from the system.

Here is a brief description of the current version:

1) All stars are represented by $n = 1.5$ polytropes with the entropy calculated to obtain the specified mass and radius. The polytropes are constructed by placing SPH particles inside a sphere on a cubic lattice and varying the individual particle masses. No attempt is made to produce a "relaxed" initial condition, which means that the SPH representation of each star is not in strict equilibrium. A constant number density of SPH particles is used for the entire system, so that the number of particles assigned to each star is proportional to its volume. A sufficiently small star may be represented by a single particle (point mass).

2) The SPH code is adapted from Rasio and Shapiro (1991, 1992), but with the calculation of gravitational forces and neighbor lists performed by direct ($\mathcal{O}(N^2)$) summations rather than grid-based techniques. This remains efficient for a total number of SPH particles $N \leq 10^3$.

3) Individual particle smoothing lengths are allowed but not used in the current version. Kernel symmetrization is done using the method of Hernquist and Katz (1989). This feature may be dropped entirely in future versions since it appears to be of marginal benefit in low-N calculations.

4) The integration is terminated when a "stable" final configuration is reached. For a precise definition of what "stable" means in the current version, see the subroutine COMPS in the file sphcol.f. In short, each SPH particle is assigned to one of three components: two bound components representing the two (perturbed) stars and one unbound component representing the escaping gas. The component assignment is done using an iterative scheme that calculates the specific enthalpy of each SPH particle relative to each component's center of mass (cf. Rasio and Shapiro 1991). A "stable" final configuration is obtained when the component assignments are no longer changing in time, and either two bound components remain, receding from each other, or only one (or zero) bound component remains, in which case one of the two stars has been disrupted (or both were). When both stars survive the encounter, their final relative orbit may be elliptical (i.e., a binary forms). The SPH code will return control to the calling N-body code if the orbital period of the binary is sufficiently long. In that case the orbit should be calculated by the N-body code, and the SPH routine called again at the next periastron passage.

5) The final "radius" of each star is calculated as that of a polytrope with the same mass and binding energy. This is not strictly correct, since the specific entropy is in general no longer uniform inside the perturbed star, but produces more reasonable results than any definition based on the actual positions of the outermost particles (because the outer layers may not yet have reached their final hydrostatic equilibrium when the integration terminates). The N-body code should implement the recontraction of the radius back to its thermal equilibrium value on a Kelvin time.

WARNING: *The present version has been debugged, but not yet tested extensively.* Conservation of energy and momentum, and the monotonic increase of entropy in a collision have been checked. A limited number of test calculations have been performed, primarily to make sure that qualitatively reasonable results are obtained for a variety of initial conditions. The code is still in the early stages of development, and the present version is only a snapshot. It has not been optimized in any way (A typical collision with ~ 300 SPH particles per star can be calculated in about 10 minutes on a Sparc 10 workstation).

9. Ownership and Authorship

The primary authors of Starlab 1.0 are Piet Hut, Steve McMillan, and Jun Makino. Other contributors to this initial release are Fred Rasio, whose SPH code is in `src/hydro/alt/fred`, and Simon Portegies-Zwart, whose stellar-evolution integrators may be found in src/usr/spz. We encourage users of Starlab to add their programs and libraries to the package. It is our hope that subsequent releases will contain substantial amounts of user-written material. They can send their code to Piet or Steve, after they have carefully checked that it is working and is compatible with the latest starlab version. We will then install that code in the appropriate `usr/name` directory, from which it may eventually find its way into `src` directory.

The prospect of having several people share (and possibly augment) a software package leads to the thorny question of ownership of software, and, more concretely, authorship of papers written using that software. We want the distribution to be as free as possible, but we must all recognize that Starlab represents a substantial investment of programming time and effort on the part of the contributors. For that reason, while we do not wish to establish hard-and-fast rules, we feel that some guidelines for the use of the package may be helpful.

1. If Starlab is used in a project and is found useful, but not critical to the outcome, then both the package and the authors of the programs used should simply be acknowledged at the end of any resultant papers. A similar acknowledgement in the later papers of a long series would also be appropriate. Examples would be the use of Starlab for data reduction, for organizing series of experiments, or simply for providing an embedding environment for wiring together new pieces of code in a standardized format.
2. Users should try to avoid collision in publications. For example, the SPH code provided here will probably be used soon (i.e. in 1993), by Fred, Steve, and possibly others to do some simulations of small star clusters. If you would like to use the SPH code provided here in order to write an article about the same topic, priority should be given to the author (in this case Fred), so that he can publish his results well before the competing article. In practice, the easiest way to check for collisions is to keep Piet and/or Steve informed of the uses to which the code is being put – something we'd like to know, in any case!
3. If Starlab, or a particular section of Starlab, is an integral part of a piece of research, the author(s) of the code in question should be offered co-authorship on any paper(s) resulting from the research, even if there are no direct collisions. If a series of papers

ensues, we would expect that the author of the software would typically be offered co-authorship on the first one or two. The authorship of any given Starlab program will be prominently displayed near the start of the source file.

4. In case of doubt, ask Piet or Steve.

REFERENCES

Herquist, L., & Katz, N. 1989, ApJS, 70, 419

Hut, 1993, Blue Stragglers as Tracers of Globular Cluster Evolution, I.A.S. preprint, to appear in the proceedings of the workshop on Blue Stragglers, Space Telescope Science Institute, October 1992.

Makino, J., Kokubo, E. & Taiji, M. 1993, to appear in PASJ.

Rasio, F.A., & Shapiro, S.L. 1991, ApJ, 377, 559

Rasio, F.A., & Shapiro, S.L. 1992, ApJ, 401, 226