TREECODE GUIDE

Joshua E. Barnes

Institute for Astronomy, University of Hawai'i,
2680 Woodlawn Drive, Honolulu, Hawai'i 96822

---

treecode is a new program for self-consistent N-body simulation. It is faster than previous codes, and generally provides better error control. This document provides an overview of the algorithm and links to the actual source code.

0. Introduction

Hierarchical force calculation algorithms (*e.g.* Greengard 1990) provide fast, general, and reasonably accurate approximations for gravity and other inverse-square forces. They fill the gap between direct sum methods, which are accurate and general but require O(N^2) operations for a complete force calculation, and field methods, which have limited generality and accuracy but require only O(N) operations. All hierarchical methods partition the mass distribution into a tree structure, where each node of the tree provides a concise description of the matter within some spatial volume. This tree structure is used to introduce explicit approximations into the force calculation. Hierarchical methods require either O(N) or O(N log N) operations per force calculation, depending on the representation employed. The algorithm described here improves on an earlier hierarchical O(N log N) method (Barnes & Hut 1986, hereafter BH86) which has been widely employed in astrophysical simulations.

Below, Section 1 outlines the general strategy of the new code. The data structures used in the tree construction and force calculation are discussed in Section 2. Routines for tree construction, force calculation, control & integration, and input/output are described in Section 3. Instructions for copying, compiling, and running **treecode** are given in Section 4. Finally, system-level software used in the code is described in the Appendix.

1. Strategies

treecode reduces the overhead of tree search by using the fact that neighboring bodies have similar interaction lists. This idea was previously used to speed up hierarchical force calculation on vector machines (Barnes 1990), but the earlier code simply performed a tree search for some small volume defined by a single cell. In the new code this idea is applied to all levels of the tree.

Forces on bodies are computed during a single recursive scan of the entire tree. This scan maintains and continuously updates an *interaction list*; at each level, the fact that a body `b` lies somewhere within a cell `c` is used to winnow the set of possible interactions that `b` might have. (The information used to define interaction lists is thus similar to that used in an early parallel code (Barnes 1986), and a similar strategy figures in the Fast Multipole Method (Greengard & Rokhlin 1987).) The cost of this winnowing is spread over all bodies within `c`, promising a significant speed-up over codes which construct a separate

interaction list for each body (*e.g.* BH86). When the recursive scan arrives at a body `b`, the interaction list on hand is used to obtain the gravitational force and potential at `b`.

Typical hierarchical N-body codes spend about half their time searching the tree (Hernquist 1987); thus one might hope that a factor of two could be gained by quickly generating interaction lists for all bodies in a single scan of the tree. In practice, since the interaction lists so generated are not precisely tailor-made for individual bodies, they must be somewhat longer for a given accuracy. Thus to obtain the expected speed-up requires faster evaluation of interactions, achieved largely via tighter coding. Preliminary timing tests show that **treecode** is indeed about a factor of two faster than the BH86 code *at a fixed level of accuracy*; it also outperforms other versions of that algorithm (Barnes 1998). Moreover, it is significantly more robust when presented with 'perverse' mass distributions (Salmon & Warren 1994).

The new algorithm should lend itself to efficient use of vector processors. On a scalar machine, it spends 90% to 95% of the time summing up interaction lists; a large speed-up would be realized by offloading this task to a vector unit. This approach can make efficient use of N-body force calculation hardware such as the GRAPE (Gravity PIpe) processor (Sugimoto *et al.* 1990).

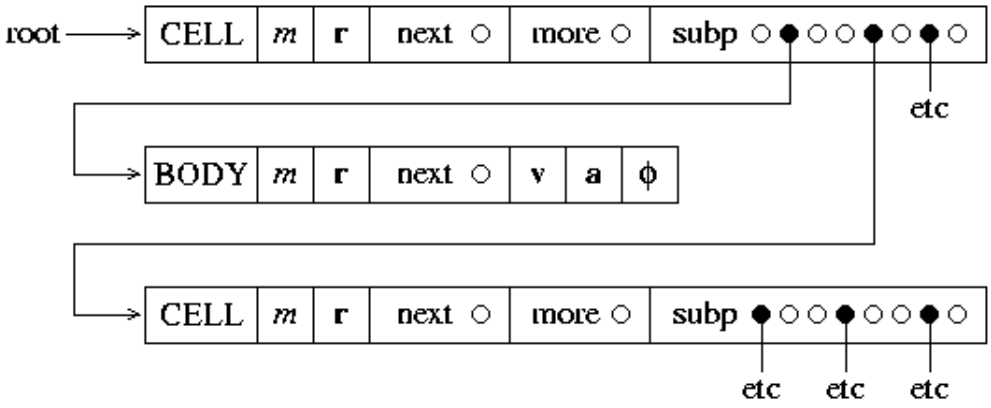
Parallel implementation should also be quite straight-forward provided that each processor has enough memory to hold a copy of the entire particle array and tree structure. At a minimum, the task of summing each interaction list can be distributed across a small number of processors; the bottleneck is then that each processor must search the entire tree. A better approach is to order bodies as they would be encountered in a tree-walk, estimate the computational work required to calculate the force on each, and give each processor the job of computing forces for a contiguous block of bodies.

2. Data Structures

The file `treedefs.h` has specifications for the data structures and global variables used in tree construction and force calculation. This file requires definitions from several general-purpose C include files, which are further described in the Appendix.

2.1. Tree Structure

The primary data structure used in the code is an eight-way tree (or 'oct-tree'), composed of *bodies* and *cells*. Bodies are the leaves of the tree, while cells are the internal branches. The entire tree can be reached starting from a single cell, known as the *root*. A simple tree is shown here:



node structures contain the information common to both bodies and cells. In principle, each element of the tree could be represented as a `union` of a body and a cell, but this would be inefficient since bodies and cells require different amounts of memory. Instead, a node is a common header for bodies and cells, and casts are used to convert a pointer of arbitrary type into a pointer to a node, cell or a body. All of this ugly code is hidden in macros; the macros used to access the components of nodes are:

- **Type(q)** is the actual type of node `q`; the only possible values are the constants `BODY` and `CELL`.
- **Update(q)** is a boolean value which governs the scope of force calculation. Bodies with `Update` equal to `TRUE` get updated forces; cells with `Update` equal to `TRUE` are searched for bodies needing updated forces.
- **Next(q)** is a pointer to the next node encountered in a recursive scan of the tree *after* `q`'s descendents (if any) have been visited.
- **Mass(q)** is the mass of a body, or the total mass of all bodies within a cell.
- **Pos(q)** is the position of a body, or the position of the center of mass of all bodies within a cell.

body structures represent particles. The macros used to access the components of bodies, besides those for nodes, are:

- **Vel(b)** is the velocity of body `b`.
- **Acc(b)** is the acceleration of body `b`.
- **Phi(b)** is the potential of body `b`.

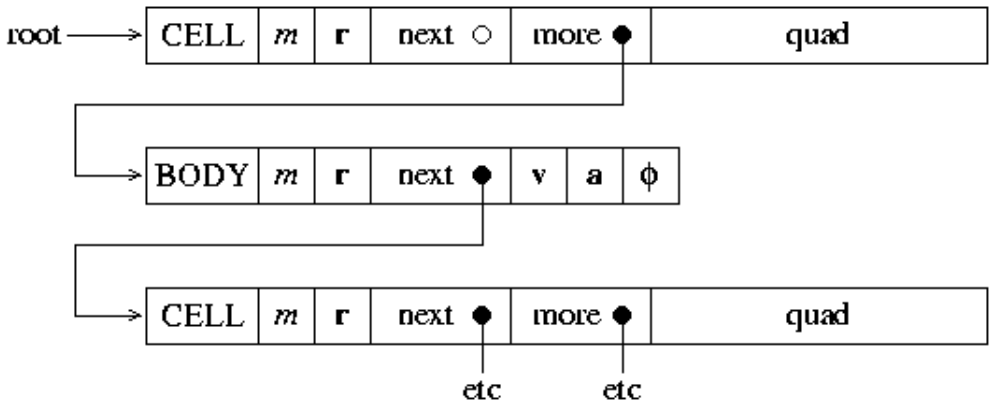
cell structures represent the eight-way internal branchings of the tree. The macros used to access the components of cells, besides those for nodes, are:

- **Subp(c)** is an array of pointers to the descendents of cell `c`.
- **More(c)** is a pointer to the first of these descendents.
- **Quad(c)** is a matrix of quadrupole moments.
- **Rcrit2(c)** is the square of the critical radius beyond which the cell `c` can safely appear in interaction lists. This is *not* defined if the `QUICKSCAN` version is compiled.

(Note that the `Subp` and `Quad` fields share memory; thus only one of each is defined at any point in the calculation.)

The `Next` and `More` fields of the tree are initialized in the second phase of tree construction. In this

phase, the structure is ‘threaded’ in such a way that a tree search can be performed by a simple iterative procedure. This transforms the tree sketched above into the following:



In essence, the `More` link is followed to get more detailed information on the mass distribution, and the `Next` link is followed to move on to the next part of the tree. Threading was originally done to speed up force calculation; now, however, it’s used to free up the memory previously used to store the `Subp` array.

Before threading, the standard idiom to iterate over the immediate descendents of a `cellptr p` is

```
int i;
for (i = 0; i < NSUB; i++)
    if (Subp(p)[i] != NULL)
        <process Subp(p)[i]>;
```

where `NSUB` = 2^3 is the maximum number of direct descendents, while after threading, the idiom is

```
nodeptr q;
for (q = More(p); q != Next(p); q = Next(q))
    <process q>;
```

2.2. Global Variables

Global variables are declared with the symbol **global**, introduced to deal with ANSI C’s requirement that the `extern` keyword be used in all but one compilation module. In compiling the tree construction, force calculation, and input/output routines, `global` expands to `extern`; in the main module `treecode.c`, the `global` symbol is predefined before `treedefs.h` is included:

```
#define global  /* don't default to extern  */
```

This definition prevents `global` from expanding to `extern` when compiling the main module.

a) Input parameters for tree construction and force calculation are as follows:

- **theta** governs the accuracy of the force calculation. This parameter is *not* defined if the `QUICKSCAN` version is compiled.
- **options** is a string listing various run-time control options.
- **usequad** is a flag governing the use of quadrupole corrections.

b) Tree construction assigns values to the following variables:

- **root** is a pointer to the root cell.
- **rsize** is the linear size of the root cell.
- **ncell** counts the number of cells used to build the tree.
- **tdepth** counts the number of levels in the tree below the root.
- **cputree** is the CPU time required for tree construction.

c) Force calculation assigns values to the following variables:

- **actmax** is the maximum length of the active list during force calculation.
- **nbbcalc** is the total number of interactions between bodies and bodies.
- **nbccalc** is the total number of interactions between bodies and cells.
- **cpuforce** is the CPU time required for force calculation.

3. Routines

The routines implementing **treecode** are grouped into four categories: tree construction, force calculation, control & integration, and input/output. Each set of routines is defined in a separate file.

3.1. Tree Construction

The tree construction task is handled by the routines in `treeload.c`. Construction begins by fitting a cube, of linear dimension `rsize`, around the body positions. This cube is identified with the `root` cell of the tree. Bodies are then loaded into the tree one at a time. A given cell can hold up to eight bodies if each one happens to fall in a different octant. Whenever two bodies fall in the *same* octant, a new cell is allocated and the tree is extended to another level. Once all bodies are loaded, total mass and center-of-mass information is propagated from the bodies towards the root.

maketree supervises the process of tree construction. Its prototype is

```
void maketree(bodyptr btab, int nbody);
```

where `btab` is an array of `nbody` body structures. When `maketree` returns, the root of the tree is addressed by the global cell pointer, `root`; also updated are the root cell size, `rsize`, the number of cells in the tree, `ncell`, the depth of the tree, `tdepth` (counting from `0`), and the CPU time used, `cputree`.

After initializing an empty `root` cell, `maketree` loops over `btab`, calling `loadbody` to install each body in the tree. It then propagates information from the leaves to the root, threads the tree structure, and optionally computes quadrupole moments.

newtree scavenges the cells in the existing tree and prepares to build a new one. Its prototype is

```
void newtree(void);
```

makecell returns a pointer to a free cell. Its prototype is

```
cellptr makecell(void);
```

In addition, `makecell` also updates `ncell`.

expandbox fits a box around the body distribution. Its prototype is

```
void expandbox(bodyptr btab, int nbody);
```

where `btab` is an array of `nbody` body structures. The result is stored in `rsize`.

To take advantage of the exact floating-point representation of powers of two in binary computers, the box size is successively doubled until it fits the bodies. This insures that the corners and midpoints of cells have exact binary representations.

loadbody inserts a body in the tree. Its prototype is

```
void loadbody(bodyptr p);
```

where `p` is the body in question.

To find the appropriate place to insert the body, `loadbody` traces out a path from the `root` node toward the leaves of the tree. At each level, it calls `subindex` to decide which branch to take. Eventually, one of two things happens. First, the indicated subcell may be empty; the body `p` is then stuffed into the empty slot. Second, the indicated subcell may already hold a body; then a new cell is allocated to extend the tree and both bodies are installed within the new cell.

subindex decides which of a cell's octants a body falls in. Its prototype is

```
int subindex(bodyptr p, cellptr q);
```

where `p` is the body in question and `c` is the cell.

The `subindex` function uses the fact that during the first phase of tree construction the `Pos` vector of a cell is its geometric midpoint. Note that it's assumed that `p` actually lies within the volume represented by `c`; this assumption is checked for the entire tree when `hackcofm` is called.

hackcofm propagates cumulative information towards the `root` cell. Its prototype is

```
void hackcofm(cellptr p, real psize, int lev);
```

where `p` is the current cell, `psize` is its linear size, and `lev` is its level in the tree, counting the `root` as level `0`.

In outline, `hackcofm` is a simple depth-first recursive tree scan; it loops over the descendents of `p` and calls itself on those which point to cells. It thereby accumulates total masses, center-of-mass positions, and update flags. This done, `hackcofm` checks that the center-of-mass actually lies in the volume represented by `p`; failure indicates a bug in tree construction. Optionally, `setrcrit` is called to set the critical opening radius for this cell.

setrcrit assigns each cell a critical squared opening radius. Its prototype is

```
void setrcrit(cellptr p, vector cmpos, real psize);
```

where p is the cell, cmpos is its center-of-mass position, and psize is its size.

This routine is defined only if the QUICKSCAN version is *not* compiled. It offers several different methods to compute the squared critical radius, Rcrit2, beyond which a force calculation need not open p. The default is a criterion which is extra careful with cells having relatively off-center mass distributions (Barnes 1995); also provided are Salmon & Warren's (1994) *bmax* criterion (option sw94) and BH86's original criterion (option bh86).

threadtree relinks the tree using the Next and More pointers. Its prototype is

```
void threadtree(nodeptr p, nodeptr n);
```

where p is any node and n will be its successor.

In outline, threadtree is also a depth-first recursive tree walk. First, it makes n the Next link of p. Then, if p is a cell, it makes a list its descendents and sets p's More link to be the first of these; finally it calls itself on each member of the list, passing the next member as the successor (note that the successor of the last member is the successor of p).

hackquad propagates quadrupole moments towards the root cell. Its prototype is

```
void hackquad(cellptr p);
```

where p is the current cell.

Again, this routine does a depth-first tree walk. The only trick is that the storage used for the Subp pointers is reused to store the quadrupole matrix, so the latter are first copied to a temporary array. (Note: this routine could be simplified using the post-threading idiom to iterate over descendents.)

3.2. Force Calculation

The force calculation code is implemented by routines provided in treegrav.c. As described above, forces are calculated during a single recursive scan of the tree, which visits every body whose Update flag is set. Gravitational forces and potentials are assigned to these bodies.

gravcalc supervises the process of force calculation. Its prototype is

```
void gravcalc(void);
```

The tree structure to be used by gravcalc is addressed by the global root pointer; also referenced are the tree depth tdepth and root cell size rsize.

The main job of gravcalc is to set up the initial call to the worker routine walktree. It begins by allocating temporary storage for several lists; the length of these lists is estimated from the depth of the tree. The interact pointer addresses a linear array of cells which list all the interactions acting on a body; body-cell interactions are listed from the front toward the back, while body-body interactions are listed from the back toward the front. The active pointer addresses an array of node pointers which will be examined when constructing interaction lists. With these arrays in place, gravcalc places the root

node on the active list and calls `walktree` to scan the tree.

walktree is the main recursive routine for force calculation. Its prototype is

```
void walktree(nodeptr *aptr, nodeptr *nptr, cellptr cptr, cellptr bptr,
              nodeptr p, real psize, vector pmid);
```

The effect of `walktree` is to compute gravity on bodies within node `p`. This is accomplished via a recursive scan of `p` and its descendents. At each point in the scan, information from levels between the root and `p` is contained in a set of nodes which will appear on the final interaction lists of all bodies within `p`; this set is split into separate lists of cells and bodies, addressed by `cptr` and `bptr`, respectively. The rest of the tree is represented by a set of active nodes which include node `p` and surround it in space; pointers to these nodes are stored in an array between `aptr` and `nptr`. Node `p` has linear size `psize` and geometric midpoint `pmid`.

In the main loop of the routine, `walktree` examines the active nodes, deciding which may be appended to the interaction lists, and which are so close that their descendents must be examined at the next level of recursion. Cells are tested by the function `accept` to determine if they are sufficiently well-separated from `p`. If so, they are placed on the cell interaction list, headed by `cp`; if not, their *descendents* are placed on the new active list, headed by `np`. Bodies, unless they happen to be the body `p` itself, are placed directly on the body interaction list, which is headed by `bp`.

(It may be worth testing the type of a node before placing it on the new active list; that way, bodies can be copied once and for all to the body interaction list, instead of being copied again for each immediate descendent of `p`. But this will complicate the handling of self-interaction.)

If some active cells were rejected by `accept`, recursion continues to the next level of the tree, taking as active the descendents of the rejected cells. The actual recursive descent is performed by `walksub`. Otherwise, `p` points to a body, and the interaction lists are complete, so `gravsum` may be called to sum up the interactions.

accept determines if a cell passes the separation test. Its prototype is

```
bool accept(nodeptr c, real psize, vector pmid);
```

where `c` is the cell under consideration, and `psize` and `pmid` specify the volume represented by the current node `p`.

Two versions of `accept` exist. The default version insures that the squared distance from `c` to the nearest point within `p` is greater than `Rcrit2(c)`. But if the `QUICKSCAN` version is compiled, a minimal criterion is used; no cell touching `p`, even at a single corner, is accepted.

walksub performs the actual recursive call-back to `walktree`. Its prototype is

```
void walksub(nodeptr *nptr, nodeptr *np, cellptr cptr, cellptr bptr,
             nodeptr p, real psize, vector pmid);
```

and all parameters have exactly the same values that they do at the calling point in `walktree`.

Two possible cases arise in `walksub`. Most often by far, the node `p` is a cell. In this case, `walksub` loops

over its descendents, invoking `walktree` for each, with the appropriately shifted cell center. Much more rarely, `p` is actually a body. In this case, the active list contains nodes which can't be sorted into the interaction lists at the present level; `walksub` calls `walktree` exactly once, continuing the search to the next level by 'virtually' extending the tree.

gravsum supervises the process of evaluating forces from the cell and body interaction lists. Its prototype is

```
void gravsum(bodyptr p0, cellptr cptr, cellptr bptr);
```

where `p0` is the body requiring updated forces and `cptr` and `bptr` are pointers to the cell and body interaction lists, respectively.

The main job of `gravsum` is to call the worker routines `sumnode` and `sumcell` to sum up interaction terms. The latter routine is only invoked if quadrupole moments are included.

sumnode sums up interactions without quadrupole corrections. Its prototype is

```
void sumnode(cellptr start, cellptr finish,
             vector pos0, real *phi0, vector acc0);
```

where `start` and `finish` point to the front and back of the interaction list, `pos0` is the place where the force is evaluated, and `phi0` and `acc0` are the resulting potential and acceleration.

sumcell sums up interactions, including quadrupole corrections. Its prototype is

```
void sumcell(cellptr start, cellptr finish,
             vector pos0, real *phi0, vector acc0);
```

where the arguments have the same meanings they do for `sumnode`.

This routine is similar to `sumnode`, but includes quadrupole-moment corrections (Hernquist 1987) to improve the forces and potentials generated by cells.

NOTE: `sumnode` and `sumcell` together account for upwards of 90% of the cycles in typical calculations; optimizing their performance is critical. Under the IBM AIX compiler, better results are obtained when intermediate variables are stored in `double` precision. The contrary is the case on other systems which have been tested (SGI & MIPS compiler, Sun & gcc).

3.3. Control & Integration

The routines governing the N-body integration are collected in `treecode.c`. Parameters and state variables associated with control and integration are defined in `treecode.h`.

main is the main controlling routine. Its prototype is

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

treeforce is the supervisory routine for force calculation. Its prototype is

```
void treeforce(void);
```

stepsystem advances the system by one time-step. Its prototype is

```
void stepsystem(void);
```

Integration is performed using a ‘synchronized leap-frog’, which is computationally equivalent to the traditional time-centered leap-frog but retains the synchronization of positions and velocities. The formula to advance the positions **r** and velocities **v** from step n to step n+1 is:

$$\mathbf{v}_{n+1/2} = \mathbf{v}_n + \mathbf{a}_n / 2f ,$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \mathbf{v}_{n+1/2} / f ,$$

$$\mathbf{v}_{n+1} = \mathbf{v}_{n+1/2} + \mathbf{a}_{n+1} / 2f ,$$

where **a** = **a**(**r**) is the acceleration computed from positions at the corresponding step, and f is the *integration frequency*, equal to the inverse of the time-step.

startrun initializes parameters and data for the N-body run. Its prototype is

```
void startrun(void);
```

testdata sets up a Plummer model (Aarseth, Henon, & Wielen 1974). Its prototype is

```
void testdata(void);
```

3.4. Input/Output

The routines responsible for input & output of N-body data are in `treeio.c`.

inputdata reads initial conditions from an input file. Its prototype is

```
void inputdata(void);
```

startoutput prints a header describing the calculation. Its prototype is

```
void startoutput(void);
```

forcereport prints statistics on tree construction and force calculation. Its prototype is

```
void forcereport(void);
```

output prints diagnostics and determines if a data output is required. Its prototype is

```
void output(void);
```

outputdata outputs the actual N-body data. Its prototype is

```
void outputdata(void);
```

diagnostics computes various dynamical diagnostics. Its prototype is

```
local void diagnostics(void);
```

savestate writes current program state to a binary file. Its prototype is

```
void savestate(string pattern);
```

restorestate restores the program state from a binary file. Its prototype is

```
void restorestate(string file);
```

4. Instructions

4.1. Distribution

A complete version of **treecode** may be down-loaded from this web site. The entire code is available as a single gzipped tar file:

- `treecode.tar.gz`

Alternatively, individual files may be saved as plain text (with the appropriate `.c` and `.h` extensions). The files required are `treecode.h`, `treedefs.h`, `treecode.c`, `treegrav.c`, `treeio.c`, and `treeload.c`. Some general-purpose include files and library routines are also needed; these are found in `getparam.h`, `mathfns.h`, `stdinc.h`, `vectdefs.h`, `vectmath.h`, `clib.c`, `getparam.c`, and `mathfns.c`. A `Makefile` is also provided to help organize the compilation process.

The **treecode** sources are provided as free software with no restrictions on their use or modification. You may redistribute this software at will provided that you do so without restricting its use, modification, or redistribution by others. See the GNU General Public License for details.

This software is provided ‘as is’; no warranty is expressed or implied. Naturally, I have run extensive tests of this software before releasing it; just as naturally, there are probably bugs or limitations not identified in these tests. If you encounter problems in using this software, please let me know!

If you publish results obtained using **treecode**, I would appreciate a brief acknowledgment (and a preprint or reprint). I will not be able to improve this software and continue making it available without adequate research support; acknowledging the use of this software is an investment in future developments as well as a basic courtesy.

4.2. Compilation

treecode is written in ANSI C; while some I/O operations may depend on the operating system, the bulk of the code should work on any system with an ANSI C compiler. The instructions below assume that a UNIX-like operating system is available.

Begin by placing the files in a single directory. If you have copied the `treecode.tar.gz` file, this may be done by giving the following commands:

```
% gunzip treecode.tar.gz
```

```
% tar xvf treecode.tar
```

The directory should then contain all of the `.h` and `.c` files listed above, as well as the `Makefile`.

Before actually compiling the code, you may need to edit the `Makefile` and modify some of the parameters or switches. The defaults provided produce good code on an SGI machine with a MIPS compiler; suggestions for other operating system/compiler combinations will be included in the `Makefile`.

Both single-precision and double-precision versions of the code can be generated by changing the value of the `PRECISION` parameter in the `Makefile`. Single-precision is good enough for most calculations, so that is the default. Some run-time libraries, however, do not provide single-precision versions of floating-point functions. On such systems, use the `MIXEDPREC` option for best performance. Note that if you change the `PRECISION` parameter, you must delete any existing `.o` and `.a` files before recompiling.

Once the `Makefile` has been edited, build a version of the code by giving either of these commands:

```
% make treecode
% make treecode_q
```

The first of these builds the standard version, while the second builds the `QUICKSCAN` version.

4.3. Operation

A test calculation using a self-generated Plummer model (Aarseth, Henon, & Wielen 1974) may be run by giving the either of these commands

```
% treecode
% treecode_q
```

This calculation uses `nbody=4096` bodies, an integration frequency (inverse timestep) of `freq=32`, a smoothing length of `eps=0.025`, an accuracy parameter `theta=1.0` (for the standard version), and no quadrupole moments. A log of the calculation is printed out; no other output is generated. On a SGI O2, this test calculation takes about 2.5 minutes.

treecode uses the `getparam` command-line package to get all input parameters, including the names of any input and output files. To see a full list of parameters, along with their default values and descriptions, give either of these commands:

```
% treecode -help
% treecode_q -help
```

In response, the standard **treecode** prints out the following:

```
treecode                        Hierarchical N-body code (full scan)
  in=                           Input file with initial conditions
  out=                          Output file of N-body frames
  freq=32.0                     Fundamental integration frequency
  eps=0.025                     Density smoothing length
  theta=1.0                     Force accuracy parameter
  usequad=false                 If true, use quad moments
  options=                      Various control options
```

```
tstop=2.0                       Time to stop integration
freqout=4.0                     Data output frequency
nbody=4096                      Number of bodies for test run
seed=123                        Random number seed for test run
save=                           Write state file as code runs
restore=                        Continue run from state file
VERSION=1.3                     Joshua Barnes  January 10 1999
```

(the `QUICKSCAN` version produces a similar listing with one less parameter). This printout lists the name of each parameter, its default value if any, and a brief explanation of its function. The `getparam` package accepts argument values on the command line, and matches them to parameters either by position or by name. Initially, positional matching is used: the first argument is matched to the first parameter, and so on. However, if any argument has the form *name=value*, that argument and any that follow it are matched by name. An error results if a *name* does not match any parameter or if more than one value is assigned to a parameter.

At somewhat greater length than above, the parameters accepted by **treecode** are interpreted as follows:

- **in**, if given, names an input file containing initial conditions for the calculation. The format of this file is described below.
- **out**, if given, is a pattern naming output files for N-body snapshots taken at regular intervals. This pattern is used as an argument to `sprintf`, along with the integration step number, to generate an actual file name. If the resulting file already exists, the new data is appended. The format used is similar to the format used for input files.
- **freq** is the inverse of the integration time-step. It's convenient to take an integer number of steps per unit time, so a frequency is a more natural than parameter than a time increment. Powers of two are particularly convenient since times are then represented exactly as floating-point numbers. If `freq=0`, **treecode** does a single force calculation and output, and exits.
- **eps** is the smoothing length used in the gravitational force calculation. In effect, the mass distribution is smoothed by replacing each body by a Plummer sphere with scale length `eps`, and the gravitational field of this smoothed distribution is calculated.
- **theta** is the *opening angle* used to adjust the accuracy of the force calculation. Values less than unity produce more accurate forces, albeit at greater computational expense. The `QUICKSCAN` version does not use or accept this parameter.
- **usequad** determines if quadrupole corrections are used when calculating gravitational fields. These corrections can significantly improve the accuracy of force calculation at a fairly modest computational cost.
- **options** is a list of comma-separated key words used to obtain various run-time options. The options available are
  - **reset-time**: set the time to zero, regardless of the value in the input file;
  - **new-tout**: reschedule the first snapshot output;
  - **out-phi**: include potential values in the output files;
  - **out-acc**: include acceleration vectors in the output files;
  - **bh86**: use the BH86 criterion to set critical cell radii;
  - **sw94**: use Salmon & Warren's (1994) *bmax* criterion to set critical cell radii.
- **tstop** is the time at which the N-Body integration terminates.
- **freqout** is the inverse of the time interval between output files. To insure that outputs are performed when expected, `freqout` should divide `freq`.
- **nbody** is the number of bodies used to self-generate initial conditions. This parameter is only used

if no input or restore file is given.

- **seed** is the random number seed used to self-generate initial conditions. This parameter is only used if no input or restore file is given.
- **save**, if given, is a pattern naming binary files used to record the state of the code after each step. This pattern is used as an argument to `sprintf`, along with the lowest bit of the step number, to construct the actual file name.
- **restore** names a binary file written using the `save` function above. The calculation resumes from that point. New values of the `eps`, `theta`, `usequad`, `options`, `tstop`, and `freqout` parameters may be supplied; if not, the values from the saved calculation are used. If N-body data outputs are required, a new value for `out` must be given.

For example, to run a test calculation using a Plummer model with `32768` bodies and an opening angle of `0.75`, type

```
% treecode nbody=32768 theta=0.75
```

To compute forces for a N-body configuration in the input file `input.data`, writing the result to `forces.data`, type

```
% treecode input.data forces.data freq=0 options=out-phi,out-acc
```

To run the initial conditions in `input.data` with a timestep of `1/64`, writing results at intervals of `1/16` to separate files `run_000.data`, `run_004.data`, ..., type

```
% treecode input.data run_%03d.data freq=64 freqout=16
```

To perform the same calculation using the `QUICKSCAN` version, while saving the program's state after each step, type

```
% treecode_q input.data run_%03d.data freq=64 freqout=16 save=state.data
```

To continue this calculation until time `4`, type

```
% treecode_q restore=state.data out=run_%03d.data tstop=4
```

4.4. File Formats

By default, the input and output files used by **treecode** are written in ASCII. Each file contains one or more snapshots. Input files have the following format:

```
nbody
NDIM
time
mass[1]
....
mass[n]
x[1] y[1] z[1]
....
x[n] y[n] z[n]
vx[1] vy[1] vz[1]
....
vx[n] vy[n] vz[n]
```

Thus an input file contains a total of `3+3*nbody` lines. A similar format is used for output files. If the `out-phi` and/or `out-acc` options are set, velocities are followed by potential values and/or acceleration vectors, respectively.

While ASCII output is easy for people to read, it is relatively inefficient. If `treeio.c` is compiled with the `BINARYIO` preprocessor symbol defined, input and output are performed in binary. This is recommended for production calculations!

Thanks

The bucolic scenery of Leiden inspired the invention of this algorithm and I thank Tim de Zeeuw for his hospitality. I'm also grateful to Jun Makino for hospitality while I developed the public version described here. Initial development of **treecode** was supported by NASA grant NAG-2836.

Appendix. Zeno System Software

The **treecode** software depends on a number of include files and function libraries, collectively known as the 'Zeno System'. These files define keywords, data types, and library routines useful in C programs.

A.1. Standard Include File

stdinc.h is a standard include file. It defines some constants and data types, and provides prototypes for a few functions from a general-purpose C library. The following constants and types are used in **treecode**.

- **NULL** indicates a pointer to no object. This definition is identical to the one in C's I/O include file `stdio.h`; it is included here for completeness.
- **local** is a synonym for `static`, used to indicate that a file-level data object or routine is defined only within that file.
- **bool** is a storage type for logical values. Also defined are the constants `TRUE` and `FALSE`.
- **string** is a storage type for a pointer to a null-terminated sequence of characters.
- **real** is a storage type for real-valued numbers. The precision of `real` values is fixed at compile time; `real` is synonymous to `float` if either `SINGLEPREC` or `MIXEDPREC` is defined, and to `double` if `DOUBLEPREC` is defined. For more details, see the description of `mathfns.h` below.

The following functions, used in **treecode**, have prototypes in `stdinc.h`; their source code is given in clib.c.

allocate is a memory-allocation function. Its prototype is

```
void *allocate(int nbyte);
```

where `nbyte` is the number of bytes to allocate. The `allocate` function exits via `error` if it can't get the requested amount of memory. The memory is cleared before `allocate` returns.

cputime returns the total process CPU time in minutes. Its prototype is

```
double cputime(void);
```

error reports an error and exits. Its prototype is

```
void error(string fmt, ...);
```

The `fmt` string and any further arguments are interpreted exactly like the system routine `printf`, but output to `stderr` instead of `stdout`.

scanopt scans an option string for a keyword. Its prototype is

```
bool scanopt(string opts, string word);
```

where `opts` is a series of words separated by commas, and `word` is a single keyword; it returns `TRUE` if `word` appears in `opts`, and `FALSE` otherwise.

stropen opens a `stdio.h` stream for input/output. Its prototype is

```
stream stropen(string file, string mode);
```

A.2. Real Functions

mathfns.h defines `real` versions of the math functions in `math.h`, along with some extensions. Most function name are derived from their counterparts in `math.h` by prepending the letter '`r`' (for `real`). The following functions are used by **treecode**; sources appear in `mathfns.c`.

rsqrt computes a real-valued square root. Its prototype is

```
real rsqrt(real x);
```

rsqr computes the inverse of `rsqrt`. Its prototype is

```
real rsqr(real x);
```

rpow computes a real-valued power. Its prototype is

```
real rpow(real x, real y);
```

rabs computes a real absolute value. Its prototype is

```
real rabs(real x);
```

xrandom generates a random number between specified limits. Its prototype is

```
double xrandom(double x1, double x2);
```

pickshell picks a point on the surface of a shell in an n-dimensional space. Its prototype is

```
void pickshell(real *point, int ndim, real radius);
```

A.3. Getparam Package

getparam.h defines a general command-line argument processing package. Source code is given in `getparam.c`. This package provides the client program with a simple self-documenting user interface. The user may specify arguments by position or by keyword; defaults are supplied for all arguments not specified. Used in **treecode** are the following.

initparam initializes the command-line processing package. Its prototype is

```
void initparam(string argv[], string defv[]);
```

where `argv` is the `NULL`-terminated vector of command-line arguments supplied to `main`, and `defv` is a vector of parameter names, default values, and documentation messages.

getparam returns the value of a parameter. Its prototype is

```
string getparam(string name);
```

where `name` is the name of the parameter as supplied to `initparam`. An error occurs if `name` is not found.

getiparam returns the integer value of a parameter. Its prototype is

```
int getiparam(string name);
```

getdparam returns the double-precision floating-point value of a parameter. Its prototype is

```
double getdparam(string name);
```

getbparam returns the boolean value of a parameter. Its prototype is

```
bool getbparam(string name);
```

getparamstat returns an indication of parameter status. Its prototype is

```
int getparamstat(string name);
```

A.4. Vectors and Matrices

vectdefs.h defines vectors and matrices. The number of dimensions is a compile-time constant, specified by the preprocessor symbols `THREEDIM`, `TWODIM`, and `NDIM`; the default is `THREEDIM`. The types defined are as follows.

- **vector** is a storage type for a vector of `NDIM` `real` values.
- **matrix** is a storage type for a matrix of `NDIM`^{2} `real` values.

vectmath.h defines a collection of macros for operating on vectors and matrices. Those used by **treecode** are listed below.

- **ABSV(*s*,*v*)** computes the magnitude of a vector: $s = sqrt(v_i\ v_i)$.
- **ADDM(p,q,r)** adds matrices: $p_{ij} = q_{ij} + r_{ij}$.
- **ADDV(u,v,w)** adds vectors: $u_i = v_i + w_i$.

- **CLRM(p)** sets the elements of a matrix to zero: $p_{ij} = 0$.
- **CLRV(v)** sets the elements of a vector to zero: $v_i = 0$.
- **DISTV(*s*,u,v)** computes the distance between two vectors: $s = sqrt((v_i - u_i)(v_i - u_i))$.
- **DIVVS(v,u,*s*)** divides a vector by a scalar: $v_i = u_i / s$.
- **DOTVP(*s*,v,u)** forms the dot product of two vectors: $s = v_i u_i$.
- **MULMS(p,q,*s*)** multiplies a matrix by a scalar: $p_{ij} = q_{ij} s$.
- **MULVS(v,u,*s*)** multiplies a vector by a scalar: $v_i = u_i s$.
- **OUTVP(p,v,u)** forms the outer product of two vectors: $p_{ij} = v_i u_j$.
- **SETM(p,q)** sets one matrix equal to another: $p_{ij} = q_{ij}$.
- **SETV(v,u)** sets vector equal to another: $v_i = u_i$.
- **SETVS(v,*s*)** sets all components of a vector equal to a scalar: $v_i = s$.
- **SETMI(p)** sets a matrix to the identity: $p_{ij} = delta_{ij}$.
- **SUBM(p,q,r)** subtracts matrices: $p_{ij} = q_{ij} - r_{ij}$.
- **SUBV(v,u,w)** subtracts vectors: $v_i = u_i - w_i$.

There are also some macros defining compound operations specific to **treecode**. These are written so as to help the compiler produce good code in the force summation loops.

- **ADDMULVS(v,u,*s*)** scales a vector, and adds the result to another: $v_i = v_i + u_i s$.
- **ADDMULVS2(v,u,*s*,w,*r*)** scales two vectors, and adds the result to a third: $v_i = v_i + u_i s + w_i r$.
- **DOTPMULMV(*s*,v,p,u)** multiplies a vector by a matrix, and also forms the dot product: $v_j = p_{ij} u_i$ and $s = v_j v_j$
- **DOTPSUBV(*s*,v,u,w)** subtracts two vectors, and forms the dot product: $v_i = u_i - w_i$ and $s = v_i v_i$.

References

- Aarseth, S.J., Henon, M., & Wielen, R. 1974 *Astr. & Ap.* **37**, 183.
- Barnes, J.E. 1986. In *The Use of Supercomputers in Stellar Dynamics*, eds. P. Hut and S. McMillan (Berlin: Springer-Verlag), p. 175.
- Barnes, J.E. 1990. *J. Comp. Phys.* **87**, 161.
- Barnes, J.E. 1995. In *The Formation of Galaxies*, eds. C. Munoz-Tunon & F. Sanchez (Cambridge: Cambridge University Press), p. 399.
- Barnes, J.E. 1998. *Dynamics of Galaxy Interactions*, in *Galaxies: Interactions and Induced Star formation*, by R.C. Kennicutt, Jr., F. Schweizer, & J.E. Barnes (Berlin: Springer).
- Barnes, J. & Hut, P. 1986. *Nature* **324**, 446.
- Greengard, L. 1990. *Computers in Physics*, **4**, 142.
- Greengard, L. & Rokhlin, V. 1987. *J. Comp. Phys.* **73**, 325.
- Hernquist, L. 1987. *Ap. J. Suppl.* **64**, 715.
- Salmon, J.K. & Warren, M.S. 1994. *J. Comp. Phys.* **111**, 136 .

- Sugimoto, D. *et al.* 1990. *Nature* **345**, 33.

Joshua E. Barnes (barnes@galileo.ifa.hawaii.edu)

Last modified: June 12, 1999