TreeCode Guide

Noah Muldavin

July 19, 2013

Introduction

TreeCode is an N-Body simulation code in C++ which can integrate up to 10^7 particles. It uses an adaptive-timestep leapfrog integration scheme and Barnes-Hut treecode force calculation. As a default it is designed for collisionless gravitational N-Body simulations but it can be easily modified to simulate any system of particles for which the interaction potential yields a valid multipole expansion (monopole, dipole, quadripole, etc). I've divided the discussion below into three sections. The first section (Use) is designed for people who don't need to know how the code is working, but just want to use it to integrate an N-Body system. It covers the basic operation of the code, how to input initial conditions, and how to visualize results. The second section (Algorithm) is designed for those who want to know how the code works on an Algorithmic level. When I was attempting to learn about treecodes it was nearly impossible to find a source which explained how they work on a detailed level. My hope is that this section can be used as a resource for future students so that they might save themselves from a similar plight. The third section (Code) contains a detailed description of every single piece of code and what it's doing. I hope that it can be used as a resource (in conjunction with the algorithmic description in section 2) for those who want to learn how to *code* an N-Body simulation in C++, or at least for those who want to rip apart some section of my code.

1 Use

1.1 Compiling and Running

From the appropriate directory, TreeCode may be compiled with the terminal command `make -f nbd.make` and run with `./nbody`. It is assumed that you have installed the GNU C++ compiler, which can be obtained for free as part of the apple XCode developer package.

1.2 Initial Condition Input

TreeCode takes as an input a `.dat` file in the following format:

```
10000 5000
+5.0000000e-06 -1.9532973e+00 -7.5361489e-01 -3.5929412e-02  -2.9562719e-01 -8.4945238e-01 +3.9108198e-01  0
+5.0000000e-06 +1.8760459e+00 +2.8772161e+00 -9.3745588e-02  -1.2027098e+00 +1.6745953e-01 -1.0220085e-01  +1
+5.0000000e-06 -5.5344013e-01 -1.1544420e+00 -2.6506558e-01  +1.8278606e-01 +2.8373649e-01 +1.9911002e-01  0
+5.0000000e-06 -2.0775973e+00 +2.8911833e+00 -3.7780287e-01  -7.1219669e-01 -7.1631953e-01 +2.4480246e-02  +1
+5.0000000e-06 +3.6566111e+00 +8.0957525e-02 +3.8605589e-02  -3.2895574e-01 +9.5089765e-01 +4.0516959e-01  0
+5.0000000e-06 -1.5976431e+00 -4.5264739e-02 +2.2124655e-01  +1.2086866e+00 +9.9085360e-02 +2.4473230e-01  +1
+5.0000000e-06 +1.6796982e+00 +1.0504018e+00 -3.7409262e-02  +2.6707945e-01 +9.1884292e-01 -2.2742799e-01  0
+5.0000000e-06 +1.3630670e+00 +6.4176187e-01 -2.7841822e-01  -8.6888066e-01 +8.4063024e-01 -3.7819531e-01  +1
+5.0000000e-06 +4.0578158e-01 -2.6138517e-02 +1.9712671e-01  +4.4952445e-01 +8.7835439e-01 -5.0281527e-02  0
```

The first row lists the total number of particles followed by the total number of dark matter particles (if it doesn't matter just set this to 0). Each subsequent row contains the mass m, the initial position (x, y, z) and velocity (v_x, v_y, v_z) of each particle followed by a boolean value indicating if that particle is dark matter or not (1 if dark, 0 if not). The function `writeinitfile()` in `dataio.hpp` will write a file in this format given an array of `bodynode` data structures. You may want to use this function (with `#include "dataio.hpp"`) in the routine you use to build your initial conditions. I've included a sample startfile with the code.

1.3 Parameters

In order for the code to function as desired, you should edit the following parameters located in a box at the top of the `main.cpp` file:

- `startfile`: The location of the `.dat` file containing the initial conditions.
- `datadir`: The location of the folder in which you want the program to save data. Make sure the folder exists before you run, or you'll get an error.
- `dtmax`: The *maximum* timestep over which a particle can be integrated. It doubles as the frequency at which the program will save visualization files (i.e. one visualization file will be saved every integer multiple of `dtmax`). You'll want this to be related to some fundamental timescale of the system you're studying.
- `tfinal`: The time at which the integration will stop, *in units of* `dtmax`. For example, if you set `tfinal = 40`, the integration would stop when $t = 40 * \texttt{dtmax}$.
- `alpha`: A parameter controlling the size of the ideal timestep for each mass. If alpha is small, the timesteps will be small. As a default it is set to 0.1 which should be very sufficient.
- `theta`: The Barnes-Hut θ_{crit} parameter controlling the accuracy of the treecode force calculation. As a default it is set to 0.8.
- `renderbox`: The side-length of a rendering-cube in the visualization program. This doesn't really matter much, just make sure it's bigger than your system.
- `plotDM`: A boolean value indicating whether or not dark-matter particles should be plotted.

1.4 The Physics

The code is set up to perform a collisionless graviational N-Body simulation but you may wish to edit it to integrate a different system. Below is a list of some places you might look if you're trying to change "the physics:"

- `gravity()` in `treeforce.cpp`: This function defines the inter-particle gravitational force. You may wish to change this to reflect the physics of the system you're studying.
- `leapfrogvar()` in `integrate.cpp`: This is where the basic leapfrog integration routine is defined. If you wish to add extra forces in addition to the force calculated by the treecode, you can do so in line 74. I would be careful about editing the integration method in lines 75 and 76: unfortunately the variable-timestep integration scheme is tailored specifically for the leapfrog method so you can't just change this to Runge-Kutta without messing things up.

- `eps` in `main.cpp`: As a default the softening length ϵ is assigned in relation to N:

$$\epsilon = 0.98N^{-0.26}.$$

This is based on an optimization study of collisionless gravitational N-Body simulations [4]. You may wish to manually assign ϵ at line 48 of `main()`.

1.5 Accuracy and Timing

The treecode method of force calculation achieves it's dramatic speedup by approximation. Particles are grouped into boxes each with a side-length d. Each box contains 8 sub-boxes with side-length $d/2$. When calculating the force on a single mass, the algorithm tests whether $\theta = d/r < \theta_{\text{crit}}$, where r is the distance from that particle to the center of mass of the relevant box. If so, the force is calculated using that box's center of mass. If not, the acceptability is tested for each sub-box. Below is the result of a timing and accuracy test I ran using this code. You may wish to use these results to choose the optimum accuracy parameters.

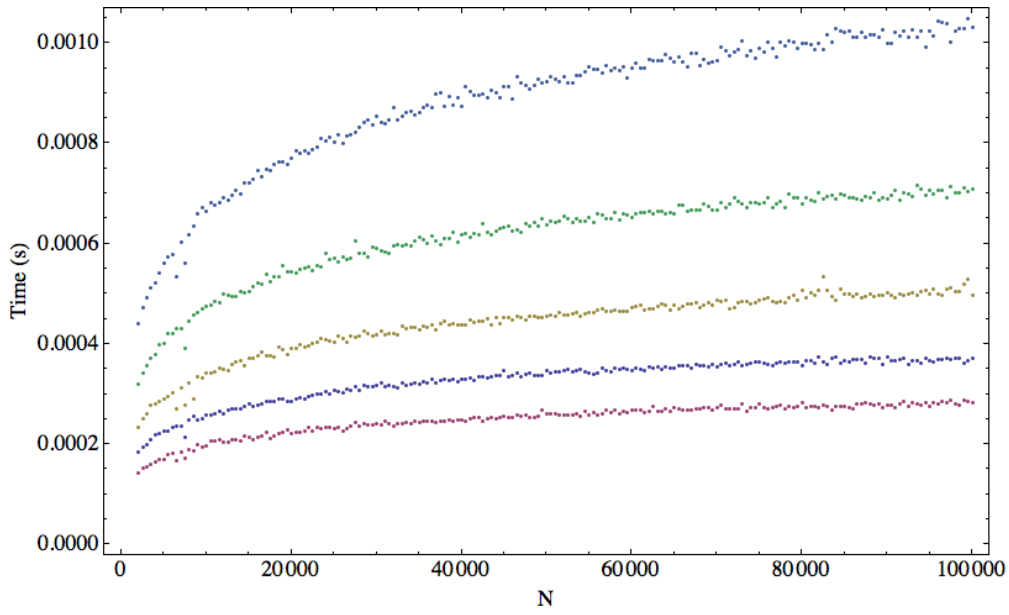

Figure 1: The time *per particle* to calculate force with the Barnes-Hut treecode for (bottom to top respectively) $\theta_{\text{crit}} = 1.0$, 0.9, 0.8, 0.7, and 0.6 as a function of the number of particles. I estimate that for 1,000,000 particles, it would take about 5 minutes to calculate the force on every particle with $\theta_{\text{crit}} = 1.0$. For $\theta_{\text{crit}} = 0.8$ it might take $8 - 9$ minutes.

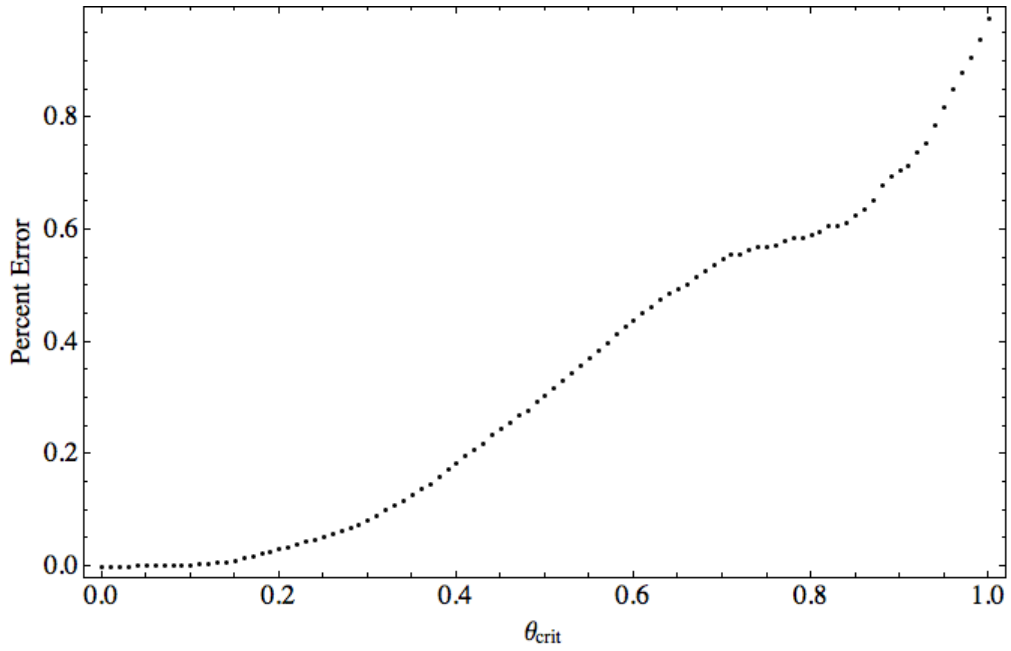

Figure 2: The average percent error (compared to direct summation) of the Barnes-Hut treecode force calculation as a function of the critical opening-angle θ_{crit}.

1.6 Visualization

This code is designed to be used with the IFrIT (Ionization Front Interactive Tool) visualization program which can be found here: `https://sites.google.com/site/ifrithome/`. It's a really well-designed, simple program which is used by the pros (I believe the millennium simulation was visualized with IFrIT?). It gives you a wide arrange of widget-controls so it only takes a few mouse clicks to make it look awesome. I recommend reading the documentation to gain some feel for how it works.

The best feature is that it will automatically animate lists of particle data with integer names (`0000.dat`, `0001.dat`, etc). This code will automatically generate a list of this sort. Once you've got the first frame looking the way you want, you can literally click "animate" and it will automatically generate a list of images which can be turned into an animation.

The only downside is that I haven't been able to install it on a Mac yet. In order to compile it from the source you need to install Kitware VTK and Trolltech Qt. I wasn't able to get VTK to install on a mac, it's one of those instances where linux doesn't work the same as OS X so all the online documentation isn't valid. The good news is that there is a binary distribution available for Windows. Unless you're a computer whiz, I recommend not wasting your time trying to install VTK, go find a PC.

2 Algorithm

The algorithm implemented in this code comes in two parts: the leapfrog time-integration scheme including variable timesteps, and the Barnes-Hut treecode force calculation. Since time integration requires that one already knows the acceleration on every mass, I'll discuss the treecode first.

2.1 Barnes-Hut Treecode

2.1.1 Tree Terminology

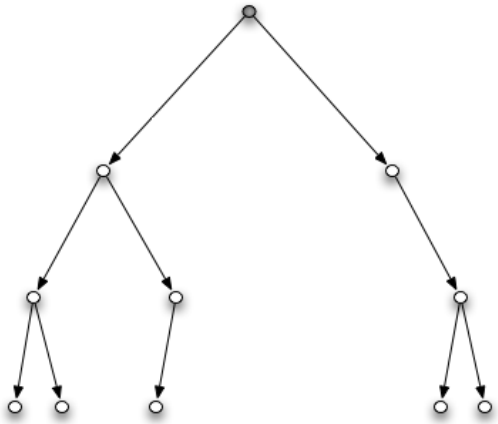

Figure 3: Schematic of a binary tree. Circles represent nodes while arrows represent branches, the links between nodes. The root node at the top is shaded grey. In this case each node may have up to two children.

A *tree* is a common data structure in computer science, at least according to the computer scientists. In a very abstract sense it consists of two parts. The first is a *node* where information is stored, either about the data itself or about the location of other nodes. The second is a *branch*, or, a reference from one node to another. A tree is a hierarchical description of data, so it makes sense to think of it having *levels*. A node on one level has *children* on the lower level, and a *parent* on the upper level. Each node may have many children but only one parent. The tree may be *traversed* only along branches. Accessing nodes via their references in other nodes is called *moving* along the tree. Naturally there must be a top and bottom to the tree. The top, the node from which all others can be accessed, is called the *root node*. The nodes on the bottom which have no children are sometimes called *leaf nodes*, though in this code I call them *body nodes* since they contain data pertaining to bodies in an N-Body simulation. A tree schematic is shown in Fig. 3.

This code makes use of a *Spatial Partitioning Tree* (SPT) in which moving from node to node along a branch is associated with a choice of coordinates. To see how an this works, consider a collection of points lying on the x-axis. The collection of points may be organized in *binary* SPT in which each node contains references to two children and the location of a *splitting point* that separates the two. Suppose that the root node's splitting point was located at the origin, then it would contain references to one node through which all positive points can be accessed, another through which all negative points can be accessed. The positive node might be split at $x = 10$ so that one of its children contained references to all points between 0 and 10, the other between 10 and ∞. This splitting would proceed until *all* of the original data could be accessed by traversing down the tree from the root node.

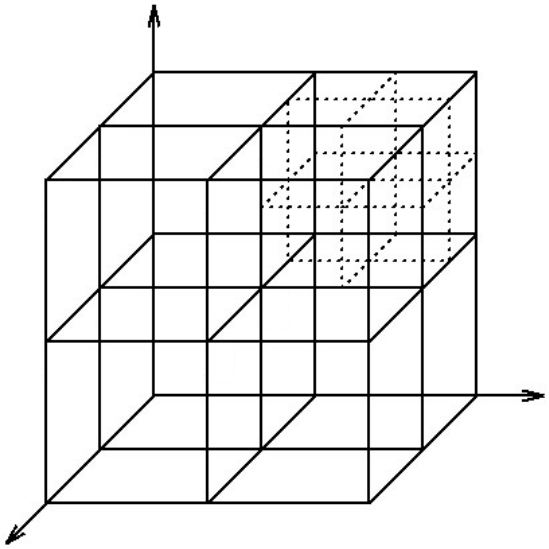

Figure 4: The Barnes-Hut octree.

2.1.2 The Barnes-Hut Octree

The basic problem with force calculation in an N-Body simulation is that in order to know the force on every particle, you must sum the gravitational field contribution from *every other* particle. That's $\mathcal{O}(N^2)$ calculations *per timestep*. This severely limits the number of particles that may be integrated even by the most powerful computers. Any significant speedup must be accomplished by approximation.

The most common way to approximate the gravitational field of a collection of objects is with it's *multipole expansion*. The problem is that the multipole approximation is only good if the point where the force is being calculated is far away from the collection of masses. When evaluating the force from a collection of masses on a particle which is far away, the multipole expansion might be very accurate. For another which is closer, it might be very inaccurate. How, then, can particles be organized such that particles can be grouped in different ways for the evaluation of the force on different masses?

Barnes and Hut were the first to realize that a Spacial Partitioning Tree is the perfect structure for this kind of flexible grouping. In three dimensions, a spatial partitioning tree can efficiently group a collection of particles into its own section of space called a *cell*. Each cell could store information about its size and location, references to other cells, how many particles it "contains," etc. Barnes and Hut proposed a variant called an *octree*, in which the nodes of the tree are a series of nested cubes such that each *parent-cube* contains eight *child-cubes*. The structure is visualized in Fig. 4.

When evaluating forces, particles can be grouped with others in cells at any level of the tree. If a grouping in a given cell is determined to be acceptable (we'll get to that later), the force may be evaluated using the multipole approximation. If not, the force may be evaluated by testing the acceptability of each of its child-cubes.

Algorithm 1 Very pseudo pseudocode describing the insertion of a single body in an existing cell. The tree is constructed by performing this algorithm, starting with the root node, for every body.

```
Function insertbody (cellnode, bodynode)

determine index j of octant containing particle
if cellnode.child[j] ≠ 0 then
    cellnode.child[j] ← reference to bodynode
else if cellnode.child[j] references an existing bodynode then
    allocate newcell
    newcell.size ← cellnode.size / 2.0
    set newcell.position to appropriate octant location
    determine index k of existing body in newcell
    newcell.child[k] ← reference to existing body
    insertbody(newcell, bodynode)
else(cellnode.child[j] references a cell)
    insertbody(cellnode.child[j], bodynode)
end if
```

2.1.3 Tree Construction

Before we get to how a tree is constructed, here's a bit more terminology: In an octree each cell node[1] contains eight "spots" which can store references to each of its child cells. Generally there is some indexing pattern for these spots so that one might say that the ith child node is associated with an octant.[2] If there is no node referenced in a particular octant, the corresponding reference will be set to 0 and the octant is said to be *free*.

The octree must be constructed such that every particle is linked all the way up the tree to the root node. The easiest way to achieve this the *top-down* method in which every particle is *inserted* into the tree starting with the root node.

The routine for inserting a body in a cell proceeds recursively. First the index of the child node that should contain the body is determined. If that octant is free then the body is linked there (as in, a reference to the body is created at the appropriate location). If the node already references a body at that location, a new cell must be created to house both the new and old bodies. That cell is placed with its geometric center offset in the appropriate direction from the original cell. Next, the old body is placed in the appropriate location in the new cell. The body-insertion algorithm is then repeated for the *new* body in the *new* cell. Finally, if the octant which should contain the new body in the original cell is occupied by another cell node, the body-insertion algorithm is repeated for the new body in that cell. This routine is shown in very pseudo pseudocode in Algorithm 1.

The entire tree is constructed by repeating Algorithm 1 for every body, until the entire system of particles may be accessed by traversing down the tree from the root. Once it is constructed, the mass of each cell is assigned recursively by summing over the mass of each of its child-nodes. The position of each cell is set to the center-of-mass of the cell, again by summing recursively over the weighted position (mass $\times$ position) of each of its child nodes. If the quadrupole moment is used,

[1] I'm going back to referring to nodes which are not leafs as *cells* and leafs as *bodies*. Just remember that now a cell is assumed to be a cube containing eight sub-cubes

[2] In this code the indexes are as such: for indices $0-7$, $0-3$ are negative (with respect to the center of the relevant cell) in the x direction while $4-7$ are positive. Of each set of 4 indices, the first two are negative in y while the second are positive. Of each pair of indices, the first is negative in z and the second is positive. For example, index 6 corresponds to the octant which is positive in x, positive in y, but negative in z.

Algorithm 2 Force calculation

```
Function treeforce (treenode)

if θ = d/r < θ_crit then
    force += a_node
else
    for i = 0 → 7 do
        if treenode.child[i] ≠ 0 then
            force += treeforce(child[i])
        end if
    end for
end if
Return force
```

this too may be set recursively, though in this code the quadrupole correction is not used.

2.1.4 The Barnes-Hut Multipole Acceptability Criterion

As mentioned above, there must be some way of determining if the multipole approximation is valid when calculating the force from a collection of particles in a cell. There are a number of ways of doing this, known as Multipole Acceptability Criterions (MACs). The simplest is the Barnes-Hut MAC, which characterizes body-cell interaction in terms of the *opening angle* $\theta = d/r$ where d is the side-length of the cell and r is the distance from the particle to the center of mass of the cell. If θ is less than some defined value θ_{crit}, the force is evaluated using the multipole approximation (for this code, just the center of mass). If not, the acceptability is tested for each child node. The recursion is guaranteed to terminate since $d = 0$ for all body nodes. In the limit where $\theta_{\text{crit}} = 0$, the method converges to regular direct summation. The timing of this method is $\mathcal{O}(N \log N)$, a *very significant* speedup over direct summation.

2.1.5 Force Calculation

As it stands, this code uses the multipole approximation only out to the dipole terms. This means that for every node (body or cell) the contribution to the gravitational field at the location of a body is simply the softened gravitational acceleration

$$\mathbf{a}_{\text{node}} = -\frac{m_{\text{node}}(\mathbf{x}_{\text{body}} - \mathbf{x}_{\text{node}})}{\left(|\mathbf{x}_{\text{body}} - \mathbf{x}_{\text{node}}|^2 + \epsilon^2\right)^{3/2}}, \tag{1}$$

where $\mathbf{x}_{\text{node}}$ is the center of mass of the node, m_{node} is its mass and $\mathbf{x}_{\text{body}}$ is the location of the body where the force is being calculated. This makes recursive tree-force calculation really easy. Algorithm 2 lays it out quite nicely.

2.2 Time-Integration

Once the acceleration on every mass is known, the differential equation governing the N-Body system is just Newton's second law:

$$\ddot{\mathbf{x}}_j = \mathbf{a}_j \tag{2}$$

where $\mathbf{a}_j$ is the acceleration. A vast number of methods exist for solving Newton's second law numerically. This code uses the leapfrog, or Verlet, method for solving Newton's second law. The leapfrog method may be derived from a discretized version of the Hamiltonian for the N-Body system. To keep the notation clean, I'll restrict consideration to a single particle but it should be pretty clear that the result for one particle generalizes to many. The Hamiltonian of a single particle is

$$H = \frac{1}{2}v^2 + \Phi(\mathbf{x}) \tag{3}$$

and the Hamilton equations of motion are

$$\dot{\mathbf{x}} = \mathbf{v} \qquad , \qquad \dot{\mathbf{v}} = -\nabla\Phi(\mathbf{x}) = \mathbf{a}(\mathbf{x}). \tag{4}$$

These equations are just a reformulation of Newton's second law as two first order ordinary differential equations as opposed to one second-order ordinary differential equations. Now, suppose we replace the true Hamiltonian (3) with an approximate Hamiltonian

$$\tilde{H} = \frac{1}{2}v^2 + \Phi(\mathbf{x})C(t) \tag{5}$$

where

$$C(t) = \Delta t \sum_{k=-\infty}^{\infty} \delta(t + 1/2 - k\Delta t) \tag{6}$$

is a periodic "comb function" of Dirac delta spikes and Δt is some small timestep. It is clear that as $\Delta t \longrightarrow 0$, $\tilde{H} \longrightarrow H$ so the trajectories determined by $\tilde{H}$ should therefore approach those determined by H. The equations of motion for this modified Hamiltonian are

$$\dot{\mathbf{x}} = \mathbf{v} \qquad , \qquad \dot{\mathbf{v}} = -\nabla\Phi\, C(t) = \mathbf{a}(\mathbf{x})\, C(t). \tag{7}$$

These equations can now be integrated from $t = 0$ to $t = \Delta t$. Suppose at $t = 0$ the mass is at $(\mathbf{x}_0, \mathbf{v}_0)$. During the interval from $t = 0$ to $t = \Delta t/2 - \epsilon$ where ϵ is very small, the potential is zero so the velocity does not change. The position "drifts" with constant velocity so that at $t = \Delta t/2 - \epsilon$,

$$\mathbf{x}_{1/2} = \mathbf{x}_0 + \frac{1}{2}\mathbf{v}_0\Delta t. \tag{8}$$

Over the interval from $t = \Delta t/2 - \epsilon$ to $t = \Delta t/2 + \epsilon$ the position is essentially constant but the velocity suffers a "kick" delivered by the delta spike at $t = \Delta t/2$. Thus the new velocity $\mathbf{v}_1$ is updated as

$$\mathbf{v}_1 = \mathbf{v}_0 + \mathbf{a}(\mathbf{x}_{1/2})\, \Delta t. \tag{9}$$

From $t = \Delta t/2 + \epsilon$ to $t = \Delta t$ the position "drifts" again with constant velocity so that

$$\mathbf{x}_1 = \mathbf{x}_{1/2} + \frac{1}{2}\mathbf{v}_1\Delta t. \tag{10}$$

Equations (8) and (10) can be combined for an interesting result,

$$\mathbf{x}_1 = \mathbf{x}_0 + \frac{1}{2}\left(\mathbf{v}_0 + \mathbf{v}_1\right)\Delta t \tag{11}$$

In other words, the position is updated at the end of each timestep using the average of original and final velocities, while velocity is updated in the middle of the timestep using the acceleration

calculated with the position $\mathbf{x}_{1/2}$. This is why the method is known as the leapfrog method: the updates of position and velocity are offset by $\Delta t/2$.

The leapfrog method will produce a list of positions and velocities at a discrete set of times t_n separated by Δt. If $\mathbf{x}_n = \mathbf{x}(t_n)$ and $\mathbf{v}_n = \mathbf{v}(t_n)$ then in total

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}_n \Delta t + \frac{1}{2}\mathbf{a}(\mathbf{x}_{n+1/2})\Delta t^2 \tag{12a}$$

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}(\mathbf{x}_{n+1/2})\Delta t. \tag{12b}$$

This method is advantageous because it is $\mathcal{O}(\Delta t^2)$ accurate but only requires one force calculation per timestep. For this reason it is the most widely used integration method for N-Body simulations.

2.2.1 Individual Timesteps

This code uses an adaptive timestep algorithm so that different particles may be integrated with different timesteps. When performing a numerical integration using the leapfrog method or any other, the timestep Δt must be small enough to capture important dynamical effects and minimize the error occurring as a result of the numerical approximation. In an integration of Newton's second law the factor which determines the dynamical timescale of a particle's motion is the magnitude of the acceleration $\mathbf{a}$. If $\mathbf{a}$ is small and relatively constant, the velocity won't vary much in time and to a good approximation, the position advances in a straight line over small timescales. If $\mathbf{a}$ is large or not very constant the velocity might change dramatically in a short timescale, the particle's path might be very far from a straight line even over a short timescale. The way to mitigate the damage is to reduce the timestep Δt until you are satisfied with the accuracy of the integration.

Adaptive timestep algorithms allow each mass to be integrated on its own individual timestep, so that the computer doesn't waste precious computation time evaluating forces that don't need to be evaluated. There must be a finite number of allowed timesteps so that large groups of particles may be advanced simultaneously. Such quantization is achieved by restricting the allowed timesteps to factors of two of some maximum timestep $\Delta t_{\max}$,

$$\Delta t_k = \frac{\Delta t_{\max}}{2^k} \qquad , \qquad k = 0,\ 1,\ 2, \ldots \tag{13}$$

All the particles with a single timestep value may be advanced simultaneously. Figure 5 shows a schematic of this quantized timestep structure. Each arc represents the integration of a single group of particles, with the horizontal axis representing time. The vertical line in the center of each arc represents the time at which the force is calculated and the velocities of that group are updated. In a way one can think of the vertical lines as barriers: in between the vertical lines, the system can be advanced backwards and forwards relatively easily since all the particles are traveling in straight lines. At the vertical lines, the system cannot be advanced without an expensive force calculation. The groups of particles must be integrated in the order in which the velocity updates occur, chronologically.

Algorithmically this is achieved by keeping track of the "prediction time" $\tau_j = t_j + \Delta t_j/2$ [3] at which the next velocity update should occur for each mass j, if t_j is the time after its last integration and Δt_j is its individual timestep. Only particles whose prediction time is equal to the *minimum* prediction time out of the entire set are advanced. When the force is calculated on particle j it is

[3] Now that I'm talking about the full set of N particles I am using subscripts to denote a mass's index as opposed to the timestep number. So, t_i is the current time of the i^{th} particle.

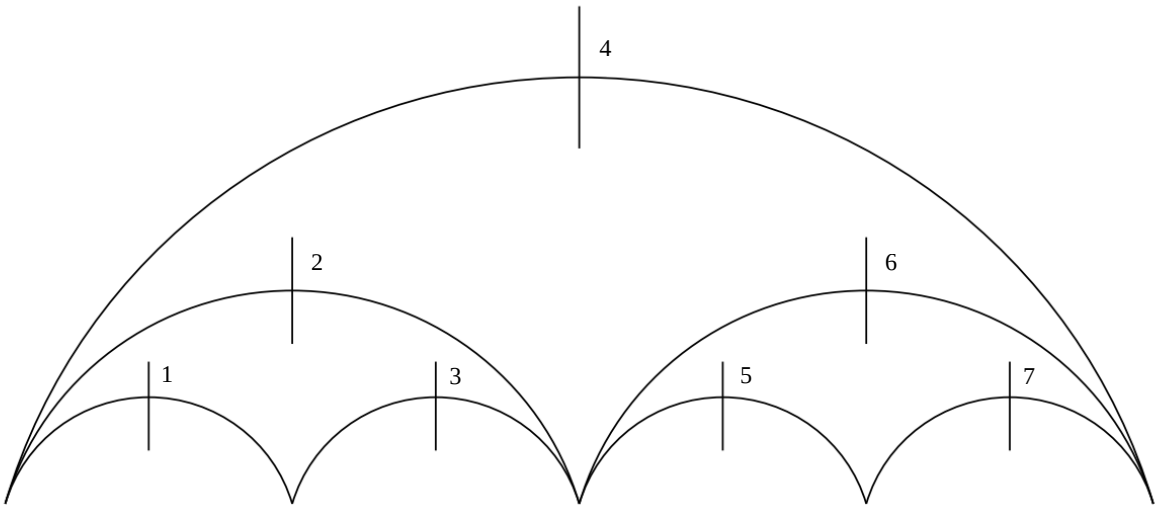

Figure 5: Schematic of the quantized individual timestep routine. Arcs represent the advancement of groups of particles while vertical lines represent force calculations. The advancements are numbered in the order that they would be performed. Modified from T. R. Quinn, N. Katz, J. Stadel, and G. Lake, Astrophys.J. (1997), astro- ph/9710043.

done using the positions $\mathbf{x}_i'$ of *all* masses at the time when the velocity update occurs,

$$\mathbf{x}_i^{1/2} = \mathbf{x}_i + (\tau_{\min} - t_i)\mathbf{v}_i. \tag{14}$$

Once a particle is integrated, its prediction time is updated and a new minimum time is selected. Thus the integration proceeds through time for all groups of masses, with each group integrated twice as often as the one above it.

There must be some way of choosing the ideal timestep for a particle given available information. Suppose we put a constraint on the magnitude of the trailing term in (12a):

$$\left|\frac{1}{2}\mathbf{a}(\mathbf{x}_{n+1/2})\Delta t^2\right| < \delta \tag{15}$$

where δ is some small length. This in turn constrains Δt,

$$\Delta t < \sqrt{\frac{2\delta}{|\mathbf{a}|}}. \tag{16}$$

The parameter δ limits the amount by which the path of a particle may deviate from a straight line to the influence of the force over one timestep. It is arbitrary but it should be related to some characteristic scale length of the system you're trying to integrate. In the code δ is assigned such that $2\delta = \alpha_{\rm tol}\epsilon$ where ϵ is the softening length and $\alpha_{\rm tol}$ is a tolerance parameter controlled by the user. Thus the ideal timestep is

$$\Delta t_{\rm ideal} = \sqrt{\frac{\alpha_{\rm tol}\epsilon}{|\mathbf{a}|}}. \tag{17}$$

Since the magnitude of the acceleration $|\mathbf{a}|$ is not known *before* the integration it must suffice to

Algorithm 3 Selecting a new timestep Δt_{new} given the previous timestep Δt_p and an ideal timestep Δt_{ideal}

if $\Delta t_{\text{ideal}} < \Delta t_p$ **then**
  $\Delta t_{\text{new}} = \Delta t_p/2$
else if $\Delta t_{\text{ideal}} > 2\Delta t_p$ AND $\text{Mod}(t/\Delta t_p, 2) = 0$ **then**
  $\Delta t_{\text{new}} = 2\Delta t_p$
else
  $\Delta t_{\text{new}} = \Delta t_p$
end if

use the magnitude of the force from the *previous* timestep. This shouldn't be so far off since the force on a single particle is a smoothly varying function of time.

Once the ideal timestep is selected, it must be sorted into the largest quantized step Δt_k such that $\Delta t_k < \Delta t_{\text{ideal}}$. In order for the method to maintain its organizational structure it must be ensured that the timestep groups remain commensurate to powers of two. In other words, that every timestep is subdivided equally by two smaller timesteps. This can be achieved by requiring that a particle's timestep may increase only *every other* integration when the current time of the particle is commensurate with the current time of the particles in a higher group. Algorithm 3 describes the method for selecting a new timestep Δt_{new} given the previous one Δt_p and the ideal timestep Δt_{ideal}.

2.2.2 Full Time-Integration Algorithm

Below is presented the full time-integration algorithm as it is implemented in the code.

Algorithm 4 Schematic of leapfrog integration including individual timesteps.

Select $\tau_{\min}$

for $j = 1 \rightarrow N$ **do**
  $\mathbf{x}_j^{1/2} = \mathbf{x}_j + (\tau_{\min} - t_j)\mathbf{v}_j$
end for

Build Tree (Algorithm 1)
for $j = 1 \rightarrow N$ **do**
  if $\tau_j = \tau_{\min}$ **then**
    Compute $\mathbf{a}_j(\mathbf{x}^{1/2})$ with treecode (Algorithm 2)
    Update $\mathbf{x}_j$ and $\mathbf{v}_j$ by leapfrog
    Calculate Δt_{ideal}
    Sort and assign Δt_j according to Algorithm 3
    Update t_j, τ_j
  end if
end for

3 Code

I've separated the N-Body code into a number of files, each containing a specific category of definitions/routines. Each item is declared in a header file with the `".hpp"` suffix, but defined in

the accompanying ".cpp" file. Below I've explained each file "bottom-up," so to speak, in that the later pieces generally depend on the definitions/routines found in the former.

1. `Vector.hpp/Vector.cpp`

   These files define a variable-length `Vector` class and its accompanying operations. When declaring a `Vector`, one must either use the copy constructor `Vector(otherVector)` or the overridden constructor `Vector(length)`. For example, a normal 3-vector could be declared with `Vector(3)`. All operators are overloaded so that they function as they should according to usual vector arithmetic, it's pretty straightforward. One less straightforward feature is that the `"<<"` operator has been overloaded so that `"std::cout << vectorname"` will print all entries of the Vector separated by a space.

2. `treenode.hpp/treenode.cpp`

   Here is defined the treenode class, the building-block of the tree structure. Each treenode contains:

   - `isbody`: a boolean value which indicates the type of node. If `true` the node is a bodynode, if `false` it is a cellnode.
   - `d`: The "size" of the `treenode`, or half of the side length of the cube associated with the node. If the node is a body, this is zero.
   - `mass`: The mass of the node. If the node is a body, this is simply the particles mass. If the node is a cell, this is the *total* mass of all particles contained in the cell.
   - `pos`: A 3-Vector containing the position of the node. In different contexts this value holds different things: If the node is a body, it holds the position of the particle. If the node is a cell, it can contain either the geometric center of the cell (during the construction of the tree), or the center-of-mass of the collection of masses contained in the cell (during force calculation).

   The treenode class is further divided into two subclasses containing information specific to body or cellnodes. Each `bodynode` contains:

   - `t`: The current time of the particle.
   - `dt`: The timestep for the next integration step.
   - `tp`: The prediction time of the particle (the time at which the velocity will be updated).
   - `vel`: A 3-Vector containing the particle's velocity.
   - `id`: An integer ID for the particle

   Each `cellnode` contains:

   - `child`: An array of 8 pointers to the child-nodes of the cell. If the child node is empty the pointer is set to 0.

3. `maketree.hpp/maketree.cpp`

   These files contain routines responsible for the creation and organization of the tree datastructure. From bottom up:

- `makecell()`: Returns a pointer to an unused, allocated `cellnode`. This is essentially a replacement for the `"new"` command for allocating memory which keeps track of previously allocated cells. A list of pointers to previously allocated but unused `cellnodes` is stored in the standard library vector `freecells` (a global variable). If `freecells` contains any pointers at all, `makecell()` returns the last one, clears all linkage data, and deletes it from the list. If `freecells` is empty it allocates new memory and returns the relevant pointer.
- `collectcells()`: Scans the tree starting at the root, collecting all `cellnodes` into the `freecells` list. It is defined recursively, so that if the relevant node is a cell, `cellnode()` is applied to each of its unempty child nodes.
- `makeboundingvol()`: Sets the box-size of the root node so that it contains all the particles. The root size `d` is set to 1.001 times the maximum difference between the position of the root node and the position of a particle in any coordinate. Takes as arguments a pointer to the array of `bodynodes`, the number of bodies `N`, and a pointer to the root node.
- `childindex()`: Finds the index of the child node containing a given `bodynode` within a `cellnode`. The indexing scheme is as follows: for indices $0-7$, $0-3$ are negative in the x direction (with respect to the center of the relevant cell) while $4-7$ are positive. Of each set of 4 indices, the first two are negative in y while the second are positive. Of each pair of indices, the first is negative in z and the second is positive. For example, index 6 corresponds to the octant which is positive in x, positive in y, but negative in z. Additionally, `makecell()` updates the 3-Vector `offset` (a global variable) so that it points from the `cellnode` in question to the child cell which would contain the `bodynode`.
- `insertbody()`: Places a `bodynode` into its appropriate location in the tree by scanning downwards from the root until a free child cell is reached. Beginning with the root, the index of the child-node containing the relevant `bodynode` is calculated using `childindex`. If that child-node is empty, the `bodynode` is linked in that location. If the child-node is a bodynode, a new cell is created to house both the new `bodynode` and the old one occupying its spot. The old cell is first placed in its relevant spot in the new cell, then `insertbody()` is called for the new body in the new cell. Finally, if the relevant child-node is a another `cellnode`, the process is repeated for that cell.
- `setcoms()`: Sets the mass and center of mass of each `cellnode` recursively. First the mass and center of mass of each child which is a cell is set, then the mass and center of mass are summed appropriately.
- `maketree()`: The main routine for the construction of the tree. First any previously existing tree is destroyed with `collectcells()`, and a new root node allocated with `makecell()`. The size of the root is set with `makeboudingvol()`, then each `bodynode` is inserted with `insertbody()`. Finally, the masses and centers of mass are set with `setcoms()`. The time required to construct the tree is reported.

4. `treeforce.hpp/treeforce.cpp`

   These files contain the routines for calculating the force on a single mass due using the tree. There are only two:

   - `gravity()`: Returns the force on an object i from another object j given its mass, a vector pointing from i to j, and the softening length ϵ.

- `treeforce()`: Returns the force on a body due to the collection of bodies contained in the tree. The tree is scanned recursively: if the acceptability criterion (that $d/r < \theta_{\rm crit}$, where r is the distance from the body to the `treenode` in question), the force is calculated using that cell's center of mass. If not, `treeforce()` is run for each of the child nodes, and the contribution summed. The recursion is guaranteed to terminate because $d = 0$ for all `bodynodes`.

5. `integrate.hpp/integrate.cpp`

   These files contain routines responsible for the time-integration of the N-Body system. From bottom up:

   - `dtideal()`: Calculates the ideal timestep given the force from the previous integration, the softening length ϵ and the tolerance parameter $\alpha_{\rm tol}$. A maximum timestep is included in case it is needed.
   - `tpmin()`: Finds the minimum prediction time in the full set of N particles.
   - `setmidpos()`: Advances all particles to the minimum prediction time, the time at which the next group of particles to be advanced will have their velocities updated.
   - `roundtoint()`: Rounds a double precision floating point number to an integer.
   - `leapfrogvar()`: Advances a bodynode in time given the force calculated from the group of particles contained in the tree. First the force is calculated with the `treeforce()` routine. Next, the positions and velocities are updated according to the leapfrog method. The current time is then set. `leapfrogvar()` also updates the timestep and prediction time of the body. The ideal timestep is calculated with `dtideal`, then divided into appropriate category as such: If the ideal timestep is less than the current timestep, the current timestep is divided by two. If the ideal timestep is greater than twice the current timestep AND the current time is commensurate with the current time of the particles in the higher timestep, the timestep is doubled. If neither is true, the timestep stays the same.
   - `advance()`: Advances the *entire* collection of N bodies. First the minimum prediction time is set with `tpmin()`, then all bodies are advanced to that time with `setmidpos()`. Next a tree is constructed with `maketree()`. Finally, if a particle's prediction time is within a rounding-error bound (10^{-10}) of the minimum prediction time `tpmin` its coordinates are advanced with `leapfrogvar()`. Finally, the total number of particles advanced and the total time required is reported.

6. `dataio.hpp/dataio.cpp`

   Contains routines managing data input and output:

   - `writeinitfile()`: Writes a file with a given name to store the initial positions and velocities of a group of particles. The file will look like this (here only the first 10 particles are shown):

```
10000 5000
-5.5374693e+00 +5.5872235e+00 -1.1986520e+00  -8.2462315e+00 +4.4659250e+00 -5.7443327e+00  +1
-4.9997971e+00 -5.4357498e+00 +3.6747376e+00  +8.4104578e+00 +1.3532263e+00 +1.3152109e+00  0
+4.7498303e+00 +5.4833461e+00 +1.4497997e+00  -9.6028153e+00 -1.4017106e+00 +6.7827974e+00  +1
-1.5246258e+00 +5.3157440e+00 +9.8047487e+00  -4.3859513e+00 +1.7093952e+00 +8.4113524e+00  0
+9.6001143e+00 -2.4923677e+00 +6.5110018e+00  +9.1210512e+00 -9.2243404e+00 -9.5932255e+00  +1
+6.6591251e+00 -7.8812352e+00 -5.0033938e+00  -8.4957532e-02 +8.0620968e-02 +7.9597382e+00  0
-6.8061185e-01 -7.7123646e-01 +8.3738158e+00  +9.5669833e-01 -2.1712159e+00 -1.2775637e+00  +1
+7.9875496e+00 -7.7845894e+00 -3.1941512e+00  +6.7455355e+00 +4.4063072e+00 -4.1000163e+00  0
-8.9745279e+00 -3.8945682e+00 +8.9637831e+00  +5.1095440e+00 +3.9917275e+00 -5.6970666e+00  +1
+9.4018837e+00 +7.8161835e+00 -6.2174227e+00  -2.5413330e+00 +6.5968812e+00 +3.7760921e+00  0
```

The first number in the first row is the total number of particles, in this case $10,000$. The next number is the total number of particles, in this case $5,000$. The next rows show the position (x, y, z) and velocities (v_x, v_y, v_z) of each particle followed by the boolean value indicating if it is dark. The file extension should be either `".dat"` or `".txt"`.

- `readinitfile()`: Reads an initialization file of the form listed above. Returns a pointer to an array of allocated `bodynodes` containing the relevant data. Also assigns `N` and `NDM` appropriately.
- `initialize()`: Sets the initial timesteps and prediction times for a collection of bodies. In order to do so the force on each particle is calculated with `treeforce()`, since the timestep selection depends on the force from the previous step (and of course in this case there is no previous step). Once the ideal timestep is found with `dtideal()`, the provisional timestep `dtprov` is set to the maximum `dtmax`. The provisional is divided by two until it is less than the ideal timestep, so that the timestep is in the appropriate category. This routine is designed to be used after `readinitfile()` to fill in the timesteps for the first integration.
- `inttostring()`: Takes an integer input and returns a string containing that number with exactly 4 digits, zeros used as fillers. This will be used to write file names below.
- `writeifritfile()`: Writes a file to be used by the IFrIT (Ionization Front Interactive Tool) visualization software to create animations and still frames. The IFrIT tool will animate lists of files ending with a four-digit number, so the filenames created here will be `0000.dat`, `0001.dat`, etc. The first line in each file lists the number of particles to be plotted. The second lists the coordinates of the bottom left corner of the box in which particles will be rendered, followed by the length of each size of the box. `writeifritfile()` makes the box a cube (this is better so the rendering box doesn't get stretched) centered at the origin with a given sidelength. This routine takes a boolean argument `plotDM` which controls whether dark matter particles will be plotted in the visualization. If they are plotted, each particle's position is listed along with the boolean value indicating if it is dark or not (So that IFrIT can assign colors according to this data). The file will look like this:

```
10000
-1.0 -1.0 -1.0 2.0 2.0 2.0
-4.3331764e+00 -1.5520198e+00 +5.1586378e+00  +1
-9.0472986e+00 +4.9460519e+00 -7.7803140e+00  0
-3.4324202e+00 -7.9069638e+00 -5.1634192e-01  +1
-6.0577633e+00 -5.7482959e+00 -1.5726392e+00  0
+7.8484615e-01 +8.1232858e+00 -9.4448671e+00  +1
+1.8128931e+00 -9.2423142e+00 -5.2657955e+00  0
-7.7816774e+00 +3.5410433e+00 -4.5261202e+00  +1
-2.4010427e+00 +9.1361904e-01 -2.1308159e+00  0
+1.1309183e+00 -3.8873436e+00 -5.2794891e+00  +1
-8.1731742e+00 -1.5568535e+00 +6.0595226e-01  0
```

If not, only the non-dark matter particles are listed, without a boolean value so that the file will look like this:

```
5000
-1.0 -1.0 -1.0 2.0 2.0 2.0
-4.2792442e+00 +2.8329029e+00 +4.7004230e+00
+3.5374453e+00 +4.9805027e+00 +9.5403010e+00
-2.1943934e+00 -4.4982437e+00 -5.0010178e+00
-9.4322487e+00 +7.3861468e-01 +3.0949485e+00
+2.4528003e+00 -3.5190618e+00 +5.2960323e+00
-8.4544142e+00 +9.3748294e+00 +4.5016430e+00
+2.7425469e+00 -7.4590266e+00 -7.9735593e+00
+1.0775542e+00 +9.3532133e-01 +4.2475201e-01
+9.8066783e-01 +6.7673799e+00 -2.1168991e+00
-2.7934921e+00 -8.3578250e+00 +1.5392186e+00
```

It's worth noting that any number of extra columns of data can be added to these files. Any number of visual attributes can be assigned based on its value.

- `writefullsave()`: Writes a full savefile containing *all* particle data.
- `readfullsave()`: Reads a full savefile and allocates data accordingly.

References

[1] J. E. Barnes, *Treecode Guide*, `http://www.ifa.hawaii.edu/~barnes/treecode/treeguide.html`

[2] D. M. Moore, thesis, Reed College, 1997

[3] N. Muldavin, thesis, Reed College, 2013

[4] E. Athanassoula, J. C. L. E Fady, and A. Bosma, Mon. Not. R. Astron. Soc. **000**, 1 (1999).