Building and Using a Toy N-body Code

Ian Fare

November 8, 2017

1 Introduction

N-body simulations are incredibly useful tools that allow astrophysicists to study diverse types of systems on many different scales. One such type of system is star clusters, which are composed of thousands or millions of stars, bound gravitationally. This report documents the process of building, from classical physics and a little bit of math, a very basic "toy" n-body code, and using it to describe some very basic properties of the dynamics of globular clusters.

2 Building an n-body code

2.1 Choosing an integration method

The n-body problem is the problem of determining the motion of n point masses m_i at positions $\mathbf{r}_i$, with $i = 1, 2, ..., n$. With the motion of each point mass governed by Newton's law of universal gravitaion, the n-body problem is represented by a system of n second-order ordinary differential equations:

$$\mathbf{r"}_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{Gm_j}{\|\mathbf{r}_i - \mathbf{r}_j\|^3}(\mathbf{r}_j - \mathbf{r}_i) \tag{1}$$

n-body simulations must solve this system of differential equations in one way or another. There are a number of numerical methods of solving these differential equations, differing in their accuracy and time complexity. Numerical methods step through time, advancing by some interval h of time and calculating the position and velocity of each point mass. Each step introduces a *local truncation error* (LTE) which scales

with h, and the local truncation errors of each step accumulate to some *global truncation error* (GTE) at time t. An integration method is of order p if the local truncation error is on the order of $\mathcal{O}(h^{p+1})$. So, given the initial state of an n-body system, to solve for the system at time t with a given error tolerance, a higher-order method will not require as small an interval h as a lower-order method, and will reach t with fewer time steps, and thus with less computation.

2.2 Programming the Runge-Kutta method

The fourth-order Runge-Kutta method (RK4) is a popular integrator for instructive purposes, although higher-order methods are generally used for research. To solve a second-order ODE of the form $\mathbf{r''} = \mathbf{f}(\mathbf{r})$, like Newton's law of universal gravitation, the method is as follows [1]:

$$\mathbf{r}_{n+1} = \mathbf{r}_n + h\left(\mathbf{r'}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2)\right) + \mathcal{O}(h^4) \tag{2}$$

$$\mathbf{r'}_{n+1} = \mathbf{r'}_n + \frac{1}{6}\mathbf{k}_1 + \frac{2}{3}\mathbf{k}_2 + \frac{1}{6}\mathbf{k}_3 \tag{3}$$

$$\mathbf{k}_1 = h\mathbf{f}(\mathbf{r}_n) \tag{4}$$

$$\mathbf{k}_2 = h\mathbf{f}\left(\mathbf{r}_n + \frac{h}{2}\mathbf{r'}_n + \frac{h}{8}\mathbf{k}_1\right) \tag{5}$$

$$\mathbf{k}_3 = h\mathbf{f}\left(\mathbf{r}_n + h\mathbf{r'}_n + \frac{h}{2}\mathbf{k}_2\right) \tag{6}$$

In the case of an n-body code, $\mathbf{r}$ is a particle's position in space, and $\mathbf{f}$ is the force of gravity, divided by the particle's mass, from all the other particles.

To get started programming RK4 in C++, I include the libraries `iostream`, `iomanip`, `fstream`, `cmath`, and `vector`, and use the namespace `std`. To generate particles with masses, positions, and velocities, I wrote a function called `read_file()` which takes as arguments pointers to three vectors: `masses`, which will contain the masses of the particles, `positions`, which will contain their positions, and `velocities`, which will contain their velocities. The latter two are vectors of vectors (or, 2D matrices), as each position and velocity is itself a vector of length 3, as we are working in three dimensions. Each of these vectors has a size of n, where n is the number of particles to be simulated. The function reads a file called `./fort.10`, which contains this information in the format required for NBODY6, and fills the vectors with the values found therein. I have also written a function `vectoradd()`, which takes two vectors as arguments and returns their

sum as you would expect, and `scalarmult()`, which takes a vector as its first argument and a scalar as its second argument and returns the vector multiplied by the scalar.

My implementation of RK4 lives in a function that takes as arguments the step size, as well as the pointers for the vectors `masses`, `positions`, and `velocities`, and of two other vectors, `new_positions` and `new_velocities`. It iterates through each particle (the ith element in `masses` corresponds to the ith element in `positions` and in `velocities`). For each particle, it calculates new position and velocity vectors following the equations above: Equation (2) corresponds to lines 31 through 36, Equation (3) corresponds to lines 38 through 43, Equation (4) corresponds to lines 14 and 15, Equation (5) corresponds to lines lines 17 through 22, and Equation (6) corresponds to lines 24 through 29. Finally, it swaps the old and new position and velocity vectors, to get ready for the next step.

```
1  void rk4_step(vector<double> &masses, vector<vector<double> > &positions,
2                vector<vector<double> > &velocities, double h,
3                vector<vector<double> > &new_positions,
4                vector<vector<double> > &new_velocities)
5  {
6    int N = masses.size();
7
8    vector<double> k1(3), k2(3), k3(3);
9    vector<double> k2_position(3),k3_position(3);
10
11   // For each particle
12   for(int i=0; i<N; i++){
13
14     // Calculate k1
15     k1 = scalarmult(acc(masses, positions, i, positions[i], h, print), h);
16
17     // Calculate k2
18     k2_position = vectoradd(vectoradd(positions[i],
19                                       scalarmult(velocities[i], 0.5*h)),
20                             scalarmult(k1, 0.125*h));
21
22     k2 = scalarmult(acc(masses, positions, i, k2_position, h, print), h);
23
24     // Calculate k3
25     k3_position = vectoradd(vectoradd(positions[i],
26                                       scalarmult(velocities[i], h)),
27                             scalarmult(k2, 0.5*h));
28
29     k3 = scalarmult(acc(masses, positions, i, k3_position, h, print), h);
30
31     // Calculate the particle's new position
32     new_positions[i] =
33       vectoradd(positions[i],
34         scalarmult(vectoradd(velocities[i],
35                              scalarmult(vectoradd(k1,scalarmult(k2,2.0)),
36                                         1.0/6.0)), h));
37
38     // Calculate the particle's new velocity
39     new_velocities[i] =
40       vectoradd(velocities[i],
41                 vectoradd(scalarmult(k1,1.0/6.0),
42                           vectoradd(scalarmult(k2,2.0/3.0),
43                                     scalarmult(k3,1.0/6.0))));
44   }
```

```
45
46    // Swap new and old positions and velocities
47    std::swap(positions,new_positions);
48    std::swap(velocities,new_velocities);
49  }
```

That's all that it takes to program RK4. There are even simpler implementations, but this way the lines of code correspond directly to lines in the method's mathematical notation. Now, each time $\mathbf{k}_1$, $\mathbf{k}_2$, and $\mathbf{k}_3$ are calculated, the function calls another function: `acc()`, which calculates the acceleration vector on the particle currently being stepped forwards, simply evaluating the right-hand side of Equation (1). For gravity, I have the function take as inputs the index of the mass being stepped forward (its position in the mass, position, and velocity vectors), and pointers to the mass, position, and velocity vectors, as well as to another vector (which I probably should have named less confusingly), `position`. The `position` (singular) vector contains the position at which to calculate acceleration; for $\mathbf{k}_2$ and $\mathbf{k}_3$, this is not necessarily the particle's actual (previous) position in the `positions` (plural) vector. Here is the code:

```
1  vector<double> acc(vector<double> &masses, vector<vector<double> > &positions,
2                     int index, vector<double> &position)
3  {
4    int N = masses.size();
5    vector<double> current_acc(3), r(3), rnorm(3);
6    double rsq,rmag;
7
8    // For each particle
9    for (int i=0; i<N; i++){
10     // If it's not the one being stepped forward
11     if (i != index){
12       rsq = 0;
13       // Calculate difference in position, and |r|^2
14       for (int j=0; j<3; j++){
15         r[j] = positions[i][j] - position[j];
16         rsq += r[j]*r[j];
17       }
18
19       // Calculate |r|, use it to normalize r
20       rmag = sqrt(rsq + 0.0001);
21       rnorm = scalarmult(r,1.0/rmag);
22
23       // Add to acceleration vector
24       for (int j=0; j<3; j++){
25         current_acc[j] += masses[i]*rnorm[j]/(rmag*rmag);
26       }
27     }
28   }
29
30   return current_acc;
31 }
```

Now, it is worth mentioning that I actually had to make a choice in writing this code. There are two mathematically equivalent ways of representing Newton's law of universal gravitation. I gave one of them in

Equation (1):

$$\mathbf{r"}_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{Gm_j}{\|\mathbf{r}_i - \mathbf{r}_j\|^3}(\mathbf{r}_j - \mathbf{r}_i)$$

Equivalently,

$$\mathbf{r"}_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{Gm_j}{\|\mathbf{r}_i - \mathbf{r}_j\|^2}\frac{(\mathbf{r}_j - \mathbf{r}_i)}{\|\mathbf{r}_i - \mathbf{r}_j\|} \tag{7}$$

While these mathematically mean the same thing, they might produce different results in a simulation. That's because, in Equation (1), we run the risk of getting a very small number for $\|\mathbf{r}_i - \mathbf{r}_j\|^3$ in the denominator, and running into floating point errors. Alice and Bob, the characters in Hut and Makino's Maya Open Lab Development Series [2], seem happy to use Equation (1), but they seem to know what they're doing better than I do, so I would defer to their judgement if they disagreed.

These two functions contain most of my n-body code. The main function just runs `rk4_step()` repeatedly, with a step size `h` and for a length of simulation time given in an input file.

2.3 Other integration methods

N-body codes used for research tend to employ higher-order integrators than the fourth-order Runge-Kutta method. NBODY6 uses Hermite integration, which runs well in parallelized codes [3]. Hermite integration was introduced by Junichiro Makino for use on the special-purpose HARP computer [4]. It is a predictor-corrector scheme for solving a differential equation of the form $\frac{dx}{dt} = f(x,t)$. In NBODY6, at each time step, force and its time-derivative are explicitly calculated, and expanded in a Taylor series to the third order. Given the force and its time-derivative at the current and previous time steps, the second and third time-derivatives are calculated from those expansions. Then, the Hermite interpolation polynomial $f_h(t)$ is calculated. Using force and its first, second, and third time-derivatives at r points $t_{i-r+1}, ..., t_i$ to calculate $f(t_{i+1})$, it satisfies:

$$f_h^{(k)}(t_l) = f^{(k)}(x_l, t_l) \quad , \quad (k = 0, 1, 2, 3\,;\, l = i - r + 1, ..., i) \tag{8}$$

This Hermite interpolation polynomial is used to calculate the predictor and corrector. The predictor $x_{p,i+1}$ is given by:

$$x_{p,i+1} = x_i + \int_{t_i}^{t_{i+1}} f_h(t)dt \tag{9}$$

To produce a corrector, Equation (8) is applied again, with a slight change:

$$f_h^{(k)}(t_l) = f^{(k)}(x_l, t_l) \quad , \quad (k = 0, 1, 2, 3\,;\, l = i - r + 2, ..., i + 1) \tag{10}$$

with $x_{i+1} = x_p$. The Hermite interpolation polynomial interpolates the last r steps, including the current one, using the predictor as x_{i+1}. Then, the corrector $x_{c,i+1}$ is calculated the same way the predictor was:

$$x_{c,i+1} = x_i + \int_{t_i}^{t_{i+1}} f_h(t)dt \tag{11}$$

3 Studying star clusters using *n*-body codes

Now that we have a working (or working to "toy n-body" standards) code, there are many features of a star cluster's dynamic life that we can study. Many of the features of a globular cluster's life should be visible in simulations carried out using my code. However, since my code uses only a fourth-order method, it has to use a smaller step size than higher-order methods, and since it just runs on a single CPU thread (rather than a GPU, or even just using multithreading), it is quite slow. So for the purpose of this report I have been limited to simulations of 1,000 stars or fewer; I will be trying to identify properties of star cluster dynamics in the results of such simulations.

3.1 Equipartition and mass segregation

Many of the large-scale dynamical properties of star clusters are statistical results of individual, small-scale interactions. For example, stars in two-body encounters are statistically likely to equalize their kinetic energies; as a result, we observe equipartition of kinetic energies in star clusters:

$$m_1\langle v_1^2\rangle = m_2\langle v_2^2\rangle \tag{12}$$

for any ranges m_1 and m_2 of mass.

The time it takes for stars in a cluster to (more or less) satisfy Equation 12 is called the relaxation time, $t_n = \frac{N}{8 \ln N} \times t_{cross}$, with t_{cross}, the crossing time, being the time it takes a star to cross the cluster.

In simulations, it is possible to verify equipartition of energy by plotting v^2 against mass and seeing that they are independent. However, it is arguably more fun to see how equipartition changes the physical structure of a star cluster, and measure that instead.

Equipartition of energy leads high-mass stars to be concentrated near the centre of clusters, and low-mass stars to be spread around its edges; this is called mass segregation. When stars of different masses have roughly equal kinetic energies, higher-mass stars will be travelling slower on average than lower-mass stars. As a result, higher-mass stars sink in the system's potential well, to the centre of the cluster, while lower-mass stars expand outward. So, if we can measure the distribution in mass and space of stars in a cluster, at some point in the simulation, and observe that higher-mass stars tend to be near the centre, then at that point the cluster has reached equipartition. Additionally, the time it takes for the mass distribution in a cluster to reach a fairly constant state is the relaxation time.

To explore equipartition and mass segregation, I had my n-body code run a couple different simulations. The first is 1000 stars, using for initial conditions a King model with $w_0 = 7$, with a mass function from Kroupa 2001 [5] from 0.1 to 50Msun, virialized, generated using McLuster [6]. I used a step size of $h = 0.001$. From hereon in, this will be referred to as **Sim A**. As shown in **Figure 1**, although at the start of the simulation the Lagrangian shells all contain stars with very similar average masses, early on in the simulation the 10% and 25% Lagrangian shells increase in average mass, as high-mass stars migrate inwards and low-mass stars migrate outwards.

I ran a second simulation (**Sim B**) with 1000 stars, using for initial conditions no density gradient, with a mass function from Kroupa 2001 [5] from 0.1 to 50Msun, and "cold" – that is, with all stars having no velocity, also generated using McLuster. I used a step size of $h = 0.001$. As shown in **Figure 2**, the average masses in the 10% and 25% Lagrangian shells are very noisy for the first half of the simulation; this is because during this time, the cluster is rapidly collapsing and all the stars are passing through the centre at around the same time. However, throughout the simulation we find that the average mass in the 50% Lagrangian shell is clearly higher than the average mass in the 75% shell, and after halfway through the simulation, the simulation, the average mass in the 25% shell is higher than that of the 50% shell too. The 10% shell is still too confused to mean much, however.

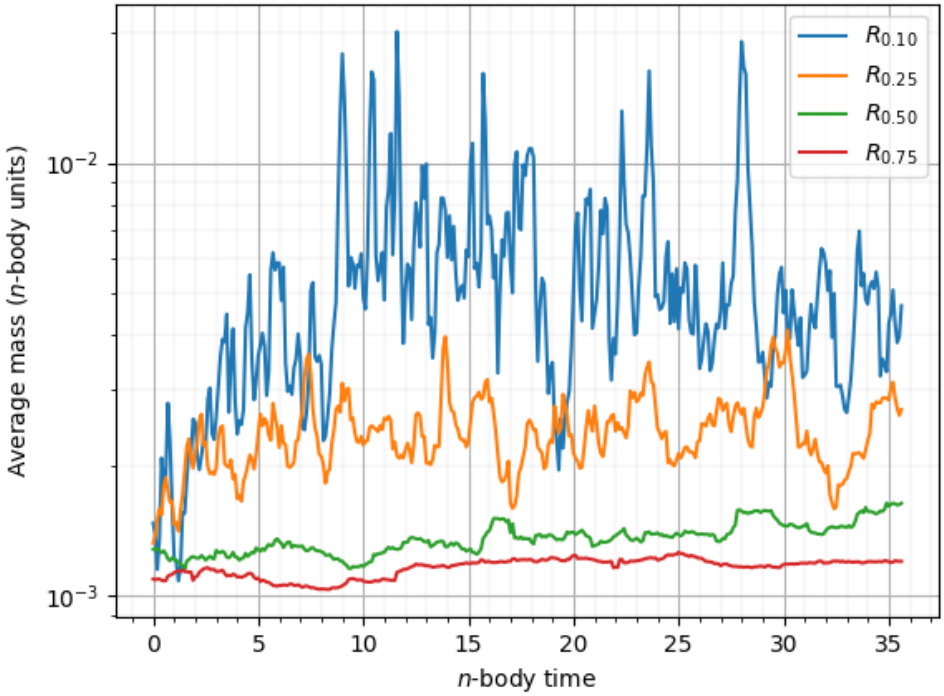

Figure 1: Average mass of stars contained in different Lagrangian shells through simulation, **Sim A**.

3.2 Escape and evaporation

If at any point a star has a kinetic energy exceeding the gravitational potential of its stellar system at its location, it will almost certainly escape the system [7]. This condition is shown in Equation 13.

$$\frac{v_{esc}^2}{2} + \phi > 0 \tag{13}$$

where v_{esc} is the escape velocity ϕ is the smoothed potential of the system at the star's location.

Stars can gain become escapers by gaining velocity through interactions with other stars, or by, rather than gaining kinetic energy, having a constant energy while stars in the system evolve and lose mass, decreasing the depth of the its potential well [7].

As previously discussed, in two-body encounters between stars, their energies tend to equalize. Therefore, in encounters between higher- and lower-mass stars, there is a tendency for lower-mass stars to come out with higher velocities than the higher-mass stars with which they interact. It is low-mass stars, then, that tend to reach escape velocity and escape the cluster. Over the lifetime of a cluster, more and more stars will

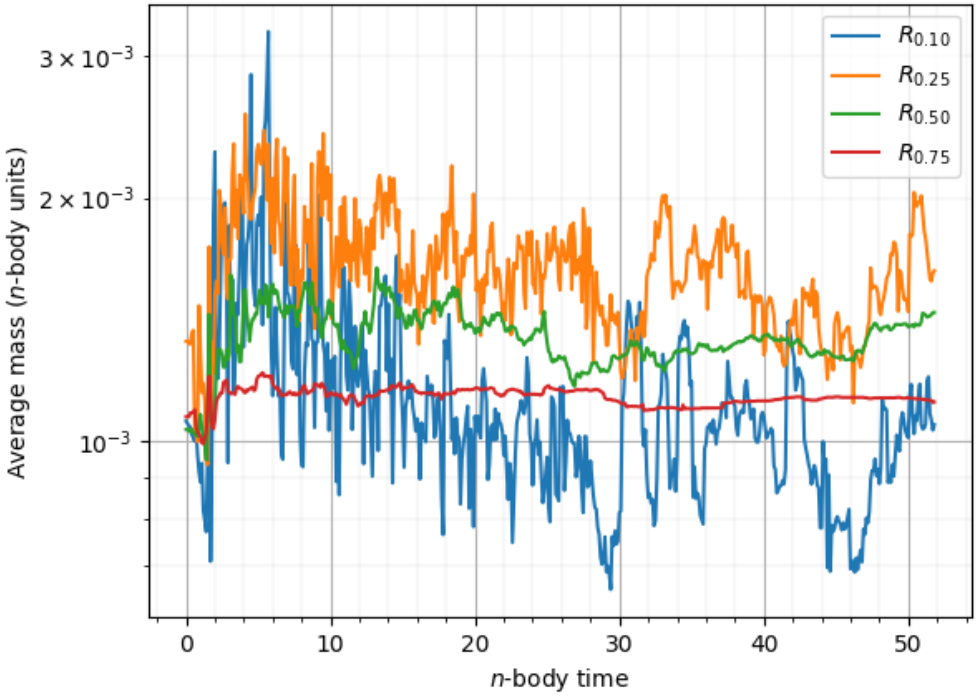

Figure 2: Average mass of stars contained in different Lagrangian shells through simulation, **Sim B**.

escape in this way, and eventually evaporate. The cluster in **Sim A** is dissolving from the very beginning. As shown in **Figure 3**, the Lagrangian radii are expanding, with the outer radii expanding faster than the inner radii. This is because the low-mass stars are expanding outwards, and the 75% Lagrangian radius with them, and many are being ejected from the cluster. Meanwhile, the 1%, 5%, and 10% Lagrangian radii, whose masses are mostly composed of large stars with low velocities, do not expand much, if at all.

In "real" n-body simulations, often a cluster is limited by the tidal that it lives in; depending on the tidal field, the cluster will have some tidal radius, beyond which the external gravitational field of the galaxy dominates the dynamics of any stars. Stars that pass the tidal radius are considered lost to the cluster, and so the cluster sheds mass over time and evaporates.

4 Conclusion

The process of constructing and using an n-body code has been full of choices. For example, I chose to use a fourth-order Runge-Kutta integrator, even though there are plenty of better ones, like the Hermite

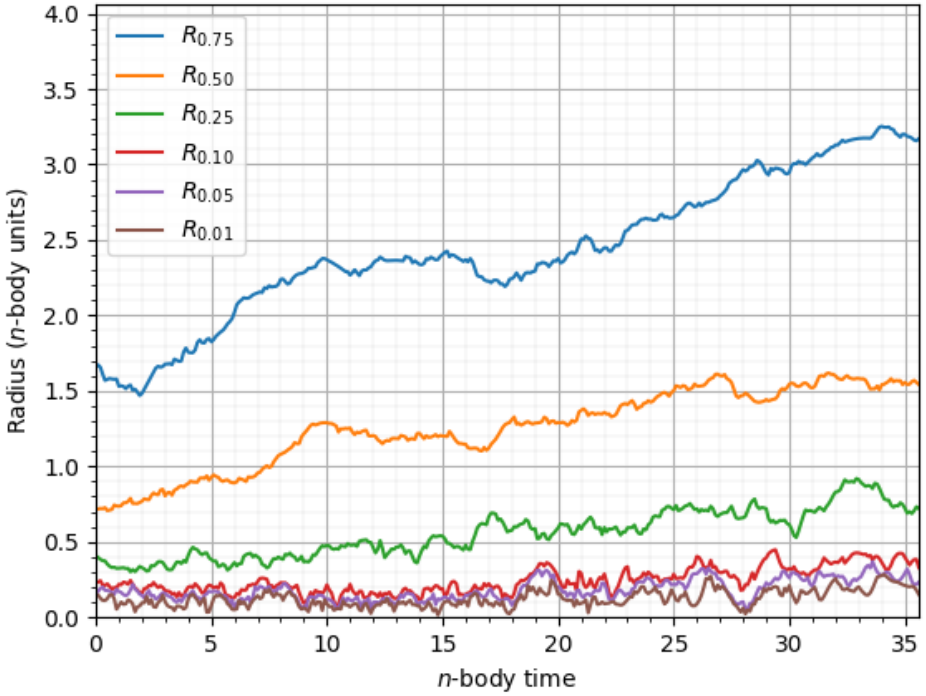

Figure 3: Lagrangian radii of **Sim A**.

integrator. I chose not to include a tidal field, and not to implement stellar evolution. I chose not to deal with mergers, or use variable time steps, or step stars forward through time individually. I chose to analyze a paltry few aspects of my simulations' results. The "toy" n-body code is orders of magnitude simpler than any the professional codes used for research, and even so, there is an immense amount of things I could learn from it, by running different analyses on the data it has produced, or by tweaking the initial conditions, or by simply letting it run for longer and watching how my clusters dissolve. The simplicity of my code highlights the careful thought and rigour and comprehensiveness of other codes, and reveals the staggering volume of *things* to be understood about star clusters, and stellar dynamics in general. I can learn (and already have learned) a lot about stellar dynamics from my own code, and it's very basic and slow, so it's exciting to imagine what can be learned from studying realistic systems using professional codes. The work described in this report will serve as a foundation for the development of such tools in the coming months, which will hopefully help illuminate an interesting little corner of the vast terrain of stellar dynamics.

References

[1] F. Olver, "Handbook of mathematical functions ed m abramowitz and ia stegun," 1965.

[2] P. Hut and J. Makino, "The art of computational science: The maya open lab development series, vol. 1," 2007.

[3] S. J. Aarseth, "From nbody1 to nbody6: The growth of an industry," *Publications of the Astronomical Society of the Pacific*, vol. 111, no. 765, p. 1333, 1999.

[4] J. Makino, "Optimal order and time-step criterion for aarseth-type n-body integrators," *The Astrophysical Journal*, vol. 369, pp. 200–212, 1991.

[5] P. Kroupa, "On the variation of the initial mass function," *Monthly Notices of the Royal Astronomical Society*, vol. 322, no. 2, pp. 231–246, 2001.

[6] A. H. Küpper, T. Maschberger, P. Kroupa, and H. Baumgardt, "Mass segregation and fractal substructure in young massive clusters–i. the mcluster code and method calibration," *Monthly Notices of the Royal Astronomical Society*, vol. 417, no. 3, pp. 2300–2317, 2011.

[7] D. Heggie and P. Hut, "The gravitational million-body problem: a multidisciplinary approach to star cluster dynamics," 2003.

5 Appendix

nbody_rk4.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
#include <vector>

using namespace std;

// Vector addition
vector<double> vectoradd(vector<double> vec1, vector<double> vec2)
{
  int vecsize = vec1.size();
  vector<double> sum(vecsize);
  for(int i=0; i<vecsize; i++){
    sum[i] = vec1[i] + vec2[i];
  }
  return sum;
}

// Scalar multiplication
vector<double> scalarmult(vector<double> vec, float scalar)
{
  int vecsize = vec.size();
  vector<double> prod(vecsize);
  for(int i=0; i<vec.size(); i++){
    prod[i] = vec[i]*scalar;
  }
  return prod;
}

// Read NBODYx fort.10 file for initial state
void read_file(vector<double> &masses, vector<vector<double> > &positions,
               vector<vector<double> > &velocities)
{
  ifstream inFile;

  inFile.open("fort.10");
  if (!inFile){
    cout << "Unable to open fort.10" << endl;
  }

  double number;
  int counter = 0;
  vector<double> current_position(3);
  vector<double> current_velocity(3);

  // For all the floats in fort.10
  while (inFile >> number){
    /* Each particle gets 7 values
         0 mass
         1 position x
         2 position y
         3 position z
         4 velocity x
         5 velocity y
         6 velocity z
       Wrap through sets of 7 to assign values for individual stars */

    if(counter == 7){counter = 0;}
    if(counter == 0){masses.push_back(number);}
    else if(counter == 1){current_position[0] = number;}
    else if(counter == 2){current_position[1] = number;}
    else if(counter == 3){current_position[2] = number;}
```

```
    else if(counter == 4){current_velocity[0] = number;}
    else if(counter == 5){current_velocity[1] = number;}
    else if(counter == 6){
      current_velocity[2] = number;
      positions.push_back(current_position);
      velocities.push_back(current_velocity);
    }
    counter ++;
  }
}

vector<double> acc(vector<double> &masses, vector<vector<double> > &positions,
                   int index, vector<double> &position)
{
  int N = masses.size();
  vector<double> current_acc(3), r(3), rnorm(3);
  double rsq,rmag;

  // For each particle
  for (int i=0; i<N; i++){
    // If it's not the one being stepped forward
    if (i != index){
      rsq = 0;
      // Calculate difference in position, and |r|^2
      for (int j=0; j<3; j++){
        r[j] = positions[i][j] - position[j];
        rsq += r[j]*r[j];
      }

      // Calculate |r|, use it to normalize r
      rmag = sqrt(rsq + 0.0001);
      rnorm = scalarmult(r,1.0/rmag);

      // Add to acceleration vector
      for (int j=0; j<3; j++){
        current_acc[j] += masses[i]*rnorm[j]/(rmag*rmag);
      }
    }
  }

  return current_acc;
}

void rk4_step(vector<double>& masses, vector<vector<double> > &positions,
              vector<vector<double> > &velocities, double h,
              vector<vector<double> > &new_positions,
              vector<vector<double> > &new_velocities,bool print)
{
  int N = masses.size();

  vector<double> k1(3), k2(3), k3(3);
  vector<double> k2_position(3),k3_position(3);

  // For each particle
  for(int i=0; i<N; i++){

    // Calculate k1
    k1 = scalarmult(acc(masses, positions, i, positions[i]), h);

    // Calculate k2
    k2_position = vectoradd(vectoradd(positions[i],
                                      scalarmult(velocities[i], 0.5*h)),
                            scalarmult(k1, 0.125*h));

    k2 = scalarmult(acc(masses, positions, i, k2_position), h);

    // Calculate k3
    k3_position = vectoradd(vectoradd(positions[i],
```

```
                                      scalarmult(velocities[i], h)),
                            scalarmult(k2, 0.5*h));

    k3 = scalarmult(acc(masses, positions, i, k3_position), h);

    // Calculate the particle's new position
    new_positions[i] =
      vectoradd(positions[i],
        scalarmult(vectoradd(velocities[i],
                             scalarmult(vectoradd(k1,scalarmult(k2,2.0)),
                                        1.0/6.0)), h));

    // Calculate the particle's new velocity
    new_velocities[i] =
      vectoradd(velocities[i],
                vectoradd(scalarmult(k1,1.0/6.0),
                          vectoradd(scalarmult(k2,2.0/3.0),
                                    scalarmult(k3,1.0/6.0))));
  }

  // Swap new and old positions and velocities
  std::swap(positions,new_positions);
  std::swap(velocities,new_velocities);
}



int main()
{
  double h;
  double time;

  // Get input file
  ifstream inFile;
  inFile.open("input");
  if (!inFile){
    cout << "Unable to open input" << endl;
  }

  // Get timestep and time to run from input file
  double number;
  int counter = 0;
  while (inFile >> number){
    if (counter == 0){
      h = number;
      counter++;
    }
    else{
      time = number;
    }
  }

  unsigned long int timesteps = (time/h);

  // Positions and velocities from NBODYx fort.10
  vector<double> masses;
  vector<vector<double> > positions, velocities;

  read_file(masses,positions,velocities);
  int N = masses.size();

  vector<vector<double> > new_positions(masses.size());
  vector<vector<double> > new_velocities(masses.size());

  double Et,velocity_sq;

  // For each timestep
  for (int step=0; step<timesteps; step++){
```

```
200     // Step forward RK4
201     rk4_step(masses,positions,velocities,h,new_positions, new_velocities, false);
202     // Every 100 steps print out state of system
203     if(step%100==0){
204       for (int star=0; star<N; star++){
205         cout << masses[star] << " " << positions[star][0] << " "
206              << positions[star][1] << " " << positions[star][2] << " "
207              << velocities[star][0] << " " << velocities[star][1] << " "
208              << velocities[star][2] << endl;
209       }
210     }
211     if(step%100==0){cout << "!!!!!!!!!" <<step<< endl;}
212   }
213   return 0;
214 }
```