Measuring Rate of Globular Cluster Evaporation using Particle Mesh Models

Nick Stern
Brown University Physics Department
(Dated: May 11, 2017)

In the following report I use a Particle-Particle Particle-Mesh (P3M) algorithm to model a cluster of stars under the influence of a gravitational field, and measure the rate at which the cluster evaporates as a function of the initial mass and velocity distribution. Results show that the evaporation time decreases with an increase in the average velocity of the globular cluster, though there was not a significant correlation between the evaporation time and the mass distribution of the globular cluster.

I. INTRODUCTION

Globular clusters were formed very early on in the history of our galaxy, typically found to be around 13 billion years in age, with a variation of less than 5 billion years.[1] Sizes of globular clusters range from about 10^4 to 10^7 stars, and the tidal radii of clusters (distance at which the stars are less influenced by the cluster than they are by another system) tend to be on the order of 10 kpc. It is unlikely to find any that are much smaller than that because of a process known as "globular cluster evaporation." Globular cluster evaporation is the process of ejecting stars due to standard gravitational mechanics and stellar interactions within the globular mass. It is the aim of this research paper to model, and quantify this phenomenon. Specifically, in this research paper we seek to answer the following question: how does the rate at which stars get ejected from a globular cluster depend on the initial mass and velocity distributions within the cluster? We seek to understand the relationship between stellar mass, speed, and the evaporation rate.

Celestial mechanics does a fantastic job of producing a fully-formed, analytical solution to a two body orbit. However, our solar system alone is occupied by millions of asteroids and planetary objects of comparable mass, all exerting gravitational forces upon one another in a way that is too intricate for analytical solutions alone to decipher. Therefore, numerical methods are essential to modeling and simulating celestial interactions. Restricting our view to only the most massive objects within our solar system, it is feasible to treat each object individually and calculate the gravitational force one object exerts on each of the N-1 remaining objects. However, this approach quickly gets out of hand as we relax our mass constraint, or increase our point of view to the size of a galaxy. When millions or even billions of objects are interacting with each other gravitationally, it is no longer feasible to treat each body individually, due to limitations on computational power and time. To that extent, a number of new algorithms have been developed since the 1970's to decrease the computational complexity of N-body simulations from $O(N^2)$ to $O(Nlog(N))$.[2] One of these algorithms, and the one that we shall make use of in this research paper to study the dependence of evaporation rate on stellar mass and velocity, is known as the Particle-Mesh (PM) model, described in further detail in the method section.

II. THEORY

It is important to first note that the dominant force within Globular clusters is gravity, and therefore a simulation that examines stellar behavior in a gravitational field alone is appropriate for our purposes. Additionally, unless we are interested in super-massive objects, it is reliable to use a Newtonian model for gravity. The equation that determines the strength of the gravitational field is Poisson's equation, depicted below:

$$\nabla^2\Phi(\vec{x}) = 4\pi G\rho(\vec{x}) \tag{1}$$

There are three important sources of evaporation in isolated clusters of stars. The first source comes from the random distribution of velocities that stars can take on, very similar to the Maxwell-Boltzmann distribution of an ideal gas.

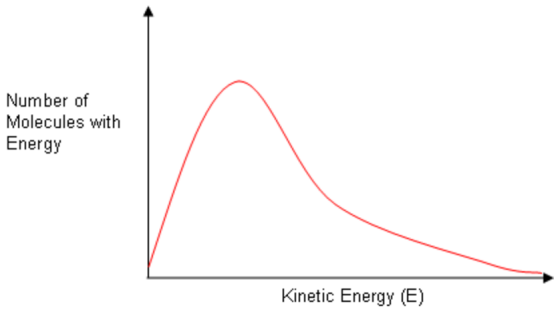

FIG. 1: The Maxwell-Boltzmann Distribution

As is shown in Figure 1, there is a tail to the Maxwell-Boltzmann distribution at higher energies. Therefore, stars within the cluster can obtain random velocities greater than the escape velocity, and shoot out of the gravitationally bound system. The second common source of cluster evaporation is motivated by binary star systems. In an open cluster, stellar interaction is generally not frequent enough to transfer energy effectively in a way that can lead to offhand, excessively high velocities. Binary star systems, however, *are* able to transfer

energy effectively. The introduction of a third star passing by a binary system can lead to the passing of energy to the third star, and propel it out of the cluster in a way that a unitary star cannot. The third source of evaporation comes from binary systems in which one of the stars reaches the end of its life cycle and explodes into a supernova.[3] This explosion is dramatic enough to give its binary counterpart the velocity necessary to escape the cluster. The preliminary simulation used in this research paper looks exclusively at the first of these three sources of globular cluster evaporation.

III. METHOD

As mentioned in the introduction, for N-body particle simulations with higher and higher N, it becomes increasingly difficult to use an algorithm that calculates forces from all the particles on each other. This type of algorithm is referred to as a Particle-Particle (PP) model, and is very effective at smaller scales because of its greater force resolution. In this research paper we intend to explore large scale interactions across many particles, and therefore we make use of a Particle-Mesh model. The process of the particle mesh model is outlined in the following stages. The first step to implementing the PM algorithm is to implement a "mesh" or a grid that spans the space of the simulation. The purpose of this mesh is to capture and record the masses of nearby particles to create a mass distribution. For our purposes, we generate a 3D cube and divide it into cells with nodes in the center of each, as depicted in Figure 2 below:

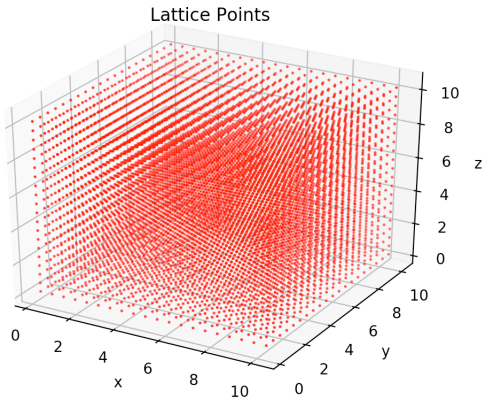

FIG. 2: 10x10x10 Example of Mesh Grid

Next, we initialize a series of particles with randomized masses, initial positions, and initial velocities, as shown in Figure 3. The sizes of the particles in the simulation reflect their relative mass. We choose to initialize the particles in a central region of the grid, to avoid getting sucked into the periodic boundary, which shall be discussed later.

It is worth noting at this point that there are several ways in which to interpolate the particle masses onto the

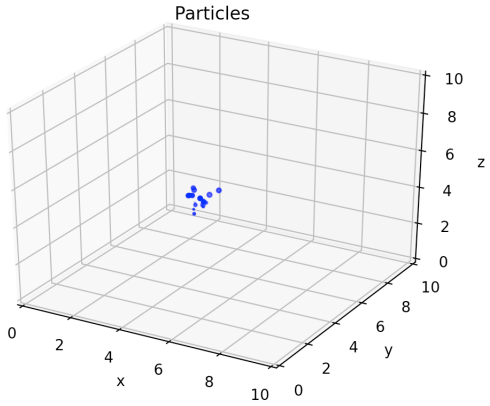

FIG. 3: 20 Particles Initialized in 10x10x10 Grid

grid. The simplest method, and the one used in this research paper is the Nearest Grid-point (NGP) method, whereby all of the particle's mass is assigned to the nearest grid point. The NGP method does come at a loss of resolution, as it is a zero-order interpolation. Another method, one that perhaps could be implemented in a later version of the simulation, is known as the Cloud in Cell (CIC) assignment. CIC assignment splits the particles mass among several adjacent grid points in three dimensions, and is equivalent to a first order interpolation. The CIC algorithm is widely used in popular practice of N-body simulations.

Once the particle masses have been interpolated into the grid, we can use Equation 1 to back-solve for the potential distribution. However, we must do so with a method that is less complex than $O(N^2)$, and therefore an improvement over the PP algorithm mentioned earlier. To overcome this, the Particle-Mesh algorithm Fourier transforms the Poisson equation, and solves for gravitational potential in Fourier space, in accordance with Ewald summation. This results in a decrease in complexity from $O(N^2)$ to $O(Nlog(N))$ because of the low-complexity Fast Fourier Transform (FFT) algorithm.[2] The Poisson equation (continuous solution) in Fourier space becomes:

$$\widetilde{\Phi}(\vec{k}) = \frac{-4\pi G}{|\vec{k}|^2}\widetilde{\rho}(\vec{k}) \tag{2}$$

where the tilde's indicate that Φ and ρ are functions of the wavenumber, k. With periodic boundary conditions and a discrete Laplacian, the wavenumber is defined to range from $k = -n\pi/L$ to $k = n\pi/L$, for n_1, n_2, n_3 and L_1, L_2, L_3, corresponding to each of the three dimensions. n is referred to as the wavenumber index, and is defined to be $n = L/\Delta x$, where Δx in this case is the cell width.

Once the gravitational potential distribution is solved for in Fourier space, we can then inverse Fourier transform it back to the spatial dimension. An example of the

potential distribution for a single point mass is plotted in Figure 4.

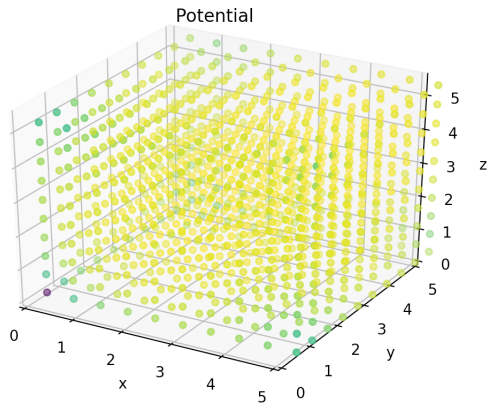

FIG. 4: Potential distribution associated with a single particle in the lower left hand corner of the a 5x5x5 grid.

With the potential distribution in real space obtained, we can utilize the relationship between force and potential, $F_g(\vec{x}) = -\nabla\Phi(\vec{x})$, to solve for the force using a centralized finite difference numerical method. Once we have a gravitational force vector associated with each node in the mesh, interpolate that back to the particle *using the same interpolation scheme as before*. It is imperative to use the same interpolation scheme that was used to assign mass to the mesh, so that the particles do not create forces on themselves, thus disrupting Newton's 3rd law.

Finally, we can obtain new positions and velocities for the particles by using a leapfrog integration method with the following ordinary differential equations:

$$\frac{d\vec{v}}{dt} = \frac{F_g(\vec{x})}{m},$$
$$\frac{d\vec{x}}{dt} = \vec{v}$$

where m in this case is the mass of the particle. Repeating this process an integration results in particle motion!

Periodic boundary conditions were imposed in the situation because although the system is considered in isolation, it exists within a universe with a nonzero average density. Simulations that do not use periodic boundary conditions experience gravitational collapse toward the center of the model, which results in improper density perturbations, and erroneous results.[4]

In order to quantify the rate of ejecta from the globular cluster, we may define the boundary of the cube to be the point of no return, and keep a counter of how many stars hit the edges of the cubic grid. We then record at what time every star has exited the cube, which we denote to be the complete evaporation of the system.

IV. RESULTS

In order to determine whether the potential falls off as $1/r$ as expected, in the same grid setup as Figure 4, a slice along the x axis ($y = 0$, $z = 0$) was plotted against the analytical result to yield Figure 5

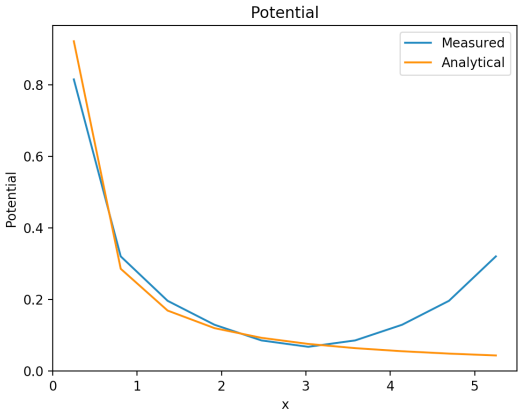

FIG. 5: Measured potential along x slice vs. analytical expectation. The graph is normalized.

The reason the measured potential curves up at larger x is due to the periodic boundary conditions imposed on the grid. Increasing the size of the grid leads to a convergence to the proper analytical potential, as demonstrated in Figure 6.

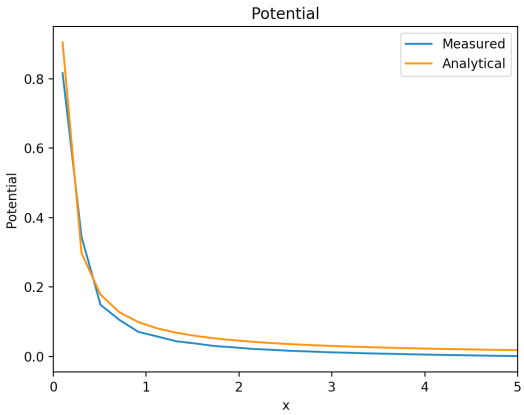

FIG. 6: Measured potential along x slice vs. analytical expectation with larger grid. The graph is normalized.

In order to determine the simulation was working correctly, we used the base case of analyzing the potential distribution and gravitational field created by black hole centered in our grid space. Figure 7 displays a color map of the black hole grid potential. The potential is mostly negative around the black hole, and is also spherically symmetric, two properties that indicate our calculated potential field is accurate.

An additional check to determine that the potential field was correct was simply verifying that the inverse

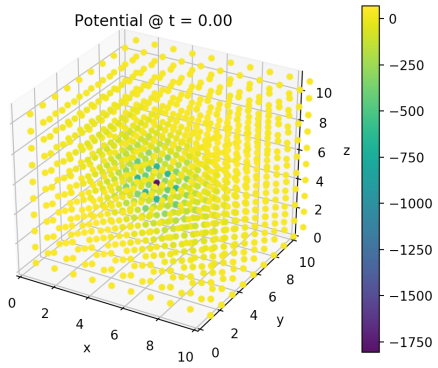

FIG. 7: Color map of black hole grid potential

Fourier transform solution to Poisson's equation was not complex valued. Nontrivial complex values within the potential distribution cannot be discarded, and visibly change the way the particles move. Animating the simulation in real time demonstrated that with complex potentials, the stars within the globular cluster did not interact, and diverged much faster than they did in accurate potential. One further check that was done on the potential distribution was to verify that the initial potential distribution was propagated accurately throughout the algorithm. By plotting potentials at various time steps within our simulation, we were able to determine that the potential distribution adjusted to particle motion correctly, and total particle mass was conserved.

Next, in order to ascertain whether the gravitational vector field was calculated as expected, we created a quiver plot of the vectors at each mesh point, and plotted this over the potential distribution of our base case, the black hole. To visualize results clearly, we implemented a mask on the potential distribution and gravitational field, blocking potentials below a magnitude of 100, and thus isolating the potential values and field vectors within the immediate vicinity of the black hole. These results are plotted in Figure 8.

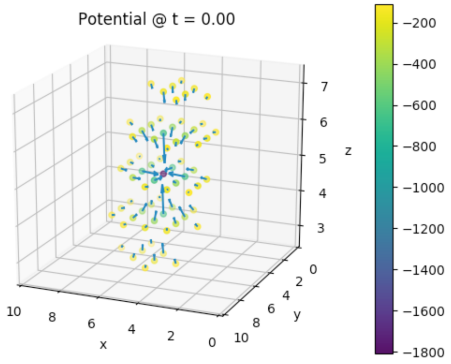

FIG. 8: Overlaid color map and quiver plot of black hole with mask threshold of 100.

As is shown in the quiver plot, all of the arrows point inward towards the black hole, which indicates that the gravitational field is calculated correctly, and follows expectation. This was also verified for distributions of multiple particles across various timescales, though the visualizations lacked the clarity of the black hole base case. The ultimate check to ensure that the results from the simulation are accurate and trustworthy, was to animate the globular cluster in real time, and watch to see that the stars are responsive to changes in their initial positions, velocities, and masses, but most of all, to check that the stars are responsive to each other. This was tested successfully with various initial conditions

After performing extensive checks to ensure that the simulation output made physical sense, we moved on to calculating and plotting the evaporation time as a function of various velocity and mass distributions. We chose to investigate this relationship by creating uniform, random distributions of velocity and mass within specified parameters, and varying the parameters with respect to one another. For the purposes of our analyses, we chose units such that Newton's gravitational constant, G, is set equal to 1. Since focus of the analysis is on the general relationship between evaporation time, mass, and velocity, units won't be specifically defined going forward. To first test the evaporation time vs. velocity, we initialized a particle distribution with the following properties:

Initial Parameters for Varying Velocity	
Grid Size	10x10x10
# Particles	5
Mass Range	.1 <m <5
Position Range	3 <p <7
Initial Velocity Range	.1 <v <1

We examined the evaporation time vs. velocity in two ways. First, we kept the initial minimum velocity the same, and increased the maximum velocity within the randomized distribution by increments of $v = .2$, until it reached a new maximum of $v = 5.0$. With each incremental velocity, we ran our simulation five times for a period of $t = 100$ with a time-step of $h = .05$, recording each evaporation time with the run. We took the mean and standard deviation of the five evaporation time data points per iteration, and plot the results in Figure 9.

Next, we varied the initial minimum velocity along with the initial maximum velocity by increments of $v = .2$ until the maximum velocity reached $v = 5.0$ once again. The rest of the procedure remained the same as before. These results are plotted in Figure 10.

For comparing the evaporation time with the mass distribution, we used a very similar procedure with the following, slightly different initialization conditions:

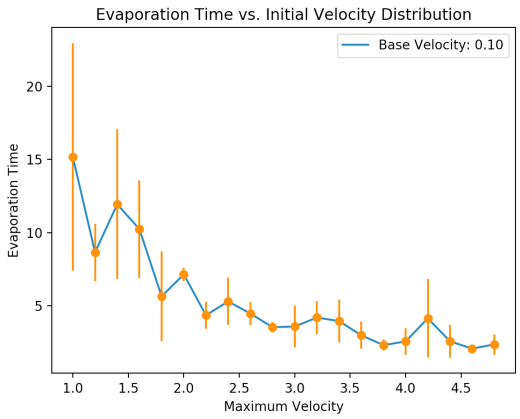

FIG. 9: Evaporation time vs. velocity distribution keeping a baseline minimum velocity

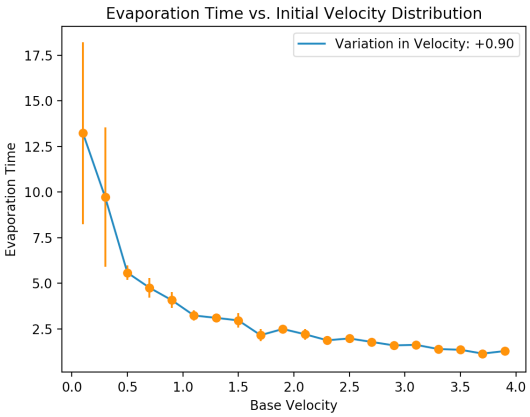

FIG. 10: Evaporation time vs. velocity distribution varying minimum velocity with maximum velocity

Initial Parameters for Varying Mass	
Grid Size	10x10x10
# Particles	5
Mass Range	.1 <m <1
Position Range	3 <p <7
Initial Velocity Range	.1 <v <1

Similar to the procedure for the velocity, we first kept the initial minimum mass the same varied the maximum mass by increments of $m = 1$ until the mass reached new maximum of $m = 20$. For each iteration we recorded the mean and standard deviation of five evaporation time data points, and plot the results in Figure 11.

We then varied the initial minimum mass along with the initial maximum mass by increments of $m = 1$ until the maximum mass reached $m = 20.0$ once again. These results are plotted in Figure 12.

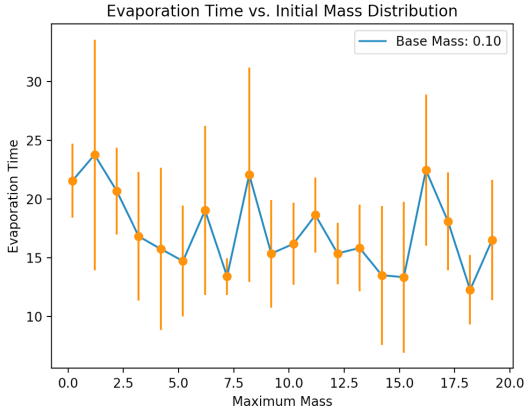

FIG. 11: Evaporation time vs. mass distribution keeping a baseline minimum mass

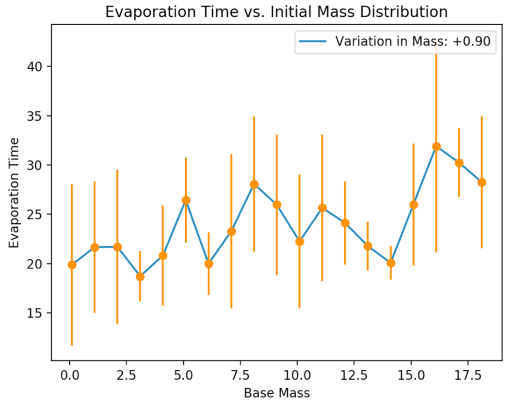

FIG. 12: Evaporation time vs. mass distribution varying minimum mass with maximum mass

V. DISCUSSION

Based on the nature of the Maxwell-Boltzmann distribution, which we anticipate to be the only physical force of motivation for globular cluster evaporation in our simulation, we expect the evaporation time to decrease with an increase in the average velocity of the velocity distribution. The Maxwell-Boltzmann distribution is often used to characterize ideal gases at various temperatures, and one property of the distribution is that the tail 'grows' with the temperature. Since temperature can be expressed as the average kinetic energy of the particles in the gaseous system, this property indicates that as the average kinetic energy of the system increases, the distribution skews to allow for more particles to exist with a kinetic energy much larger than the mean. With regard to our simulation, this forms the basis for our expectation that an increase in the average kinetic energy of our system results in a decrease in the average evaporation time. Looking at Figures 9 and 10, this relationship is indeed observed. In our analysis, we explored two different types of variation in the velocity distribution. In the first situation, where the velocity is kept at a minimum baseline

level, the average kinetic energy increases at a slower rate than the second situation, where the minimum velocity increases as well. Therefore, we expect the decrease in evaporation time to be more gradual in the first case. This phenomenon was not clearly observed. Referring to the figures in question, the evaporation times trend downwards at approximately the same rate. However, it does appear that in Figure 10 the velocity reaches a lower value at an earlier point in time, when the error bars converge.

The reason the error bars are large at the start of each velocity plot is because given the size of the grid and the number of particles, a smaller particle velocity allows for more interaction between the orbiting bodies, thus complicating the rate of evaporation, and creating a larger range of evaporation time values. For this reason as well, the error bars are larger throughout Figure 9 because of the smaller baseline velocity. It is difficult to gauge the escape velocity of the system based on the two setups for analysis. If we consider the escape velocity to be the average velocity beyond which the evaporation time does not change (i.e. the particles are traveling too fast for interactions with one other to matter), because we only record the time when all the particles have left the system, we are limited in each setup in different ways. In the first setup, where the minimum velocity is retained, we are limited by the minimum velocity, in that the slowest moving particle will be the last one to leave the system, and thus set the evaporation time. In the second system, we are often limited by the fact that the minimum velocity is so high, the cluster may travel together off the space of the grid before the particles fully diverge in the sense of a regular interaction. A future implementation of this analysis could systematically place particles with higher velocities in a cloud of low velocity particles, to more accurately test the limits of the escape velocity.

In examining Figures 11 and 12, we see no definitive correlations between the evaporation time and the average mass. One would expect that with more mass, there is more inwardly directed gravitational force, and thus the particles are less likely to diverge from one another. Interestingly, it appears that this is not necessarily the case. One explanation could be that greater average masses produce larger gravitational force on one another, and the two cancel out to maintain an equilibrium whereby masses, on average, are perturbed an equal amount by one another. Once again, we are limited in examining the full scope of the effect of mass on stellar ejecta in the way that we defined the evaporation time, especially within the first setup. In the first setup, where a minimum mass baseline is maintained, it is quite likely that the larger masses exert large forces on smaller masses, propelling them out of the globular cluster more often than in the second setup, where the minimum mass increases with the maximum mass. The large error bars in each plot suggest that the distributions of stars have more time to interact with one another. If permitted more time, it would have been fruitful to examine varieties of mass distributions with a larger baseline velocity, to see if they "stick together better" at larger average mass.

VI. CONCLUSION

In conclusion, this simulation was successful in that it expressed the expected relationship between evaporation time and various particle velocity distributions. The simulation did not find a correlation between the evaporation time and the mass distribution of the stars, which may be due to limited statistics, although possible alternative explanations were provided. Most of all, the main triumph of this simulation was the successful implementation of the Particle-Mesh algorithm to produce real-valued potential distributions, and fields of gravity that point where they are expected to. One major pitfall of the simulation was that the Particle-Mesh algorithm is best used in large scale simulations where long range forces are the primary dictator of particle motion. Though we were able to animate the simulation in real time for 20 particles in a grid size of 50x50x50, this was not feasible in collecting evaporation time data, as the simulation had to be run over and over and over again to accumulate statistics. Apart from the further analysis suggested in the discussion section, this simulation could have been improved in two other ways. First, short range forces could have been taken into account using the PP method on small scale interactions, which would turn the Particle-Mesh algorithm into a superior Particle-Particle Particle-Mesh (P3M) algorithm. Furthermore, a CIC mass density assignment operation could have improved the resolution of the simulation over the NG approach. Finally, we could have incorporated proper units into our parameters to get a better physical understanding of how our bodies orbit one another.

REFERENCES

[1] Benacquista, M. J. "Relativistic Binaries in Globular Clusters." *Living Reviews in Relativity* 16.4 (2013): 47-59.

[2] Thijssen, J. M. *Computational Physics*. Cambridge: Cambridge UP, 2000. 220-27.

[3] Perets, Hagai B., and Ladislav Subr. "The Properties of Dynamically Ejected Runaway and Hyper-Runaway Stars." *The Astrophysical Journal* 751.133 (2012): 1-6.

[4] Bagla, J. S. "Cosmological N Body Simulation: Techniques, Scope and Status." Current Science 88.7 (2005): 1088-100.

APPENDIX: CODE

```
# Gravitational N Body Simulation using P3M Method
# Created by Nick Stern on 4/24/2017

import numpy as np
from numpy import matlib
import matplotlib
# matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import random as r
import pandas as pd
import matplotlib.animation as am
import time
# Initialize 3D Grid
class grid:
    def __init__(self,l,n):
    # x, y, z are the N x N x N dimensions you want your grid to be
    # n is the number of cells you want in a given direction
        self.l = l
        self.cell_width = np.float(l)/n
        self.gs = l / (2.*n)
        self.x, self.y, self.z = np.meshgrid(np.linspace(0,l,n)+self.gs,np.linspace(0,l,n)+
    self.gs, np.linspace(0,l,n)+self.gs)
        self.n = n
        self.density = np.zeros((n,n,n))

    def plot(self):
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(self.x,self.y,self.z,color='r',s=1)
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z')
        plt.title('Lattice Points')
    def reset_density(self,n):
        self.density = np.zeros((n,n,n))

class particle:
    def __init__(self,n,mrange,vrange,prange):
    # n particles w/ random masses, velocities, positions in mrange, vrange, and prange
    # mrange, vrange, and prange are all tuples
        self.n = n
        self.m = np.zeros(n)
        self.vx, self.vy, self.vz = np.zeros(n), np.zeros(n), np.zeros(n)
        self.px, self.py, self.pz = np.zeros(n), np.zeros(n), np.zeros(n)
        for i in range(n):
            self.m[i] = r.uniform(min(mrange),max(mrange))
            self.vx[i] = r.uniform(min(vrange),max(vrange)) * r.choice([-1, 1])
            self.vy[i] = r.uniform(min(vrange),max(vrange)) * r.choice([-1, 1])
            self.vz[i] = r.uniform(min(vrange),max(vrange)) * r.choice([-1, 1])
            self.px[i] = r.uniform(min(prange),max(prange))
            self.py[i] = r.uniform(min(prange),max(prange))
            self.pz[i] = r.uniform(min(prange),max(prange))
        # make first particle a black hole in center of grid
        # self.m[0] = 1000
        # self.px[0] = ((max(prange)-min(prange))/2)+min(prange)
        # self.py[0] = ((max(prange)-min(prange))/2)+min(prange)
        # self.pz[0] = ((max(prange)-min(prange))/2)+min(prange)
        # self.vx[0] = 0
        # self.vy[0] = 0
        # self.vz[0] = 0
```

```
61
63      def reset_pos(self, pos):
65          self.px = pos[0:self.n, 0]
            self.py = pos[0:self.n, 1]
            self.pz = pos[0:self.n, 2]
67
69      def randv(self,vrange):
        # randomize initial velocities without changing other properties of the particle
        distribution
71          for i in range(self.n):
                self.vx[i] = r.uniform(min(vrange), max(vrange)) * r.choice([-1, 1])
73              self.vy[i] = r.uniform(min(vrange), max(vrange)) * r.choice([-1, 1])
                self.vz[i] = r.uniform(min(vrange), max(vrange)) * r.choice([-1, 1])
75
        def randm(self,mrange):
77      # randomize initial masses without changing other properties of the particle
        distribution
            for i in range(self.n):
79              self.m[i] = r.uniform(min(mrange), max(mrange))
81
        def plot(self, grid):
83
            fig = plt.figure()
85          ax = fig.add_subplot(111, projection='3d')
            ax.scatter(self.px, self.py, self.pz, color='b',s = 10*self.m)
87          ax.set_xlabel('x')
            ax.set_ylabel('y')
89          ax.set_zlabel('z')
            ax.set_xlim3d(0,grid.l)
91          ax.set_ylim3d(0,grid.l)
            ax.set_zlim3d(0,grid.l)
93          plt.title('Particles')
            plt.show()
95
        def nearest(self,grid): # unfinished
97      # returns nearest grid point
            nrst, idx = np.zeros(self.n),np.zeros((self.n,3))
99          for i in range(self.n):
                dist = np.sqrt((grid.x-self.px[i])**2 + (grid.y-self.py[i])**2 + (grid.z-self.
    pz[i])**2)
101             nrst[i] = np.ndarray.min(dist)
                temp_idx = np.where(dist == nrst[i])
103             idx[i,:] = np.array([x.item() for x in temp_idx])

105         return idx

107 def assign_density(grid,particle):
        grid.reset_density(grid.n)
109     idx = particle.nearest(grid)
        for i in range(particle.n):
111         grid.density[idx[i, 0], idx[i, 1], idx[i, 2]] = particle.m[i]
        return grid.density
113
    def leapfrog(lfdiffeq, r0, v0, t, h, G):        # vectorized leapfrog
115     """ vector leapfrog method using numpy arrays.
            It solves general (r,v) ODEs as:
117         dr[i]/dt = f[i](v), and dv[i]/dt = g[i](r).
            User supplied lfdiffeq(id, r, v, t) returns
119         f[i](r) if id=0, or g[i](v) if id=1.
            It must return a numpy array if i>1 """
121     hh = h/2.0
```

```
    r1 = r0 + hh*lfdiffeq(0, r0, v0, t, G)     # 1st: r at h/2 using v0
    v1 = v0 +  h*lfdiffeq(1, r1, v0, t+hh, G)  # 2nd: v1 using a(r) at h/2
    r1 = r1 + hh*lfdiffeq(0, r0, v1, t+h, G)   # 3rd: r1 at h using v1
    return r1, v1

def gravity(id, r, v, t, G):
    if (id == 0): return v  # velocity, dr/dt
    return G # dv/dt

def escaped(grid,particle):
    esc = 0
    for i in range(particle.n):
        pos = np.array([particle.px[i], particle.py[i], particle.pz[i]])
        test1 = np.sum((pos > grid.l).astype(int)) # is one of the coordinates greater than grid dim
        test2 = np.sum((pos < 0).astype(int)) #is one of the coordinates less than 0
        test = test1 + test2 # this will be greater than zero if one of these conditions is violated
        if test > 0:
            esc += 1
    return esc


def step_pvals(grid,particle,Fgx,Fgy,Fgz, t, h):

    # particle-mesh method
    idx = particle.nearest(grid)
    encounter = 0
    for i in range(particle.n):
        # For each particle, test whether there's a particle within a nearby cell
        # cell_tolerance = 4 #test whether the index of another particle is within this number
        # # subtract row i from each row of idx and sum rows, then test against tolerance
        # test = np.sum(np.absolute(idx - np.matlib.repmat(idx[i,:],np.shape(idx)[0],1)), axis=1) < cell_tolerance
        # if np.sum(test.astype(int)) > 1:
        #     encounter += 1
        # take each component of gravity and divide by particle mass
        rx, ry, rz = particle.px[i],particle.py[i],particle.pz[i]
        vx, vy, vz = particle.vx[i],particle.vy[i],particle.vz[i]
        Gx = Fgx[idx[i,0],idx[i,1],idx[i,2]]/(particle.m[i])
        Gy = Fgy[idx[i,0],idx[i,1],idx[i,2]]/(particle.m[i])
        Gz = Fgz[idx[i,0],idx[i,1],idx[i,2]]/(particle.m[i])
        particle.px[i], particle.vx[i] = leapfrog(gravity, rx, vx, t, h, Gx)
        particle.py[i], particle.vy[i] = leapfrog(gravity, ry, vy, t, h, Gy)
        particle.pz[i], particle.vz[i] = leapfrog(gravity, rz, vz, t, h, Gz)

    # print('Particles within cell tolerance: %i' %encounter)
    return particle.px,particle.py,particle.pz,particle.vx,particle.vy,particle.vz

def diagnostic(g,p,c,t,h,tmax):
    for i in range(1):
        # assign masses to grid density matrix
        g.density = assign_density(g, p)
        # Fourier transform the density to find the potential
        potentialf = c * np.fft.fftn(g.density)

        # inverse fourier transform this back to find spatial potential
        potential = np.fft.ifftn(potentialf)
        print(potential)
        # plot the initial spatial potential as a surface plot
        # create a threshold mask for the potential:
        thresh = 2 # minimum potential
        mask = np.abs(np.real(potential)) <= thresh
```

```
        gx_mask = np.ma.masked_where(mask, g.x)
        gy_mask = np.ma.masked_where(mask, g.y)
        gz_mask = np.ma.masked_where(mask, g.z)
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        # with mask
        p1 = ax.scatter(gx_mask, gy_mask, gz_mask, c=np.real(potential), s=3)
        # without mask
        # p1 = ax.scatter(g.x, g.y, g.z, c=np.real(potential))
        fig.colorbar(p1)
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z')
        # ax.set_zlabel('Potential')
        plt.title('Potential @ t = %.2f' %t)
        ax.set_xlim3d(0, g.l)
        ax.set_ylim3d(0, g.l)
        ax.set_zlim3d(0, g.l)

        # calculate the gravity vector from the potential field

        Fg = np.gradient(np.real(potential)) # taking the real values b/c I still get imaginary when inverse Fourier Transforming
        Fgy, Fgx, Fgz = Fg[0]*-1, Fg[1]*-1, Fg[2]*-1

        # add quiver plot on top of potential:
        ax = fig.gca(projection='3d')
        scalar = .2 #scalar multiple to reign in lengths of arrows
        ax.quiver(gx_mask, gy_mask, gz_mask, Fgx*scalar, Fgy*scalar, Fgz*scalar)


        p.px, p.py, p.pz, p.vx, p.vy, p.vz = step_pvals(g, p, Fgx, Fgy, Fgz, t, h)
        positions = np.zeros((p.n, 3))
        positions[:, 0] = p.px
        positions[:, 1] = p.py
        positions[:, 2] = p.pz

        t += h
    return potential



def initialize():
# initialize grid and particles
    g = grid(11,11)
#     g = grid(11, 11)
    # g.plot()

    p = particle(5, [.1,5],[.01,1],[3,7])
    # p = particle(5, [.1,5],[.01,1],[3,7])
    # p.plot(g)

# Initialize Fourier Values
    dx = g.cell_width
    n = g.l / g.cell_width  # does this have to be an integer?
    # Initialize k


    m = np.linspace(-1, 1, n)
    m = np.roll(m,np.ceil(len(m)/2.).astype(int))
    # Initialize k, the wave vector
    k = np.zeros((g.n, g.n, g.n))

    for i in range(g.n):
```

```
        for j in range(g.n):
            for x in range(g.n):
                # k[i,j,x] = (np.pi*m[i]/g.l)**2 + (np.pi*m[j]/g.l)**2 + (np.pi*m[x]/g.l)**2 #this is k^2
                # k[i,j,x] = (2*np.pi*i/g.l)**2 + (2*np.pi*j/g.l)**2 + (2*np.pi*x/g.l)**2
                k[i, j, x] = (np.pi * m[i] / dx) ** 2 + (np.pi * m[j] / dx) ** 2 + (np.pi * m[x] / dx) ** 2


    c = -4.*np.pi/(k)
    c[0, 0, 0] = 0

# Initialize time and time step
    t, h, tmax = 0.0, 0.05, 100
    positions = np.zeros((p.n, 3))  # initialize so that each iteration of particle positions is recorded
    positions[0:p.n, 0] = p.px
    positions[0:p.n, 1] = p.py
    positions[0:p.n, 2] = p.pz

    return g,p,c,t,h,tmax,positions

# data generator function:
def data_gen(g,p,c,t,h,tmax,positions):
    tc = 0
    while t < tmax:
    # assign masses to grid density matrix
        g.density = assign_density(g,p)
    # Fourier transform the density to find the potential
        potentialf = c*np.fft.fftn(g.density)

    # inverse fourier transform this back to
        potential = np.fft.ifftn(potentialf)

    # calculate the gravity vector from the potential field
        Fg = np.gradient(potential) # taking the real values b/c I still get imaginary when inverse Fourier Transforming
        Fgy, Fgx, Fgz = Fg[0]*-1, Fg[1]*-1, Fg[2]*-1
        p.px,p.py,p.pz,p.vx,p.vy,p.vz = step_pvals(g,p,Fgx,Fgy,Fgz,t,h)
        esc = escaped(g,p)
        if esc > 1:
            print('escaped particles = %i' %esc)

        positions[:,0] = p.px
        positions[:,1] = p.py
        positions[:,2] = p.pz

        t += h
        tc += 1
        yield positions,tc,potential,t



def animate(data, h):
    positions, tc, potential,t = data
    # diagnostic iterative plots
    if t%.5 < .001:
        print(potential)
    #     # plot the potential
    #     fig = plt.figure()
    #     ax = fig.add_subplot(111, projection='3d')
    #     # ax.scatter(g.x,g.y,np.real(potential))
    #     p1 = ax.scatter(g.x, g.y, g.z, c=potential, s = .5)
    #     fig.colorbar(p1)
```

```
    #     ax.set_xlabel('x')
    #     ax.set_ylabel('y')
    #     ax.set_zlabel('z')
    #     # ax.set_zlabel('Potential')
    #     plt.title('Potential @ t = %.2f' %t)
    #     ax.set_xlim3d(0, g.l)
    #     ax.set_ylim3d(0, g.l)
    time = np.zeros(np.shape(positions)[0]) + tc * h
    df = pd.DataFrame({"time": time, "x": positions[:, 0], "y": positions[:, 1], "z":
    positions[:, 2]})
    graph.set_data(df.x, df.y)
    graph.set_3d_properties(df.z)
    title.set_text('3D Test, time={}'.format(tc*h))
    return title, graph


def run(g, p, c, t, h, tmax, positions):
    while t < tmax:
        # assign masses to grid density matrix
        g.density = assign_density(g, p)
        # Fourier transform the density to find the potential
        potentialf = c * np.fft.fftn(g.density)

        # inverse fourier transform this back to find spatial potential
        potential = np.fft.ifftn(potentialf)

        # find the gravitational field vectors
        Fg = np.gradient(np.real(potential))  # taking the real values b/c there is complex
     noise
        Fgy, Fgx, Fgz = Fg[0] * -1, Fg[1] * -1, Fg[2] * -1

        # iterate to find the new positions
        p.px, p.py, p.pz, p.vx, p.vy, p.vz = step_pvals(g, p, Fgx, Fgy, Fgz, t, h)
        positions[:, 0] = p.px
        positions[:, 1] = p.py
        positions[:, 2] = p.pz

        esc_time = 0
        esc_num = escaped(g,p)
        if esc_num == p.n:
            esc_time = t
            t = tmax
        else:
            t += h

    return esc_num, esc_time

def gather_velocity_data(vmin,vmax,vrandscale,vstep):
    start = time.time()
    g, p, c, t, h, tmax, positions = initialize()
    p1 = particle(5, [.1,5],[1,1.1],[3,7])
    p1.px = np.copy(p.px) # initial particle values were linking with iteration
    p1.py = np.copy(p.py) # initial particle values were linking with iteration
    p1.pz = np.copy(p.pz) # initial particle values were linking with iteration
    masses = p.m # initial particle values were linking with iteration
    # create list of minimums
    a = np.arange(vmin, (vmax - vrandscale) + vstep, vstep)
    # create list of maximums:
    b = np.arange(vmin + vrandscale, vmax + vstep, vstep)
    # a = np.ones(len(b))*vmin # keep a constant at vmin
    # create zippered list
    c1 = zip(a, b)
    c1 = np.array(c1) #list of tuples that represent velocity ranges
    datapoints = 5
```

```
    esc_num, esc_time = np.zeros((len(a)-1,datapoints)), np.zeros((len(a)-1,datapoints))
    for i in range(len(a)-1):
        for j in range(datapoints):
            # p.reset_pos(init_positions)
            p = particle(5, [.1,5],[1,1.1],[3,7])
            p.m = masses # initial particle values were linking with iteration
            p.px = np.copy(p1.px) # initial particle values were linking with iteration
            p.py = np.copy(p1.py) # initial particle values were linking with iteration
            p.pz = np.copy(p1.pz) # initial particle values were linking with iteration
            p.randv([c1[i,0], c1[i,1]])  # set uniform random velocity dist between [vmin, vmax]
            esc_num[i,j], esc_time[i,j] = run(g,p,c,t,h,tmax,positions) # run w/ velocity dist j times to record evap times
    avg_esc_num, avg_esc_time = np.mean(esc_num,1), np.mean(esc_time,1)
    std_esc_num, std_esc_time = np.std(esc_num,1), np.std(esc_time,1)
    return avg_esc_num, std_esc_num, avg_esc_time, std_esc_time

def gather_mass_data(mmin,mmax,mrandscale,mstep):
    start = time.time()
    g, p, c, t, h, tmax, positions = initialize()
    p1 = particle(5, [.1,5],[.1,1],[3,7])
    p1.px = np.copy(p.px) # initial particle values were linking with iteration
    p1.py = np.copy(p.py) # initial particle values were linking with iteration
    p1.pz = np.copy(p.pz) # initial particle values were linking with iteration
    p1.vx = np.copy(p.vx) # initial particle values were linking with iteration
    p1.vy = np.copy(p.vy) # initial particle values were linking with iteration
    p1.vz = np.copy(p.vz) # initial particle values were linking with iteration
    # create list of minimums
    a = np.arange(mmin, (mmax - mrandscale) + mstep, mstep)
    # create list of maximums:
    b = np.arange(mmin + mrandscale, mmax + mstep, mstep)
    # a = np.ones(len(b))*mmin # keep a constant at mmin
    # create zippered list
    c1 = zip(a, b)
    c1 = np.array(c1) #list of tuples that represent velocity ranges
    datapoints = 5
    esc_num, esc_time = np.zeros((len(a)-1,datapoints)), np.zeros((len(a)-1,datapoints))
    for i in range(len(a)-1):
        for j in range(datapoints):
            # p.reset_pos(init_positions)
            p = particle(5, [.1,5],[1,1.1],[3,7])
            p.px = np.copy(p1.px) # initial particle values were linking with iteration
            p.py = np.copy(p1.py) # initial particle values were linking with iteration
            p.pz = np.copy(p1.pz) # initial particle values were linking with iteration
            p.vx = np.copy(p1.vx) # initial particle values were linking with iteration
            p.vy = np.copy(p1.vy) # initial particle values were linking with iteration
            p.vz = np.copy(p1.vz) # initial particle values were linking with iteration
            p.randm([c1[i,0], c1[i,1]])  # set uniform random mass between [mmin,mmax]
            esc_num[i,j], esc_time[i,j] = run(g,p,c,t,h,tmax,positions) # run w/ mass dist j times to record evap times
    avg_esc_num, avg_esc_time = np.mean(esc_num,1), np.mean(esc_time,1)
    std_esc_num, std_esc_time = np.std(esc_num,1), np.std(esc_time,1)
    return avg_esc_num, std_esc_num, avg_esc_time, std_esc_time

data_gathering = 1
if data_gathering:
    vmin, vmax, vrandscale, vstep = .1, 5, .9 ,.2
    mmin, mmax, mrandscale, mstep = .1, 20, .9, 1
    avg_esc_num, std_esc_num, avg_esc_time, std_esc_time = gather_velocity_data(vmin,vmax,vrandscale,vstep)
    avg_esc_num1, std_esc_num1, avg_esc_time1, std_esc_time1 = gather_mass_data(mmin,mmax,mrandscale,mstep)

    # plot of evaporation time vs. velocity distributions
```

```
    plt.figure()
    plt.title('Evaporation Time vs. Initial Velocity Distribution')
    plt.plot(np.arange(vmin,vmax-vrandscale,vstep), avg_esc_time)
    plt.errorbar(np.arange(vmin,vmax-vrandscale,vstep),avg_esc_time, yerr=std_esc_time,fmt='o')
    # plt.plot(np.arange(vmin+vrandscale,vmax,vstep), avg_esc_time)
    # plt.errorbar(np.arange(vmin+vrandscale,vmax,vstep),avg_esc_time, yerr=std_esc_time,fmt='o')
    plt.xlabel('Base Velocity')
    # plt.xlabel('Maximum Velocity')
    plt.ylabel('Evaporation Time')
    plt.legend(['Variation in Velocity: +%.2f' %vrandscale])
    # plt.legend(['Base Velocity: %.2f' % vmin])

    # plot of evaporation time vs. mass distributions
    plt.figure()
    plt.title('Evaporation Time vs. Initial Mass Distribution')
    plt.plot(np.arange(mmin,mmax-mrandscale,mstep), avg_esc_time1)
    plt.errorbar(np.arange(mmin,mmax-mrandscale,mstep),avg_esc_time1, yerr=std_esc_time1,fmt='o')
    # plt.plot(np.arange(mmin+mrandscale,mmax,mstep), avg_esc_time1)
    # plt.errorbar(np.arange(mmin+mrandscale,mmax,mstep),avg_esc_time1, yerr=std_esc_time1,fmt='o')
    plt.xlabel('Base Mass')
    # plt.xlabel('Maximum Mass')
    plt.ylabel('Evaporation Time')
    plt.legend(['Variation in Mass: +%.2f' %mrandscale])
    # plt.legend(['Base Mass: %.2f' % mmin])



# Initialize Values
g,p,c,t,h,tmax,positions = initialize()
# esc_num, esc_time = run(g,p,c,t,h,tmax)
# potential = diagnostic(g,p,c,t,h,tmax)

# Animate
an = 0
if an:
    time = np.zeros(p.n)
    df = pd.DataFrame({"time": time ,"x" : positions[:,0], "y" : positions[:,1], "z" : positions[:,2]})

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.set_xlim3d(0,g.l)
    ax.set_ylim3d(0,g.l)
    ax.set_zlim3d(0,g.l)

    title = ax.set_title('Simulation')

    data=df[df['time']==0]
    graph, = ax.plot(data.x, data.y, data.z, linestyle="", marker="o")

    ani = am.FuncAnimation(fig, animate, data_gen(g,p,c,t,h,tmax,positions), fargs=(h,), interval=100, save_count=50, blit=True)
```