The Numerical Open-Source Many-Body Simulator (NOMS)

A Senior Project

presented to

the Faculty of the Aerospace Engineering Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science

by

Jason Lloyd Daniel
Javen Kyle Foster-O'Neal

June, 2012

The Numerical Open-Source Many-Body Simulator (NOMS)

Jason L. Daniel* and Javen K. Foster-O'Neal†
California Polytechnic University, San Luis Obispo, CA, 93405, United States

This paper outlines the setup and creation of an object-oriented N-body simulator as part of a continued project to explore physical phenomenon and human-computer natural interaction technologies. The tools and processes required to build an N-body simulator are also included. Several integrators were evaluated based on their ability to maintain system energy The 2nd order integrator Verlet and 3rd order integrator Hermite algorithms had the greatest accuracy to model large-scale N-body dynamics for their given computation time. Other algorithms required significantly shorter time steps to achieve similar short-term accuracy. At present, NOMS can reasonably simulate 10,000 particles at less than one minute per iteration.

Nomenclature

$\mathbf{a}$	Acceleration vector
a	Acceleration magnitude
δt	Time increment
E	Change in total system energy between time steps
E_0	Initial total energy of the system
F	Force
G	Universal gravitation constant
$\mathbf{H}$	Angular momentum
h	Time interval
$\mathbf{j}$	Jerk (1st derivative of acceleration)
$\mathbf{k}$	Snap (2nd derivative of acceleration) correction factor
k	Time increment
$\mathbf{k_r}$	Runge-Kutta position derivative coefficient
$\mathbf{k_v}$	Runge-Kutta velocity derivative coefficient
$\mathbf{l}$	Crackle (3rd derivative of acceleration) correction factor
m	Mass
N	Number of particles
$\mathbf{r}$	Position vector
r	Distance
t	Time
U	State
$\mathbf{v}$	Velocity vector
v	Velocity magnitude
Subscript	
c	Corrected (predictor-corrector method)
p	Predicted (predictor-corrector method)

I. Introduction

Isaac Newton's law of gravitation states that two bodies with mass are gravitationally attracted to each other by a force. The strength of the force decays with the square of the distance between the bodies and is proportional to the product of the masses of the bodies. The law of gravitation is summarized in Eqn. 1.

*Researcher, Aerospace Engineering, 1 Grand Ave, San Luis Obispo CA 93405.
†Researcher, Aerospace Engineering, 1 Grand Ave, San Luis Obispo CA 93405.

$$F_{grav} = \frac{Gm_1m_2}{r^2} \tag{1}$$

With the addition of Newton's second and third laws of motion, we can derive an expression for the acceleration of each particle in a gravity field (Eqn. 2). To avoid an extra N floating point operations, N-body simulators typically scale units such that the universal gravitation constant G = 1.

$$\mathbf{a}_i = G \sum_{j=1, j\neq i}^{N} \frac{m_j(\mathbf{r}_j - \mathbf{r}_i)}{|\mathbf{r}_j - \mathbf{r}_i|^3} \tag{2}$$

To date, no closed-form general solution exists for $N > 2$. In order to solve the above equation, acceleration is integrated twice to yield the position and velocity at the next time step. Fortunately, this equation is well-suited to numerical computation. Acceleration is only a function of the particles' positions, which reduces the amount of data that must be passed around. The acceleration and velocity of one body is independent of the acceleration and velocity of another body. This makes the task well-suited for parallelization.

For collisionless systems, a softening parameter ϵ is often added to numerical gravitation simulators, which attenuates the gravitational force between two nearby objects. Without a softening parameter, the divergence of $\mathbf{a_i}$ requires extremely small time steps δt, which can "bring integration virtually to a halt."[3] In this case, the addition of the softening parameter reduces the realism of the simulator.

Without a softening parameter, the system can diverge and gain total energy without continuous time step integration.[3] At the same time, the addition of a softening parameter is not that poor of an approximation for stellar dynamics. N-body simulations often use point masses to model bodies with distributed mass (like stars) for computational simplicity, the addition of a softening parameter enhances the realism of the system by attempting to model a star's mass distribution.[3]

If the time step is too large after one iteration, the particles will pass each other with a velocity in the opposite direction of the barycenter. If the distance between the two particles after one time step is too large, the gravitational force may be too weak to bring the particles back together. This nonphysical behavior may result a "loss" of particles within the system.

Equations 3 and 4 are two examples of softened versions of Eqn. 2, adapted from Binney and Tremaine. In Eqn. 3 and Eqn. 4, particles that are closer than a distance ϵ have a reduced gravitational attraction. Particles that are much greater than a distance ϵ are virtually unaffected from this smoothing parameter. Equation 4 is more computationally expensive, but attempts to model each body as a sphere with radius ϵ rather than as a point mass. For clarity, the substitution $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is used in these equations.

$$\mathbf{a}_i = G \sum_{j=1, j\neq i}^{N} \frac{m_j(\mathbf{r}_i - \mathbf{r}_j)}{\left(r_{ij}^2 + \epsilon^2\right)^{3/2}} \tag{3}$$

$$\mathbf{a}_i = G \sum_{j=1, j\neq i}^{N} \frac{m_j(r_{ij}^2 + \frac{3}{2}\epsilon^2)(\mathbf{r}_i - \mathbf{r}_j)}{\left(r_{ij}^2 + \epsilon^2\right)^{3/2}(r_{ij}^2)} \tag{4}$$

Even without an exact solution to ordinary differential equation Eqn. 2, it is still possible to measure the error of each integrator and observe the effects of adding a softening parameter. Several common values to look at are the variation in total energy (kinetic and potential energy), the linear momentum of the system, and the angular momentum of the system. All of these quantities should be constant for conservative systems. Symplectic integrators, like leapfrog, are useful for modeling systems where total energy is the primary parameter of concern, whereas non-symplectic integrators, like Runge-Kutta, are useful for modeling systems where particle position error is the primary parameter of concern. Position error can be measured by comparing the solution from an integrator with the 2-body exact solution. Each integration scheme has its own signature traits in relation to gravitational simulations, and is exhibited with a changing argument of periapsis, inclination, or semi-major axis.

The goal of NOMS is to simulate any meshless particle model, including particle physics, plasma, rarefied flow, electrostatics, and magnetic fields. Each of these problems can be solved with minor adaptations to an N-body simulator.

II. Initial Planning and Setup

Initial planning for the project began with choosing the programming language and computational environment. Due to previous experience, computational efficiency, and object oriented features, C++ was chosen as the primary language for the code. Linux was selected as the development platform due to on-hand compuational resources in the lab and the variety open source tools that a Linux environment offers. The code is platform-independent and may be run on other operating systems with minor modifications.

The schedule for the first eleven weeks is shown in Fig. 1. The first phase of the project focused on researching existing N-body codes and designing an architecture that would provide flexibility for the code to be used for further research with compile-time and run-time algorithm-selection optimization, stellar collision modelling, and a novel user interface. After an extensive literature review, program architecture was devised and some early code was written. Due to the unexpected scale of the N-body problem and proper code design, each of theses aspects required additional time beyond what was allocated in the schedule.

	2-Jan	9-Jan	16-Jan	23-Jan	30-Jan	6-Feb	13-Feb	20-Feb	27-Feb	5-Mar	12-Mar
	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11
Literature Review											
CUDA research											
Code Structure											
Preliminary Review											
N-Body Development											
Visualization/ Post Processing											
N-Body Validation											
Code Review											
Stellar Physics Implementation											
Code Review											
Validation											
Final Report Work											
Final Review											

Figure 1. Project Schedule for First 11 Weeks

The schedule for the second portion of the project is shown in Fig. 2. The primary focus for the second phase was to write code, validate N-body physics, and generate documentation. Again, due to unexpected bugs and complexity, stellar physics and the validation of these physics was not implemented, however a variety of useful tools for future code development were created which will aid in the progress of the project.

	19-Mar	26-Mar	2-Apr	9-Apr	16-Apr	23-Apr	30-Apr	7-May	14-May	21-May	28-May	4-Jun
	Week 12	Week 13	Week 14	Week 15	Week 16	Week 17	Week 18	Week 19	Week 20	Week 21	Week 22	Week 23
Literature Review												
CUDA research												
Code Structure												
Preliminary Review												
N-Body Development												
Visualization/ Post Processing												
N-Body Validation												
Code Review												
Stellar Physics Implementation												
Code Review												
Validation												
Final Report Work												
Final Review												

Figure 2. Project Schedule for First 11 Weeks

A. Code Structure

In pursuit of an object-oriented apporoach, a considerable amount of initial planning was performed in order to have a strong understanding of what each class would be responsible for and how it would interact with other classes.

In order to help with the creation and visualization of the code structure, Unified Markup Language (UML 2.0) was used. Figure 3 is a UML class diagram of the class relationships in NOMS.

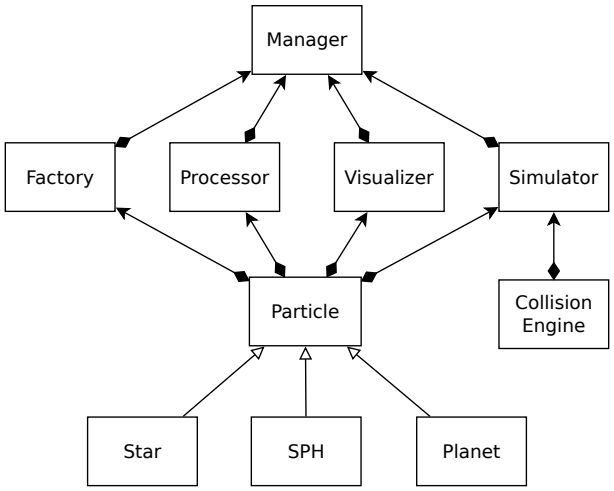

Figure 3. Diagram Showing Class Inheritance

1. Manager

The manager is the top-level object in NOMS. Its job is to create a simulator and factory object (and in the future, a visualizer object) and perform file input and output.

Table 1. Description of the Manager's methods

Method	Description
Manager();	Manager constructor
~Manager();	Manager destructor
bool readSimFile(…);	Reads a simulator file into memory
bool writeSimFile(…);	Writes the simulator settings to disk
bool readParticleFile(…);	Reads a particle file into memory
bool writeParticleFile(…);	Writes the particles at the current time step to disk
bool writeStatFile(…);	Writes statistics (time step, energy, momentum, center of mass location, and total system mass) from the current iteration to disk
void advanceOneTimestep(…);	Runs the simulator forward one iteration using the specified integration scheme
void freeParticles();	Commands the factory to free dynamically allocated particles
void scaleUnits();	Scales the units to avoid exceeding the floating point exponent dynamic range
FLOAT updateDynDt();	Calculates an appropriate adaptive time step for the current iteration and updates dt accordingly
FLOAT getDt();	Returns the current simulator time step dt
void setDt(…);	Sets the current simulator time step dt
void getStats(….);	Calculates statistics from the current iteration and writes them to the vectors provided as function inputs.
bool setSimParameter(…);	Sets the simulator parameter from a parameter-value pair from readSimFile
void printSimParameter(…);	Prints a simulator parameter-value pair to the screen

2. Factory

The factory is responsible for all memory management tasks–including allocation and deallocation of particles. The factory will be responsible for copying data to and from CPU and GPU memory as well as building an octree when those features are implemented.

Table 2. Description of the Factory's methods

Method	Description
Factory()	Factory constructor
~Factory();	Factory destructor
Particle** makeParticles(...);	Calls Particle constructor for specified particle type and number
Particle** allocateParticleList(...);	Allocates memory for the Particle list
Particle* allocateParticle(...);	Allocates memory for an individual particle
float3* createFloat3Array(...);	Allocates memory for an array of Float3 structs
void destroyFloat3Array();	Deallocates memory for an array of Float3 structs

3. Processor

The processor is used for generating system statistics and calculating the appropriate time step for the simulation. System statistics include total energy, kinetic energy, potential energy, center of mass position, center of mass velocity (linear momentum), angular momentum, total system mass, and time step.

Table 3. Description of the Processor's methods

Method	Description
Processor();	Processor constructor
~Processor();	Processor destructor
FLOAT calcDynamicTimeStep(..) const;	Calculates dynamic time step
FLOAT calcKineticEnergy(...) const;	Calculates system kinetic energy
void dispKE(...) const;	Displays the calculated kinetic energy
FLOAT calcPotentialEnergy(...) const;	Calculates the system potential energy
void dispPE(...) const;	Displays the calculated potential energy
float3 calcAngularMomentum(...) const;	Calculates the system angular momentum
void dispAngMom(...) const;	Displays the calculated angular momentum
FLOAT calcTidalEnergy();	Calculates the system tidal energy (currently a stub)
float3 calcCG(...);	Calculates the system center of mass
void centerAtCG(...);	Translates coordinates to center of mass
void scaleMasses(...);	Scales system masses to sum to one
void scaleToNbodyUnits(...);	Scales system to N-Body units

4. Visualizer

The visualizer will be able to be used both online and offline as a graphical front-end to the user, displaying both the viewport (a rendering of the current iteration) and corresponding statistics. With the addition of a simple graphical front end and eventually a Natural Interface, the user will be able to move through the simulation space. The graphical interface will also allow real-time modification of important parameters like the universal gravitation constant or the time step. Offline, the visualizer can be used to generate images and videos from a set of particle files and statistics files.

Currently, data visualization is accomplished by generating particle and statistics files from NOMS and visualizing the results in MATLAB. Visualization and plotting will evenutually be implemented in C++ using a graphical library, allowing the application to display real-time results, plot integration error over time, and display a particle's state vector at run-time within a graphical user interface.

5. Simulator

The simulator is the core of NOMS. The heaviest functions are in the simulator, and thus optimizing these functions are most important. These include calculating the fractional acceleration contribution by each body on a single body as well as integrating acceleration to get to the next time step.

Table 4. Description of the Factory's methods

Method	Description
Simulator();	Simulator constructor
~Simulator();	Simulator destructor
void update(...);	Updates all particle positions and velocities
void ForwardEuler(...);	Forward Euler integration algorithm
void Verlet(...);	Verlet integration algorithm
void RungeKutta4(...);	Runge-Kutta 4 integration algorithm
void RungeKuttaFehlberg(...);	Runge-Kutta-Fehlberg integration algorithm (broken)
void RungeKutta5(...);	Runge-Kutta 5 integration algorithm (broken)
void Hermite(...);	Hermite integration algorithm
void SingleForwardEuler(...);	Single particle forward Euler integration
void SingleVerlet(...);	Single particle Verlet integration
void SingleRungeKutta4(...);	Single particle Runge-Kutta 4 integration
void SingleRungeKuttaFehlberg(...);	Single particle Runge-Kutta-Fehlberg integration
void SingleRungeKutta5(..);	Single particle Runge-Kutta 5 integration
void HermitePredictor(...);	Hermite integrator prediction step
void HermiteCorrector(...);	Hermite integrator correction step
float3 OneBodyAccel(...);	Acceleration calculation on one body
float3 OneBodyAccelUsingTempPos(...);	Acceleration calculation on one body using temporary position
float3 bodyBodyInteraction(...);	Force interaction between two bodies
float3 bodyBodyInteractionUsingTempPos(...);	Force interaction between two bodies using temporary position
float3 OneBodyJerk(...);	One body jerk calculation
float3 OneBodyJerkUsingTempPos(...);	One body jerk calculation using temporary position

6. Particle

Each particle has a mass, radius, position, and velocity. Particle subclasses may contain additional information specific to its type.

Table 5. Description of the Particle's methods

Method	Description
Particle();	Particle constructor
~Particle();	Particle destructor
Particle(...);	Alternate particle constructor with defined member variables
void update(...);	Update particle position and velocity
void displayStatePretty() const;	Display information about a particles member variables

7. Star, SPH, and Planet

These three objects inherit the data members from the Particle class. The child objects include rendering parameters and interaction settings.

B. Makefile

The project is built using Makefiles. The Makefiles provide several build targets, including a release build, a debug build, and a profile build. Per `make` convention, two Makefiles were used: one in the project root directory and one in the project `src/` directory. The former Makefile calls the latter Makefile. The latter Makefile compiles the source files with the given build targets below. Dependencies are explicitly defined within the Makefile so that object files are built with the necessary external methods and members.

For this project, the `g++` compiler was used, which compiles the source for the host platform. The release build is the default build, which compiles the source code with compiler optimizations. The debug build defines the `DEBUG` pre-processor flag and outputs extra information to the command line to assist the development process. The profile build compiles the source code with extra profiling information, which generates a file called `gmon.out` after the program executes. The `gmon.out` file can be examined with the program `gprof` afterwards to determine which functions were called the most and which functions took the longest amount of time to execute.

C. Memory Leaks

A software tool called `valgrind` was used to check for memory leaks in the program. Memory leaks can occur anywhere memory is dynamically allocated using either `malloc()` (C-style) or `new` (C++-style). In order to simplify memory management, constructors and destructors were used, so memory is implicitly freed when the parent class goes out of scope. The Factory can also explicity free memory if needed.

D. Software Version Control

Because this project was developed by three people simultaneously, it was necessary to establish a revision control system. Services like Dropbox were not suitable for this project because of limited version history and poor version conflict resolution. Additionally, this was an opportunity to learn how to set up and use a real version control system. For this project, Subversion was selected due to its available features and extensive community support.

E. Integrated Development Environment

Initially, NOMS was written using a text editor, but it quickly became apparent that development would be faster and easier with an integrated development environment. For this project, Eclipse C++ was selected, which is freely available, has debugging tools not available in a text editor, and has a large community for support.

F. Remote Access

Remote access between computers was accomplished by using the GNU programs `ssh` and `scp`, which allowed one person to generate data files and another person to analyze those files.

G. Code Performance

Two tools were used to measure how fast NOMS ran: `gprof` and `time`. `gprof` uses a compiler option that injects timing code around each function, then analyzes how much time was spent running each method and how many times each method was called during program execution. `time` measures the elapsesd time, total processor time, and total system time when executing a program. Unlike `gprof`, `time` does not slow down execution speed.

III. Input and Output Files

A. Input Files

There are two types of input files that NOMS used to run a simulation: the simulation input file and the particle input file. The input files were separated so that a simulator file could be reused for multiple initial particle arrangements or a different simulator file could be run from the end of a previous simulation without the need for modifying files.

These files are all human-readable in order to make the results more portable to external applications (like MATLAB or Excel) for further analysis. Plain text files are also easy for humans to edit values for what-if scenarios (like doubling a star's mass, or changing the time step). This also enhances the usability of NOMS as a framework for other particle simulations.

1. Simulation Input File

The simulation input file specifies the values of the simulator environment. These settings include graphical display properties, what resource limitations the simulator has, what level of debug or statistics information NOMS should output, the maximum number of iterations NOMS should run before exiting, what softening radius should be used,

what integration method should be used, how often output files should be generated, and whether the simulation runs on the CPU or GPU.

In order to simplify the simulator input file, all simulator variables are initialized to their default values. Thus, the simulator input file only needs to contain non-default values. The Simulator will ignore any parameter it does not recognize.

The simulation input file must meet the following specifications:

1. One parameter name and value per line.
2. Parameter name and value are separated by an equals sign and white space (space or tab).
3. Parameter names and values are not case sensitive.
4. If a parameter is not specified in the input file, the default value is used.
5. If a value is of string type, the value will be surrounded in ”double quotes.”
6. If a value is of Boolean type, the value will be either 0 or 1

Below is an example simulator input file.

```
guiWidth      = 640
guiHeight     = 1024
guiFov        = 40.0
guiEnableAtm  = 0
integrator    = "RungeKutta4"
```

2. Particle Input File

The particle input file contains a list of all particles at a particular time step. The file specifications are given below. Currently, the particle input file format cannot be used to represent multiple simulations in the event of a collisional sub-simulation.

1. The first line shall begin with a # character and is used for comments.
2. Multiple commented lines (each beginning with a #) may be used at the top of the file.
3. The second line shall contain the number of particles.
4. The third line shall contain units in the order mass, distance, time
5. One particle per line in the following format (whitespace separated). otherProperties is reserved for future use.
   `type mass radius pos.x pos.y pos.z vel.x vel.y vel.z otherProperties`

Below is an example particle input files.

```
# Small solar system.
# The first line is Sun.  The second line is Earth.
# Everything is given in planck units.
# type   mass       radius     pos.x      pos.y pos.z vel.x vel.y      vel.z
2
mp lp tp
Particle 4.3289e+22 1.1241e-29 0.0        0.0   0.0   0.0   0.0        0.0
Particle 1.3003e+17 1.0308e-31 2.4178e-27 0.0   0.0   0.0   8.9355e+09 0.0
```

B. Output Files

NOMS can currently generate three types of output files. The simulator output file is used to write the current simulator settings to disk, the particle output file is used to write the current particle positions and velocities to disk, and the statistics output file is used to track overall system-level trends. In the future, a fourth output file will be added, which tracks the location of a single particle through time.

1. Simulator Output File

NOMS can output the current simulator settings to a file, which is useful if the settings change during execution. The goal for this project is to have all controls adjustable by the user during runtime within a graphical user interface. NOMS can output either only the settings with non-default values or all the simulator settings.

2. Particle Output File

The format of the particle output file is the same as the format of the particle input file. This file is useful for storing the results of a simulation for offline rendering or post-processing within another application. The particle output file can also be used to resume a previous simulation.

3. Statistics Output File

The processor can calculate the total energy, kinetic energy, potential energy, center of gravity position, center of gravity velocity, total system mass, and angular momentum of the system, which can be written to file. The goal is to plot this information within NOMS, so that the user can immediately know if their simulation is divergent due to an inappropriate step size, smoothing radius, or integration algorithm. Each line in the statistics file summarizes the system properties for one iteration. Currently, statistics files are analyzed with MATLAB.

Below is an example statistics output file.

```
dt      KE         PE          TotalEnergy  H       CG.x    CG.y     CG.z
0.006   0.250334   -0.745356   -0.495023    1.5000  0.0005  -2.0000  -2.5000
0.006   0.250667   -0.745357   -0.494689    1.5000  0.0010  -2.0000  -2.5000
0.006   0.251002   -0.745358   -0.494356    1.5000  0.0015  -2.0000  -2.5000
0.006   0.251336   -0.745359   -0.494023    1.5000  0.0020  -2.0000  -2.5000
0.006   0.251671   -0.745361   -0.493690    1.5000  0.0025  -2.0000  -2.5000
                                  .........
```

IV. Integrators

Large scale N-Body simulations require an efficient numerical integrator to accurately predict the motion of each body and keep simulation run times low. Various classes of integrators have been developed that differ based on inputs, outputs, number of time steps, intermediate steps, and order of accuracy.

A. One-Step Methods

One-step integration methods approximate the next state ($U^(n+1)$) using only information from the current state ($U^(n)$)[1]. One-step methods can be usefull for problem that are memory bound since the program does not need to save previous state information. These methods are also refered to as "self-starting" because only initial state information is required to begin propagating a solution. Time steps can be changed at any time which improves the flexibility of the method and allows for simpler implementation of variable time steps. These methods also have the benifit of retaining thier order of accuracy when discontinuities in the derivitives are encountered.

1. Forward Euler

$$\frac{U^{n+1} - U^n}{k} = f(U^n) \tag{5}$$

The forward Euler method is a very simple integration method, calculating the derivative as the difference between two states divided by the grid size or time step for spacial and temporal discretization respectively. Forward Euler is a 1st order accurate time marching method that benifits from simple implementation but is error-prone due to a limited stability region.[1]

For implementation within the N-body context, the Forward Euler method is:

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \delta t(\frac{1}{2}\mathbf{a}_i\delta t + \mathbf{v}_i) \tag{6}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i\delta t \tag{7}$$

B. Multi-Step Methods

Multi-step methods use multiple state values to determine a derivative with increased accuracy. The added accuracy does come at the cost of processor time, for the added operations, and memory, for storing the other states.

1. Verlet Method (Leapfrog)

Verlet method, also known as modified leapfrog discretization or midpoint method, is a multi-step method that uses the $n+1$ and $n-1$ state vectors to approximate the derivative for the nth time step. This is a spatial discretization method.

$$\frac{U^{n+1} - U^{n-1}}{2k} = f(U^n) \tag{8}$$

The Verlet method is an explicit 2-step method with second order accuracy that requires information of the previous state. The Verlet method is grouped in a class known as symplectic integrators which were developed to conserve energy and are common in astrophysical simulations.[3]

In the application of the Verlet method to the N-Body problem is the same as the forward Euler method for the velocity profile and particle positions are updated in the following way,

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \delta t \left[\frac{1}{2}\mathbf{a}_i \delta t + \frac{1}{2}(\mathbf{v}_i + \mathbf{v}_{i+1}) \right] \tag{9}$$

An example of the time discretization version of leapfrog for the advection equation is:

$$\frac{U_j^{n+1} - U_j^{n-1}}{\frac{ak}{h}} = -(U_{j+1}^n - U_{j-1}^n) \tag{10}$$

This method is second order accurate in both space and time. The method is stable when $|ak/h| < 1$. The drawback to this method is that it is a 3-level method, which can become computationally expensive.[1,2]

2. Predictor-Corrector 4th Order (Hermite Integration)

The predicted position and velocity of an object, $\mathbf{r}_p$ and $\mathbf{v}_p$ respectively, are calcuated at each time step from the previous position, velocity, acceleration, and rate of change of accleration (i.e. jerk). The position, velocity, and acceleration of particle i with respect to particle j are:

$$\mathbf{r}_{ji} = \mathbf{r}_j - \mathbf{r}_i \tag{11}$$

$$\mathbf{v}_{ji} = \mathbf{v}_j - \mathbf{v}_i \tag{12}$$

$$\mathbf{a}_{ji} = \frac{m_j}{r_{ji}^3}\mathbf{r}_{ji} \tag{13}$$

The jerk is calculated as follows:

$$\mathbf{j}_{ji} = \frac{m_j}{r_{ji}} \left[\mathbf{v}_{ji} - 3\frac{v_{ji} \cdot r_{ji}}{r_{ji}^2}\mathbf{r}_{ji} \right] \tag{14}$$

The acceleration and jerk for each particle i are given by

$$\mathbf{a}_i = \sum_{j \neq i} \mathbf{a}_{ji}, \quad \mathbf{j}_j = \sum_{j \neq i} \mathbf{j}_{ji} \tag{15}$$

With the acceleration and jerk for each particle summed up for all N, the actual predicted position and velocity are shown below:

$$\mathbf{r}_p = \mathbf{r} + \mathbf{v}\delta t + \frac{1}{2}\mathbf{a}\delta t^2 + \frac{1}{6}\mathbf{j}\delta t^3 \tag{16}$$

$$\mathbf{v}_p = \mathbf{v} + \mathbf{a}\delta t + \frac{1}{2}\mathbf{j}\delta t^2 \tag{17}$$

These predicted positions and velocities are then used to find the predicted accelerations and jerks, as indicated above. The 2nd and 3rd derivatives of the acceleration (known as snap and crackle) are calculated using the predicted accleration and jerk as shown:

$$\mathbf{k} \equiv \frac{1}{2}\mathbf{a}^{''}\delta t^2 = 2(\mathbf{a} - \mathbf{a}_p) + \delta t(\mathbf{j} - \mathbf{j}_p) \tag{18}$$

$$\mathbf{l} \equiv \frac{1}{2}\mathbf{a}^{'''}\delta t^2 = -3(\mathbf{a} - \mathbf{a}_p) + \delta t(2\mathbf{j} + \mathbf{j}_p) \tag{19}$$

The higher-order derivitive correction factors $\mathbf{k}$ and $\mathbf{l}$ correct the predicted position and velocity. The corrected position and velocity for particle i is:

$$\mathbf{r}_c = \mathbf{r}_p + \left(\frac{1}{12}\mathbf{k} + \frac{1}{20}\mathbf{l}\right)\delta t^2 \tag{20}$$

$$\mathbf{v}_c = \mathbf{v}_p + \left(\frac{1}{3}\mathbf{k} + \frac{1}{4}\mathbf{l}\right)\delta t \tag{21}$$

Overall, this method is 3rd order accurate.[4]

C. Multi-Stage Methods

Multi-stage methods require multiple iterative steps to calculate the next time step in the integration. These methods do not use previous state data, alleviating issues with memory storage; however, the multiple iterations can be compuationally expenisve for each time step.

1. 4th Order Runge-Kutta

The 4th order Runge-Kutta method is a staple in explicit numerical integration. The technique was originially developed in 1900 by C. Runge and M.W. Kutta. There are several versions of the 4th order coefficients that describe this multi-stage method. For the purposes of N-body integration, the following implementation was used:

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \frac{h}{6}\left[\mathbf{k}_{v_1} + 2(\mathbf{k}_{v_2} + \mathbf{k}_{v_3}) + \mathbf{k}_{v_4}\right] \tag{22}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{h}{6}\left[\mathbf{k}_{r_1} + 2(\mathbf{k}_{r_2} + \mathbf{k}_{r_3}) + \mathbf{k}_{r_4}\right] \tag{23}$$

Where $\mathbf{k}_{v_n}$ and $\mathbf{k}_{r_n}$ are solved simultaneously for each stage.

$$\begin{aligned}
\mathbf{k}_{v_1} &= \mathbf{a}(\mathbf{r}_i) \\
\mathbf{k}_{v_2} &= \mathbf{a}(\mathbf{r}_i + \mathbf{k}_{r_1}\tfrac{h}{2}) \\
\mathbf{k}_{v_3} &= \mathbf{a}(\mathbf{r}_i + \mathbf{k}_{r_2}\tfrac{h}{2}) \\
\mathbf{k}_{v_4} &= \mathbf{a}(\mathbf{r}_i + \mathbf{k}_{r_3}h) \\
\mathbf{k}_{r_1} &= \mathbf{v}_i \\
\mathbf{k}_{r_2} &= \mathbf{k}_{v_1}\tfrac{h}{2} \\
\mathbf{k}_{r_3} &= \mathbf{k}_{v_2}\tfrac{h}{2} \\
\mathbf{k}_{r_4} &= \mathbf{k}_{v_3}h
\end{aligned} \tag{24}$$

Higher-order Runge-Kutta schemes exist, including the Runge-Kutta-Fehlberg algorithm which is 5th order accurate, but comes with a performance penalty. While higher-order integration schemes are normally valued for solving differential equations, evaluating the derivative quantity $f(U^n)$ is prohibitively expensive for N-body simulations because it requires $O(N^2)$ comparisons between particles. Furthermore, Runge-Kutta schemes are not symplectic integrators, which means they do not conserve energy. Therefore their utility in modeling globular clusters is minimal.[1 2 3]

2. Modified Runge-Kutta Explicit Time Stepping

The modified Runge-Kutta explicit time stepping function was specifically designed to handle Navier-Stokes equation. The Nadarajah-Jameson study used a solver named FLO103, which incorporated the Jameson-Schmidt-Turkel (JST) scheme for artificial dissipation. FLO103 also uses local time stepping, an implicit residual smoothing principle, and multiple grids.[5]

D. Adaptive Time Steps

Adaptive time steps are particular effective for stiff ODEs or stochastic differential equations (SDEs). As a simulation progresses, particles may move towards or away from each other. The optimal time step δt, given in Eqn. 25, changes throughout the simulation, and adaptive time-stepping is one solution to this problem. With the implementation of adaptive time steps, the simulator can use larger time steps when particles are far apart or moving slowly and smaller time steps when particles are closer together or moving quickly.

With adaptive time steps δt for every particle in the simulation changes with each iteration. However, because computing an optimal δt requires N^2 comparisons (for particles stored in a linear array) and at best $NlogN$ comparisons (for a good octree implementation)for a simulation with N particles, changing the time step each iteration is expensive (Appendix B.).

As mentioned previously, the simulation can be brought to a screeching halt with very small δt. For this reason, the smoothing parameter or other variable time step methods should be used.

$$\delta t = \min_{i,j} \left(\eta \frac{|\mathbf{r}_i - \mathbf{r}_j|}{|\mathbf{v}_i - \mathbf{v}_j|} \right) \tag{25}$$

V. Results

A. Integrator Comparison

When analyzing an N-body simulator, an important factor is energy conservation of the system. A common parameter used to look at energy conservation is the energy error, given in Eqn. 26, which has unity ideal energy error for a system. The energy error was evaluated for a two-body test case using different time step criteria. The setup simulated two particles: a small particle in a circular orbit around another particle with 100 times the mass of the small particle. The system was given zero initial linear momentum. Constant and variable time steps were used and adaptive time steps, as defined in Eqn. 25, included variation of η. Using the Verlet integrator, the energy error was tracked using the different time step criteria. The results of this study are shown in Fig. 4. A plot of the trajectory of the less massive orbiting particle is shown in Fig. 5 for each time step case and shows a comparison between the physical position of the body and how the energy error of the system varies over time.

$$EnergyError = \frac{E}{E_0} \tag{26}$$

From Fig. 4, it is clear that the constant time step of 1.0 and the variable time step with $\eta = 1.0$ do not trace a circular orbit and do not have well-behaved energy errors. These two cases are a poor approximation of the orbit and should not be used in a simulation. All of the variable time step methods converge to a single value for the energy error. The case with $\eta = 0.05$ appears to converge close to the ideal case of 1.0. Constant time steps caused the orbit to precess. Larger time steps resulted in greater orbit precession, which can be seen in the spiral pattern traced out with a constant time step of 1.0. For this particular two-body situation, a time step on the order of 0.1 to 0.01 yields expected circular behavior of the particle.

A comparison of the energy error for the Verlet, Hermite, and Forward Euler integration methods using different time steps are shown in Figs. 6 through 9. These plots illuminate some of the inherent behaviors of each integrator. The Forward Euler method consistently droped to a certain energy error and then leveled off. If the time step was too large the Hermite method behaved similarly to the Forward Euler method, as shown in Fig. 6, likely a result of a very poor predictive step. The Verlet method showed a consistant oscillation about a fixed value, and the oscillations dampened when a variable time step was used. For all cases except the constant time step of 0.01, shown in Fig. 8, the Verlet and Hermite methods had out of phase energy error oscillations.

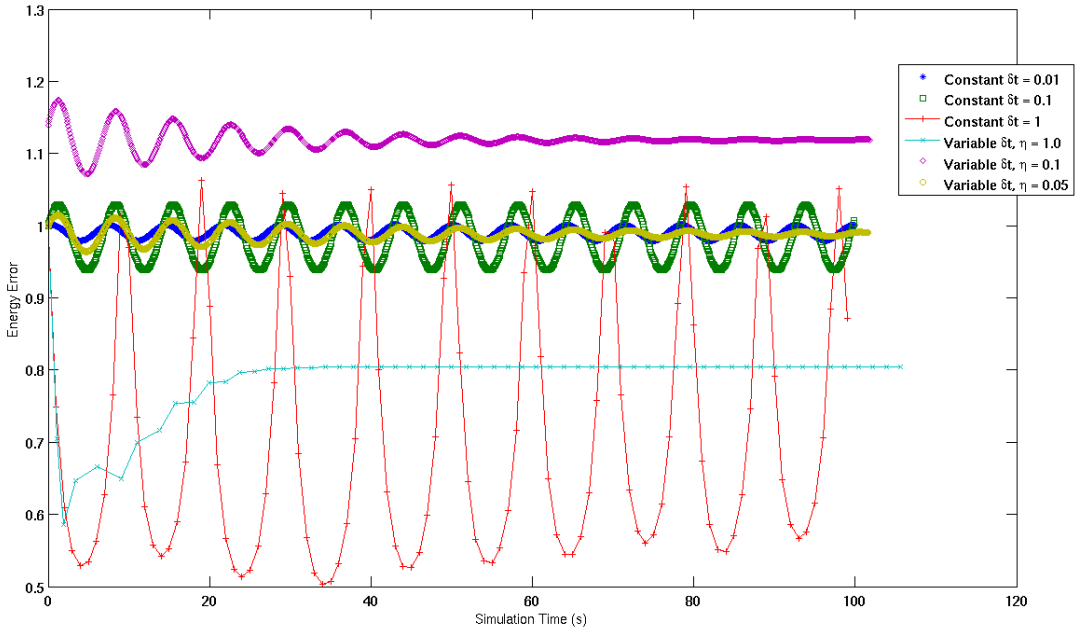

Figure 4. Different time steps using the Verlet integrator for nearly 100 seconds of simulation time show how time step affects energy error.

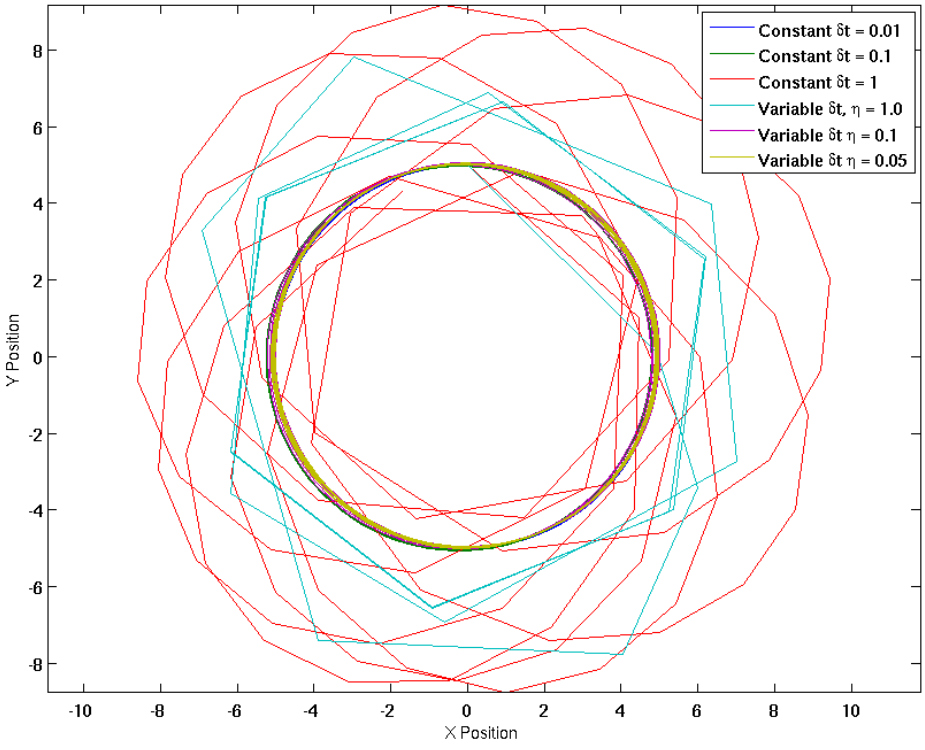

Figure 5. Orbits of the second (light) body for the two body case using different time steps and the Verlet integrator.

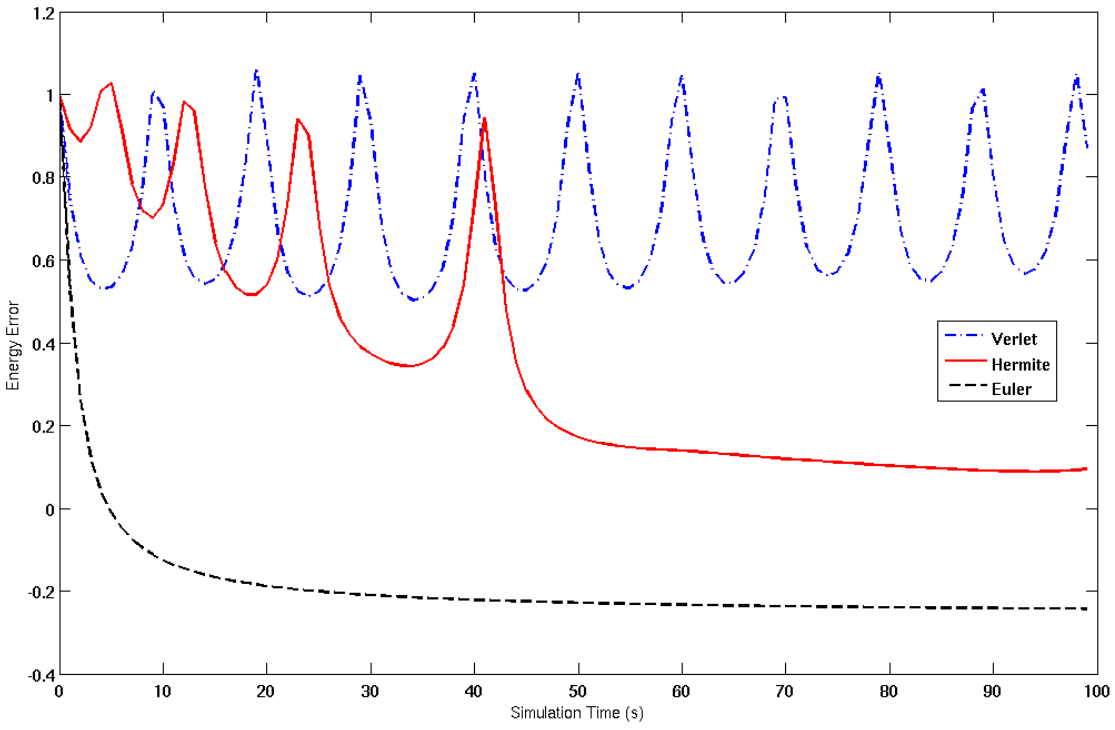

Figure 6. A comparison of energy error for various integration methods using a constant time step of 1 second

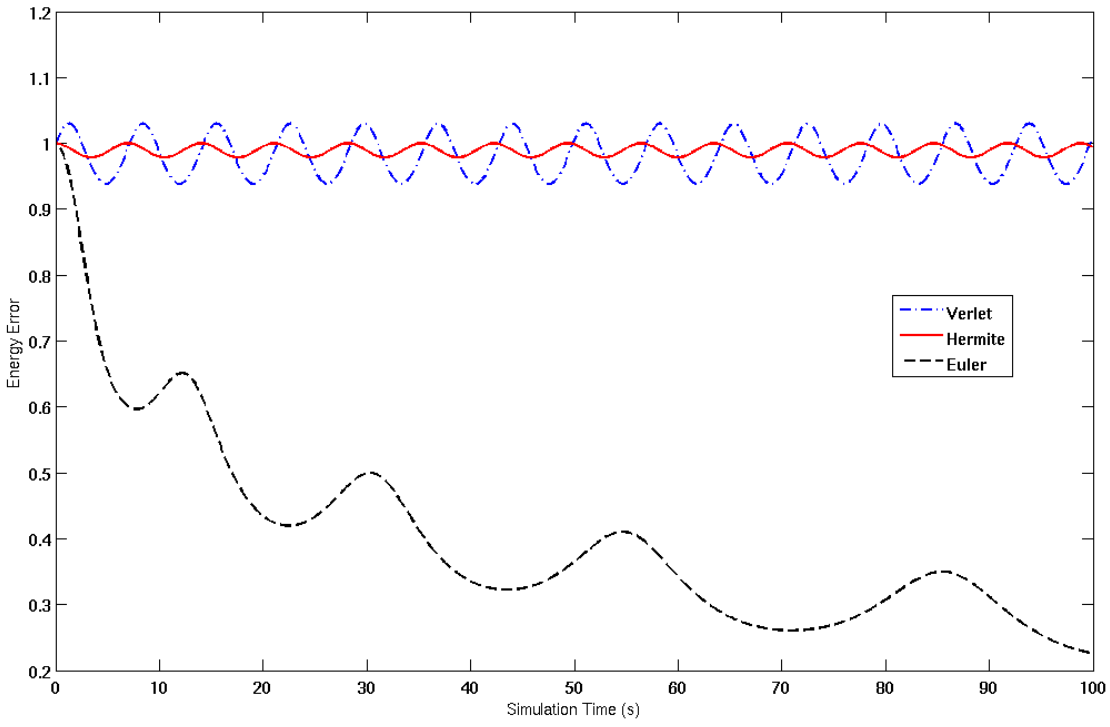

Figure 7. A comparison of energy error for various integration methods using a constant time step of 0.1 second.

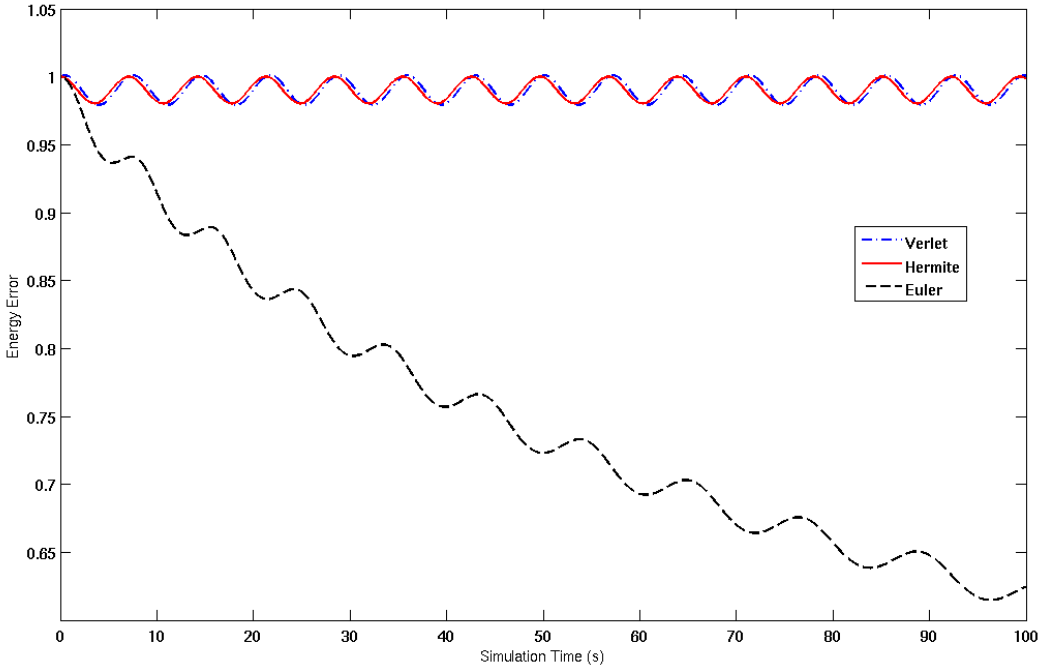

Figure 8. A comparison of energy error for various integration methods using a constant time step of 0.01 second.

B. Run Time Performance

Simulation execution time for the Forward Euler, Verlet, Hermite and Runge-Kutta methods for varied number of bodies is shown in Fig. 10. Runge-Kutta had the worst performance due to the multiple integration steps that are required by the method. Hermite had similar execution times to Runge-Kutta specifically for larger numbers of bodies. The Forward Euler and Verlet methods had nearly equal run times for each number of bodies simulated. The graph also shows that the force calculation algorithm runs in time $O(N^2)$.

VI. Conclusion

A. Integrator Choice

From the energy error study and the run time performance of each integration method it is clear why the Verlet method is commonly used in N-body simulations. The Verlet method had good energy conservation relative to its computational cost. The Hermite method had comperable energy conservation than Verlet when a variable or constant time step was used but requires more processing time. While it was the fastest method, the Forward Euler method performed poorly for conserving system energy and position accuracy and should be avoided when energy conservation is an important factor. The Runge-Kutta methods grossly violated the law of energy conservation, indicating the implementation may have been flawed.

B. Moving Forward

The simulator and results presented in this report represent the very beginning of a much larger concept. This section outlines the planned direction of the project in the near future.

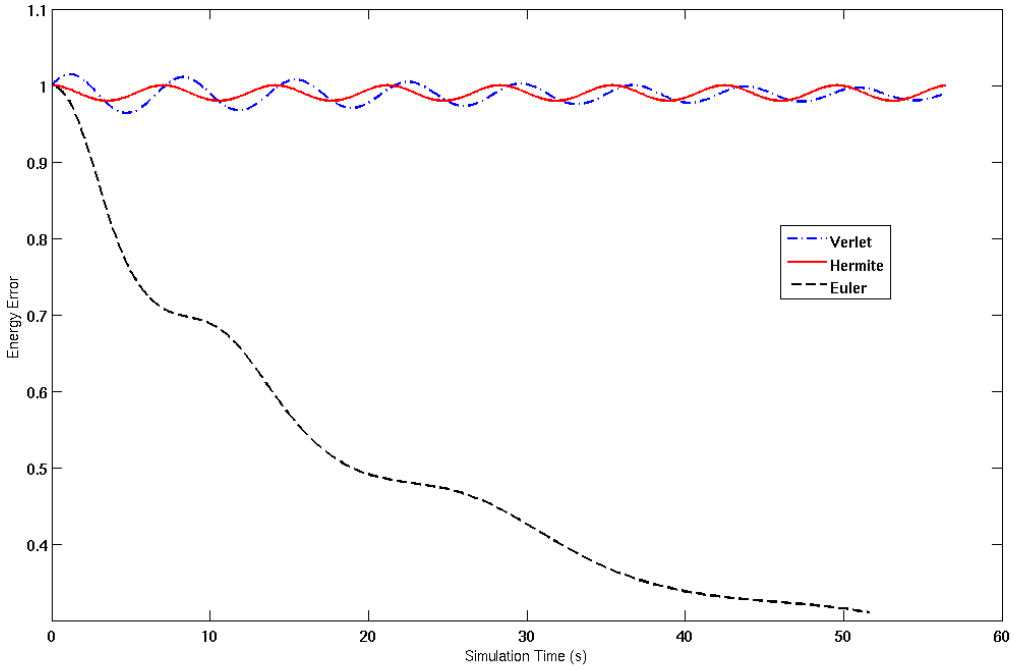

Figure 9. A comparison of energy error for various integration methods using a variable time step with $\eta = 0.05$.

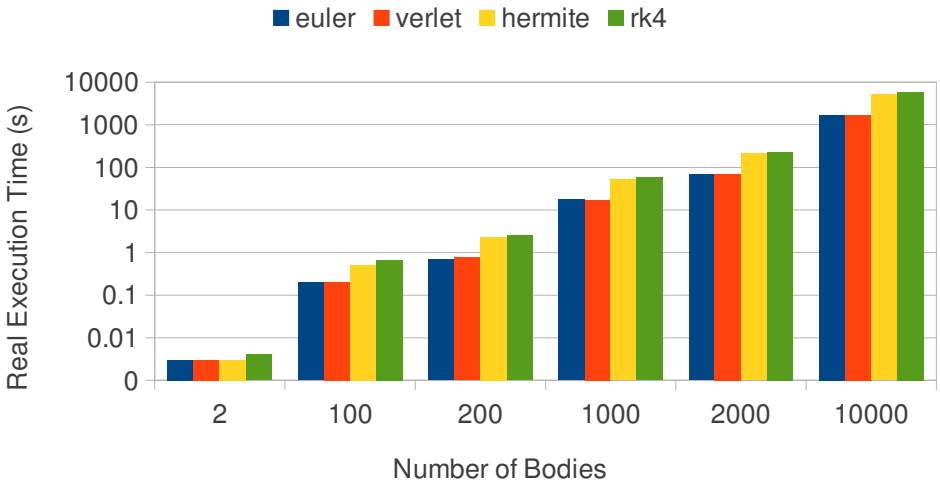

Figure 10. Run time comparison for 100 iterations using different integration methods and number of particles

1. Code Optimizations

For a simulation with 2000 bodies and 100 iterations, the Forward Euler integrator spent 94.18% of code executation time within the function `OneBodyAccel()`. An infinite speedup in the rest of the code would result in a little more than 6% of program execution, according to Amdahl's law. Thus, `OneBodyAccel()` should be optimized, and is a candidate for computation on a parallel architecture, such as the GPU.

There are tradeoffs between memory consumption and processor consumption, which NOMS will eventually be able to choose at run-time.

Table 6. At present, NOMS can reasonably simulate 10,000 particles for under 1 minute/iteration. Real time execution is plotted in Fig. 10.

Number of bodies	Number of iterations	integrator	Variable dt	Save part	Calc stat	Real time (s)	Time per iteration (s)	N body-body comparisons per second
2	100	euler	TRUE	FALSE	FALSE	0.003	0.00003	133000
100	100	euler	TRUE	FALSE	FALSE	0.199	0.00199	5030000
200	100	euler	TRUE	FALSE	FALSE	0.689	0.00689	5810000
1000	100	euler	TRUE	FALSE	FALSE	17.42	0.1742	5740000
2000	100	euler	TRUE	FALSE	FALSE	68.377	0.68377	5850000
10000	100	euler	TRUE	FALSE	FALSE	1704.541	17.04541	5870000
2	100	verlet	TRUE	FALSE	FALSE	0.003	0.00003	133000
100	100	verlet	TRUE	FALSE	FALSE	0.203	0.00203	4930000
200	100	verlet	TRUE	FALSE	FALSE	0.759	0.00759	5270000
1000	100	verlet	TRUE	FALSE	FALSE	17.117	0.17117	5840000
2000	100	verlet	TRUE	FALSE	FALSE	69.176	0.69176	5780000
10000	100	verlet	TRUE	FALSE	FALSE	1707.076	17.07076	5860000
2	100	hermite	TRUE	FALSE	FALSE	0.003	0.00003	26700
100	100	hermite	TRUE	FALSE	FALSE	0.51	0.0051	3920000
200	100	hermite	TRUE	FALSE	FALSE	2.323	0.02323	3440000
1000	100	hermite	TRUE	FALSE	FALSE	52.161	0.52161	3830000
2000	100	hermite	TRUE	FALSE	FALSE	211.246	2.11246	3790000
10000	100	hermite	TRUE	FALSE	FALSE	5215.728	52.15728	3830000
2	100	rk4	TRUE	FALSE	FALSE	0.004	0.00004	400000
100	100	rk4	TRUE	FALSE	FALSE	0.659	0.00659	6070000
200	100	rk4	TRUE	FALSE	FALSE	2.518	0.02518	6350000
1000	100	rk4	TRUE	FALSE	FALSE	58.011	0.58011	6900000
2000	100	rk4	TRUE	FALSE	FALSE	232.716	2.32716	6880000
10000	100	rk4	TRUE	FALSE	FALSE	5871.16	58.7116	6810000

By enabling compiler optimization (`-O3`), inline functions were folded into the code and not itemized on `gprof`'s output. Running NOMS without compiler optimizations revealed that there are several inline functions that are relatively slow and called frequently and are candidates for performance enhancements, namely `mag()`, which returns the magnitude of a vector.

Sample output from `gprof`, the tool used for profiling NOMS, is given in Appendix B.

2. Tree Methods

The Barnes-Hut simulation of an N-body system is a common optimization which uses an octree data structure to store particles. By using a tree data structure, a clumped group of particles far away from another body can be treated as a single body. A good implementation of the Barnes-Hut algorithm can reduce the number of pairwise comparisons from $O(N^2)$ to $O(NlogN)$. Performance improvements can be seen for simulations with $N > 100$ particles.[3] For smaller simulations, this method has worse performance than using a linear array and performing all N^2 comparisons because of the overhead of accessing particles within the tree. This method also has greater memory requirements.

3. Offloading computation to the Graphics Processing Unit

Currently, all computation is executed serially on the CPU. With the addition of OpenMP, we could take advantage of a multi-core CPU or CPU cluster. Because of the massively parallel nature of the N-body problem, running NOMS on the GPU is necessary for large-scale simulations. Memory management becomes a huge problem on the GPU as the general purpose GPU computing paradigm is still in its infancy. A huge performance cost occurs from transfers between CPU memory and GPU memory, as well executing serial code on the GPU.

4. User Interface and Natural Interaction

The future concept for the NOMS code is that it will be used by scientists and engineers that would like to rapidly explore physical phenomenon through simulation. Many programs have already been written to model a specific

phenomenon, but the learning curve to use these programs can make it very difficult for other researchers to utilize these tools. By creating an effective graphical user interface, researches can quickly get to what matters: running simulations. The other aspect that is important to a researcher is the ability to convey why a specific aspect of a simulation that was run is important and why it was run in the first place. This led to the concept of using natural interaction for the post processing environment of NOMS. Natural interactoin is the use of body motion and voice to interact with a program as opposed to a mouse and a keyboard. A group of researchers could all interact with simulation data by simply making gestures or speaking commands to explore the data set or point out a specific aspect of the simulation.

Appendix

A. Valgrind output

The program `Valgrind` is used to check for memory leaks after executing a program. This is particularly important for applications that will be running for long periods of time without exiting or will allocate memory multiple times (for multiple simulations). With the use of constructors and destructors to allocate and free memory, memory management is handled implicitly. The Valgrind output below shows that NOMS is free from memory leaks.

```
==8281== Memcheck, a memory error detector
==8281== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==8281== Using Valgrind-3.6.1-Debian and LibVEX; rerun with -h for copyright info
==8281== Command: ./noms
==8281==
Running NOMS
==8281==
==8281== HEAP SUMMARY:
==8281==     in use at exit: 0 bytes in 0 blocks
==8281==   total heap usage: 241,139 allocs, 241,139 frees, 59,120,490 bytes allocated
==8281==
==8281== All heap blocks were freed -- no leaks are possible
==8281==
==8281== For counts of detected and suppressed errors, rerun with: -v
==8281== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

B. GProf output

The profile output below was run using 2000 bodies for 100 iterations of the Forward Euler integrator. Statistics generation and particle file output were disabled to avoid distoring the results with non-critical calculations or high disk access times.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 94.18      2.26     2.26   200000     0.01     0.01  Simulator::OneBodyAccel(...)
  3.33      2.34     0.08        1    80.01    80.01  Processor::calcDynamicTimeStep(...) const
  2.08      2.39     0.05        1    50.01    50.01  Processor::calcPotentialEnergy(...) const
  0.42      2.40     0.01      100     0.10    22.70  Simulator::ForwardEuler(...)
  0.00      2.40     0.00   200000     0.00     0.00  Particle::update(...)
  0.00      2.40     0.00     2000     0.00     0.00  Factory::allocateParticle(...)
  0.00      2.40     0.00     2000     0.00     0.00  Particle::Particle(...)
  0.00      2.40     0.00     2000     0.00     0.00  Particle::~Particle()

                                                ...

granularity: each sample hit covers 2 byte(s) for 0.42% of 2.40 seconds

index % time    self  children    called     name
                0.01    2.26     100/100     Manager::advanceOneTimestep(...) [2]
[1]     94.6    0.01    2.26     100         Simulator::ForwardEuler(...) [1]
                2.26    0.00  200000/200000  Simulator::OneBodyAccel(...) [3]
-----------------------------------------------
```

```
                                                <spontaneous>
[2]     94.6    0.00    2.27                    Manager::advanceOneTimestep(...) [2]
                0.01    2.26     100/100        Simulator::ForwardEuler(...) [1]
                0.00    0.00       4/4          Factory::createFloat3Array(...) [27]
-----------------------------------------------
                2.26    0.00  200000/200000     Simulator::ForwardEuler(...) [1]
[3]     94.2    2.26    0.00  200000            Simulator::OneBodyAccel(...) [3]
-----------------------------------------------
                0.08    0.00       1/1          Manager::updateDynDt() [5]
[4]      3.3    0.08    0.00       1            Processor::calcDynamicTimeStep(...) const [4]
-----------------------------------------------
                                                <spontaneous>
[5]      3.3    0.00    0.08                    Manager::updateDynDt() [5]
                0.08    0.00       1/1          Processor::calcDynamicTimeStep(...) const [4]
-----------------------------------------------
                0.05    0.00       1/1          Manager::getStats(...) [7]
[6]      2.1    0.05    0.00       1            Processor::calcPotentialEnergy(...) const [6]
-----------------------------------------------
                                                <spontaneous>
[7]      2.1    0.00    0.05                    Manager::getStats(...) [7]
                0.05    0.00       1/1          Processor::calcPotentialEnergy(...) const [6]
                0.00    0.00       5/5          std::vector<double, std::allocator<double> >::... [26]
                0.00    0.00       1/1          Processor::calcKineticEnergy(...) const [36]
                0.00    0.00       1/1          Processor::calcAngularMomentum(...) const [37]
                0.00    0.00       1/1          Processor::calcCG(...) [35]
```

...

The same simulation was run with adaptive time steps turned on for every iteration. Execution time increased by a factor of four. The program spent 76.31% of execution time calculating an appropriate time step for the next iteration.

```
Each sample counts as 0.01 seconds.
%       cumulative self    self    total
time    seconds seconds calls   ms/call ms/call name
76.31   6.6     6.6     101     65.36   65.36   Processor::calcDynamicTimeStep(...)
22.78   8.57    1.97    200000  0.01    0.01    Simulator::OneBodyAccel(...)
0.58    8.62    0.05    1       50.01   50.01   Processor::calcPotentialEnergy(...)
0.23    8.64    0.02    100     0.2     19.9    Simulator::ForwardEuler(...)
0.12    8.65    0.01                            Simulator::update(...)
0.00    8.65    0.00    200000  0.00    0.00    Particle::update(...)
```

C. Concluding Remarks

The seemingly simple N-body problem is highly applicable to many fields of study, including mathematics, astrophysics, chemistry, and aerospace engineering. With a better understanding of the tools that are available to researchers, humankind can generate more accurate models of interplanetary trajectories by modeling 3rd body perturbations, planet oblateness, and solar pressure. Each effect exhibits a force on a body, which can be integrated to estimate the new location of each body. For these types of simulations, significantly fewer bodies are simulated, with a priority placed on position accuracy. In the opposite case, globular cluster evolution emphasizes simulating upwards of 10^6 bodies, with a priority placed on energy conservation. With today's technology, computers have the computational capacity to simulate very large environments, but because all models are imperfect representations of the physical world, it will never be possible to perfectly model the universe. Even still, there is so much yet to be learned about our universe through N-body simulation.

Acknowledgments

We would like to thank Dr. David Marshall for giving us the resources and the encouragement to pursue this massive project and Kurt Papathakis for all of his help and enthusiasm. We would also like to thank Xanadu for countless hours of compute time without putting up too much fuss.

References

[1]LeVeque, R. J., *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*, Philidelphia, 2007.

[2]Hut, P. and Makino, J., "The N-Body Problem: From Leapfrog to Runge-Kutta," *The Kali Code*, Vol. 4, 2007.

[3]Binney, J. and Tremaine, S., *Galactic Dynamics*, Princeton University Press, 2nd ed., 2008.

[4]Simon F. Portegies Zwart, R. G. B. and Geldof, P. M., "High Performance Direct Gravitational N-Body Simulations on Graphics Processing Units," 2007.

[5]Nadarajah, S. K. and Jameson, A., "Studies of the Continuous and Discrete Adjoint Approaches to Viscous Automatic Aerodnyamic Shape Optimization," Vol. 25-30, 2001.