***Gravitational N-Body Simulations***

Ryan Bleile & Ankit Adlakha

A gravitational n-body simulation is a numerical solution for the equations of motion for n particles which are interacting under the influence of gravitational forces. These simulations are used as tools in astrophysics simulations ranging from small solar system scales up to galaxy sized scales. Similar approaches can be applied to n-body systems reacting under different forces and can be used to describe molecules or electrons as well. The techniques involved with n-body simulations as well as numerical integration are still developing newer, better, faster ways to perform under a large number of bodies. In this project we will be exploring three classical numerical methods used to solve the n-body problem and will be comparing their different efficiencies by looking at the introduced error in comparison with the computational cost of performing each update. We will also be using the simulation to explore gravitational systems looking for stability versus chaos, collision handling, and any interesting factors we notice along the way.

**Project Overview**

*a. Brief description of system*

The system is a simulation which describes the motion of objects such as planets or stars according to Newton's Law of Gravity.Our simulation handles systems of arbitrary size however computation time needed for each step increases dramatically with the addition of additional bodies.

*b. Statement of goals and/or hypothesis*

For this Simulation our Goals were to weigh the induced errors in calculating the integration of Newton’s Law using different techniques, against the number of operations required for each technique.

Our goal is to understand the differences in errors to computational cost and try to balance both for a given required amount of acceptable error. That is, given a particular error range we should be able to recommend which method and step size gives optimal runtime.

Our computational methods which will be studied will be:

1. Euler
2. Runge-Kutta O(2)
3. Runge-Kutta O(4)

An additional goal is to study the effect of implementing a reduced algorithm for computing the total force on an object as opposed to the standard brute force method. By reducing the number of computations by a factor of N we may be able to see some improvements when we compare to the computations per time step to the errors induced in the time step.

Another objective we could spend time focusing on is to see how different treatments for collisions affect the outcome of a simulation run, regarding the chaos/stability, or ejections of bodies from the system. We have chosen the combination of bodies when collisions occur for now. Future work could be to study this factor changing.

**System Model**

a. Mathematical or graphical model of the system

Mathematically our system is entirely modeled from Newton’s Law of Gravity. Newton’s law is defined as:

Newtons law of Gravity:

Where m and M are the masses of two bodies between which the force is felt, r is the radius between the two objects, and G is the Newtonian Gravitational Constant. Integration of the acceleration vector with respect to time yields the velocity of the object at a given time. In the same way, integration of the velocity will give the position, as given by the equations:

We will apply all three integration techniques to the velocity calculation integrating acceleration, but will only integrate velocity using the Euler method to calculate position.

Our equation to describe collisions between objects will come from the equation for an inelastic collision between two bodies:

Collisions: (Conservation of Momentum)

This statement as noted is also a statement of the conservation of momentum which is upheld under these types of collisions.

*b. Discussion of accuracy of model and explanation of inaccuracies or omissions*

The model exactly represents a universe with only Newtons law of Gravity affecting all bodies. In other words there's nothing else that is affecting the motion of a mass except the for the other masses in the system.

Also, the perfect inelastic collision treatment for all collisions is simply a way to model collisions but does not account for all possible ways bodies could collide or how a standard planetary system would behave. Potential future work would be to include differing densities between bodies and based on this factor decide if collisions should be inelastic, elastic or possible even breaking into more bodies. Providing conditions and handling many different types of collisions would prove to be complex and computationally expensive so we have left our system simple so we can better understand the integration techniques and not the astronomical physics that it would represent.

**Simulation**

*a. Description of implementation*

We implemented our model our using vpython; this package makes it very easy to visualize using a simple library of tools. We implemented integration of Newton’s Law of Gravity using three methods: Euler, Runge-Kutta 2 and Runge-Kutta 4.

The program opens three windows for controlling the simulation and a fourth window for choosing which configuration to run. The number of bodies and their starting conditions is defined in a config file, further implementation of the config file is explained in Appendix A: How to’s.

The Reduced method which performs a decreasing number of calculations each iteration through the loop has a net effect of reducing the problem by ½ n^2 operations. This is achieved through computing information for multiple bodies simultaneously, not repeating calculations of the effects between two bodies. Comparing the for loops of the brute force and reduced methods we can see the difference:

for i = 0; i < number bodies; i++

for j = 0; j < number bodies; j++

versus

for i = 0; i < number bodies; i++

for j = i + 1; j < number bodies; j++

The net effect of adjusting the second for loop by i+1 each iteration will produce a net effect of half the total number of computations.

*b. Discussion of accuracy of the implementation (relative to model) and explanation of inaccuracies or omissions*

Inaccuracy in implementation came from the time stepped integration. Euler with the most error at a given dt and RK2 and RK4 following. At dt approaches zero each of these techniques will have error approaching zero. The euler method will have an error which is on the order of , the time step, while RK2 is and RK4 is . The order of these errors comes from deriving the methods used in the integration were the order of errors expressed is the first ignored term.

Another inaccuracy is the current treatment of Newton’s gravitational constant as having a value of 1. This could be explained by scaling the system to that where this holds true. The behavior of the system is still the same as having one over radius squared behavior but the actual values for masses and radii will need to be called in order to represent an actual planetary system.

8) **Results**

*a. Description of experiments*

Running one given configuration using multiple methods and comparing the results is gong to be the basis for our experimentation.

What we can easily compute using this code is the order of the errors at a given step in the calculation as well as the total number of floating point operations that the update functions have done so far.

What we can also do to check error in the different methods is to run a given configuration and method at multiple time steps and compare the differences we see in between the two. The one with the smallest dt will be closest to the exact solution.

*b. Summary of results*

When we compare the different methods we see the differences between computational cost and the order of errors. From this we can take a given problem and decide if we want errors down to a particular amount and we know approximatly how many updates the problem will need we will be able to figuere out what order of dt we will need for each method and from this we can see how many floating point operations will be needed to complete the simulation. After analyzing multiple runs at different dt steps we can get a sense for what will be appropriate to solve a given problem.

Analyzing the error also opens the door to start writing adaptive code which will keep error below a particular amount by adjusting the dt with each update as needed. By computing an update with dt and dt/2 simultaneously we can compare the results and see if the difference results in a dramatically different value. If so we can use the smaller dt and check dt/4 next. This would be a way to minimize our errors by implementing an understandng of errors.

Some example output from our error analysis:

with a 3 body simulation running 1000 update steps at dt = .1: our computation time and error are produced for each:

Euler:

Computations: 15000 Error: 100

Euler Reduced:

Computations: 10500 Error: 100

RK2:

Computations: 33000 Error:10

RK2 Reduced:

Computations: 21000 Error: 10

RK4:

Computations: 72000 Error: .1

RK4 Reduced:

Computations: 54000 Error: .1

*c. Analysis of results relative to goals and/or hypothesis*

Looking at the results from this one experiment we can show the true difference between the Euler and runge-kutta methods. For example, in order to produce an euler scheme whic hhas the same error as the runge kutta 4 at the same time step we would have to to run euler for much longer at a much smaller time step.

In order to make Euler produce the same results as runge kutta 4 we would have to run at a dt that is 1000 times smaller, or .00001. Running at such a small dt will incur as much error as running runge-kutta 4 over the same range.

As we make dt smaller our simulation will become more accurate but it will also incur a much higher running cost. If we keep in mind that the number oof computations required is on the order of n squared than we do not want to do that update thousands of times more than we would otherwise have to.

With respect to the initial goals of the simulation we have shown that we can produce results for our integration in six different ways using three methods. While Euler is the simplest and fastest method it brings an incredible amount of error with it. We can reduce the error in 1000 calculations by multiple orders of magnitude by only increasing computational cost by 4 times the largest factor simply switching from Euler to Runge-Kutta 4.

**9) Reflection**

*a. Brief discussion of what you learned from the project*

As a team in order to complete this project we had to learn each of the three methods used for integration. We also had to learn about error analysis and understanding computational costs for different algorithms. In addition to features of the project we also, had to learn vpython. Neither of us has had any experience with vpython previous to this course and so learning how to use the visual features provided were very important and useful.

**Appendix:**

***How to’s***

*Writing a configuration file:*

the first line is a global attribute describing to retain 10,000 updates in the trial or the entire trail behind a body. True means retain 10,000 is true. False means retaining entire trail.

configuration files contain all basic information needed to initialize a system. Each line describes a body and contains the following information:

pos=vector(x,y,z) “A position vector describing its location in 3-D space”

vel=vector(x,y,z) “A velocity vector describing its initial instantaneous velocity in 3-D space”

mass=#number “A number representing the mass of the object”

make\_trail= True/False “A boolean representing if a trail should be made behind each object”

color = (r,g,b) “A tuple in rgb format where each value ranges from 0 to 1 as a percent of color”

example config file with two bodies:

{'retain' : False}

{'pos': vector(0, -100, 0), 'vel': vector(0, 1, 0), 'mass': 10, 'make\_trail': True, 'color': (1,1,1)}

{'pos': vector(0, 100, 0), 'vel': vector(0, -1, 0), 'mass': 10, 'make\_trail': True, 'color': (1,0,0)}

Config files should be placed in the same directory as the python main file in order to be easily accessed by the programs get\_file funtion

There should be no extra space after or between or before lines in a config file. There are no checks built in yet to handle adding arbitrary whitespace

*Running the program in visualization mode:*

There is a line (#304) which has a way to make the window which the bodies are rendered in -visible or not. If running batch tests you can make the window not visible by uncommenting this line. If you wish to see the animation you need this line commented out.

*Running the program to a stopping point:*

In order to run this program to a stopping condition you may uncomment lines #372-376. If you wish to redefine the end condition to a different value you will need to change the number in the if statement at line 374

*Running the visualization in 3D:*

If you wish to turn on 3d visualization you may turn on the stereo options. Uncomment line 309 to make 3D visualizations.

*Changing dt*

Currently there is no built in way to change dt without editing its value. It is located at line #271

*Writing to a file is a work in progress:*

if you wish to write the index , error, and computational cost out to a file you may uncomment those lines of code but there is no point to them at the moment. In reality you would only ever want to see the value at the end, at least for now.

**Known Bugs** (that affect usability)

1. If you toggle the reduced algorithm and then change the algorithm you will not be able to run the reduced versions.

current work around: always choose algorithm with button first and then toggle reduced or not

**Future Work:**

Currently in thoughts for future updates to this code:

1. An error correcting algorithm to run below a certain error range

2. Implementing a serial version of a clustering update method

3. batch run versions of the code separate from visualized versions

4. Stronger error analysis between dt simultaneously

5. dt slider to control dt step size in non self correcting algorithms

6. set up controls only - no changing during run