Using numerical methods to solve the Gravitational *n*-Body Problem & represent the result graphically using OpenGL

Brian Tyrrell [1]

[1] Contributed by Robert Flood & Dr. Mike Peardon

Contents

1. Abstract

My objective in researching this topic was to produce a C++ code which gives an accurate approximation to the solution to Newton's Laws of Motion for *n* bodies. In order to do this, I wrote three sets of code, one which uses a 3 step Leapfrog Integrator, another which uses a 7 step Leapfrog Integrator and the third which uses the Barnes-Hut Algorithm. To check that the codes were functioning correctly I preformed numerous tests[2], which all three of them passed. My result is a useful, accurate simulation tool, which has many applications[3] and also room for further enhancements[4]. To conclude, after 7 weeks of research I have assembled (all original unless otherwise stated) and animated using OpenGL[5], 3 algorithms which, with varying degrees of accuracy[6], provide a solution to the *n*-Body Problem.

2. Introduction

The n-Body Problem is a classical problem which poses the question; given all the relevant details (mass, initial velocity, initial position, etc) of *n* bodies at some time *t*, what are the positions of the bodies at time $t + dt$ (for some dt), where the bodies are acted on by some external force *F*. In this case, the system will obey Newton's Law of Universal Gravitation, that is;

$$F_{1,2} = G\frac{m_1 m_2}{{r_{1,2}}^2}$$

Since solving Newton's equations of motion analytically for $n >> 1$ is practically impossible, it has become standard to use numerical approximations to the actual solutions, to the highest degree of accuracy possible. I have used 3 and 7 step Leapfrog Integrators to approximate solutions to the ODE's, and later I have created a Barnes-Hut Algorithm to deal with larger numbers of bodies (in excess of 1,000). The solution to this problem has numerous practical applications; accurately plotting the course of satellites around the Earth, plotting trajectories of planets and modelling the behaviour of flying insects to name a few. The Barnes-Hut Algorithm is specifically designed to model globular clusters for extended periods of time and can provide accurate, verifiable data.

[2] See Section 4
[3] See Section 7
[4] See Section 6
[5] See Section 5
[6] See Section 4.5

3. Leapfrog Integration and the Barnes - Hut Algorithm

(A full, exact copy of the code is presented at the end[7]).

3.1 The 3 step Leapfrog Integrator

The 3 step Leapfrog Integrator is named after the 3 'steps' or 'leaps' it takes:

I. The position at time t -- $x(t)$ -- is updated:

$$x(t + dt/2) = x(t) + v(t) * h/2$$

....where $v(t)$ is the velocity at time t and h is a 'timestep'.

II. The acceleration is then calculated.
The velocity is updated:

$$v(t + dt) = v(t) + a(t + dt/2) * h$$

III. Finally the position is updated again, with the new velocity:

$$x(t + dt) = x(t + dt/2) + v(t + dt) * h/2$$

In C++, I wrote a routine to perform these actions:

```
void update(double a[][3], double b[][3], double h, int x, int q)
{
    for(int i = 0; i<x; ++i)
    {
        for(int j = 0; j<3; ++j)
        {

             if( q == 0 )
            {   a[i][j] = a[i][j] + b[i][j]*h/2;
            }
            else
            {   a[i][j] = a[i][j] + b[i][j]*h;
            }
        }
    }
}
```

Where a is the position array (with $a[\,i\,][\,j\,]$ being the position of body i in the ($j = 0, 1, 2$) corresponding to 1st, 2nd, 3rd dimension), b is the velocity array, h is the timestep, x is the number of bodies and q is an indicator -- if $q = 0$, then the position is updated, and if q is nonzero, the velocity is updated.

[7] See Section 10

The routine is called in the main program:

```
while(time < 2000000000)
{
    update(xb, vb, h, n, 0);
    cc_acc(xb, ab, m, n);
    update(vb, ab, h, n, 1);
    update(xb, vb, h, n, 0);

    .....

    time += h;
}
```

Where *time* is obvious and *cc_acc* is another routine, which calculates acceleration:

```
void cc_acc(double a[][3], double b[][3], double m[], int n)
{
    for(int i = 0; i<n; ++i)
    {
        for(int j = 0; j<3; ++j)
        {   b[i][j] = 0;
        }
    }

    for(int i = 0; i<n; ++i)
    {
        for(int j = 0; j<n; ++j)
        {
            if(j != i)
            {
                for(int k = 0; k<3; ++k)
                {   b[i][k] -= (G*m[j]*(a[i][k] -
a[j][k]))/(dist(a,i,j)*dist(a,i,j)*dist(a,i,j));
                }
            }
        }
    }
}
```

Where *a* is the position array, *b* is the <u>acceleration</u> array, *m* is the mass array, *n* is the number of bodies, *G* is Newton's Gravitational constant and *dist(a, i, j)* returns the Euclidean distance between bodies *i* and *j*.
In this routine, the acceleration array is initialised at the start of each call of the routine, and the routine calculates the acceleration between bodies *i* and *0, 1, 2.......n (!= i)* in the *x, y and z* direction *(k = 0, 1, 2).*

3.2 The 7 step Leapfrog Integrator

The 7 step Leapfrog Integrator follows the same logic, but with 7 'steps' or 'leaps' instead of 3.

I.	The position is updated with step size	*h/2f*
II.	The velocity is updated with step size	*h/f*
III.	The position is updated with step size	*(1 - w)*h/2f*
IV.	The velocity is updated with step size	*-h*w/f*

V. The position is updated with step size *(1 - w)*h/2f*
VI. The velocity is updated with step size *h/f*
VII. The position is updated with step size *h/2f*

Where *w = the cube root of* 2 and *f = 2 - w.*

The C++ routine is:

```
void update(double a[][3], double b[][3], double h, int x, int q, int r)
{
    for(int i = 0; i<x; ++i)
    {
        for(int j = 0; j<3; ++j)
        {
            if( q == 0 )
            {
                if( r == 0 )
                {   a[i][j] = a[i][j] + b[i][j]*(h/(2*f));
                }
                else
                {   a[i][j] = a[i][j] + b[i][j]*(1 - w)*(h/(2*f));
                }
            }
            else
            {
                if( r == 0 )
                {   a[i][j] = a[i][j] + b[i][j]*h/f;
                }
                else
                {   a[i][j] = a[i][j] - b[i][j]*h*(w/f);
                }
            }
        }
    }
}
```

With *q* and *r* being similar indicators to the *q* of the previous integrator. The routine is called in the main program:

```
while(time < 2000000000)
{
    update(xb, vb, h, n, 0, 0);
    cc_acc(xb, ab, m, n);
    update(vb, ab, h, n, 1, 0);
    update(xb, vb, h, n, 0, 1);
    cc_acc(xb, ab, m, n);
    update(vb, ab, h, n, 1, 1);
    update(xb, vb, h, n, 0, 1);
    cc_acc(xb, ab, m, n);
    update(vb, ab, h, n, 1, 0);
    update(xb, vb, h, n, 0, 0);

    time += h;
}
```

With *cc_acc* being the same routine used in the 3 step integrator.

3.3 The Barnes - Hut Algorithm

For the Barnes-Hut Algorithm, the 7 step integrator is used to update the positions and velocities.
The Barnes Hut Algorithm is used for larger numbers of bodies for its ability to decide whether to calculate the force from a single body, or a larger group of bodies together. In order to do this, I wrote a class Node;

```
typedef class Node
{
    public:
        set < tuple < double, double, double > > element;       //Elements of node
        double cen_x, cen_y;                                    //Centre of node
        double com_x, com_y;                                    //COM of node
        double d;                                               //Dimension of node

        void einspn(double, double, double);                    //Modifies elements
        void calcen(double, double);                            //Modifies centre
        void calcom();                                          //Modifies COM
        void caldim(double);                                   //Modifies dimension

        int num();                                     //Outputs number of elements
        double dim();                                           //Outputs dimension
        double width(int);                                       //Outputs width
        double height(int);                                      //Outputs height
        double centre_x();                             //Outputs x coord of centre
        double centre_y();                             //Outputs y coord of centre
        double cencom_x();                                //Outputs x coord of COM
        double cencom_y();                                //Outputs y coord of COM
        double mass();                                      //Outputs mass of node
        double search(double, double, double);
        void clearnode();                                        //Resets node

} Node;
```

Where *search* outputs whether an element is in the node or not.
Another a routine *Treebuild* builds a 'quadtree' for the bodies - this routine partitions the bodies into (metaphorical) nodes;

```
void treebuild(Node ns[], double a[][2], double m[], int n, int p)
{
    double temp;
    int aa, bb, cc, dd;
    addelements(ns,a,m,n,p);
    if(ns[p].num() > 1)
    {
        temp = ns[p].dim();
        p += 4;

        aa = checkpos(p+1);
        ns[aa].calcen((ns[p-4].centre_x() - temp/4), (ns[p-4].centre_y() +
temp/4));
        ns[aa].caldim(temp/2);
        treebuild(ns,a,m,n,aa);

        bb = checkpos(p+2);
        ns[bb].calcen((ns[p-4].centre_x() + temp/4), (ns[p-4].centre_y() +
temp/4));
        ns[bb].caldim(temp/2);
        treebuild(ns,a,m,n,bb);
        cc = checkpos(p+3);
        ns[cc].calcen((ns[p-4].centre_x() - temp/4), (ns[p-4].centre_y() -
temp/4));
        ns[cc].caldim(temp/2);
        treebuild(ns,a,m,n,cc);

        dd = checkpos(p+4);
        ns[dd].calcen((ns[p-4].centre_x() + temp/4), (ns[p-4].centre_y() -
temp/4));
        ns[dd].caldim(temp/2);
        treebuild(ns,a,m,n,dd);
    }
}
```

Where *ns* is an array of type *Node*, *a* is the position array, *n* is the number of bodies and *p* is the node's number - identification for a particular node. The routine recursively computes all the children of the node, and calls another routine *addelements*, which adds elements to node *p*, which in turn calls a routine *online* which checks if a particular body intersects the boundary of a node. *checkpos* is a function which checks if the number of a node has been taken, and if so returns an available number. The program is written out in full at the end of the paper.
Once the quadtree has been built, the routine *force* is called, which determines and then calculates accelerations for bodies:

```
void force(Node ns[], double a[][2], double c[][2], double m[], int n)
{
    vector <int> listi;
    vector <int> listj;
    vector <int> listself;

    for(int l = 0; l<n; ++l)
    {
        for(int m = 0; m<2; ++m)
        {   c[l][m] = 0;
        }
    }

    for(int u = 0; u<50; ++u)
    {   ns[u].calcom();
    }

    listi.clear();
    listj.clear();
    listself.clear();

    for(int i = 0; i<n; ++i)
    {
        listself = nodefind(ns, a, m, i);
        for(int j = 0; j<n; ++j)
        {
            if( j != i)
            {
                listj = nodefind(ns, a, m, j);
                for(int k = 0; k<listj.size(); ++k)
                {
                    if(theta(ns,a,i,listj[k]))
                    {
                        if(find(listself.begin(), listself.end(), listj[k]) ==
listself.end())
                        {
                            if(find(listi.begin(), listi.end(), listj[k]) !=
listi.end())
                            {   k += 1e50;
                            }
                            else
                            {   cc_acc(ns, a, c, m, i, listj[k]);
                                listi.push_back(listj[k]);
                                k += 1e50;
                            }
                        }
                    }
                }
            }
            listj.clear();
        }
        listi.clear();
        listself.clear();
    }
}
```

Where *c* is the acceleration array (initialized to 0 on every call of *force*) and *theta* is a function which determines which nodes to calculate the acceleration for. The vectors are:

I. *nodefind(ns, a, m, i)* returns a vector which contains the nodes - shallow to deep - that body *i* is in.
II. *listself* for body *i* is a vector which contains the nodes - shallow to deep - that body *i* is in
III. *listj* is the same as II
IV. *listi* is a vector which keeps track of which nodes body *i* has calculated the acceleration with. If a node *p* is an element of *listi*, the program moves on to the next body *j*, such that the acceleration between body *i* and node *p* (or any of its children) is not calculated.

theta has arguments *i* -- for some body *i* -- and *p* -- for some node *p:*

```
bool theta(Node ns[], double a[][2], int i, int p)
{
    double thet, distance;
    distance = dist(a[i][0], a[i][1], ns[p].cencom_x(), ns[p].cencom_y());
    thet = ns[p].dim()/distance;
    if(ns[p].num() == 1)
    {   return true;
    }

    else
    {
        if(thet < 0.5)
        {   return true;
        }
        else
        {   return false;
        }
    }
}
```

theta will return true if the node has one body in it, or if *thet* < 0.5 - a standard for this type of calculation - and if *theta* returns false, the function is called again, this time with one of the children of node *p*.

Once a node has passed the criteria for calculation, *cc_acc* is called to calculate the acceleration between body *i* and Node *p:*

```
void cc_acc(Node ns[], double a[][2], double c[][2], double m[], int i, int p)
{
    c[i][0] -= (G*ns[p].mass()*(a[i][0] -
ns[p].cencom_x()))/(dist(a[i][0],a[i][1],ns[p].cencom_x(),ns[p].cencom_y())*dist(a[
i][0],a[i][1],ns[p].cencom_x(),ns[p].cencom_y())*dist(a[i][0],a[i][1],ns[p].cencom_
x(),ns[p].cencom_y()));

    c[i][1] -= (G*ns[p].mass()*(a[i][1] -
ns[p].cencom_y()))/(dist(a[i][0],a[i][1],ns[p].cencom_x(),ns[p].cencom_y())*dist(a[
i][0],a[i][1],ns[p].cencom_x(),ns[p].cencom_y())*dist(a[i][0],a[i][1],ns[p].cencom_
x(),ns[p].cencom_y()));
}
```

Where *dist(a, b, c, d)* computes the Euclidean distance between body *i* -- with coordinates *(a, b)*, and the centre of mass of node *p* -- with coordinates *(c, d).*
Once the total acceleration on body *i* has been calculated, the program returns to the original routine *leapfrog* where the code was initially called from;

```
void leapfrog(Node ns[], double a[][2], double b[][2], double c[][2], double m[],
double h, int n)
{
    reset(ns);
    ns[0].calcen(0,0);
    ns[0].caldim(2*maxm(a, n));
    treebuild(ns, a, m, n, 0);

    update(a, b, h, n, 0, 0);
    reset(ns);
    ns[0].calcen(0,0);
    ns[0].caldim(2*maxm(a, n));
    treebuild(ns, a, m, n, 0);
    force(ns, a, c, m, n);
    update(b, c, h, n, 1, 0);

    update(a, b, h, n, 0, 1);
    reset(ns);
    ns[0].calcen(0,0);
    ns[0].caldim(2*maxm(a, n));
    treebuild(ns, a, m, n, 0);
    force(ns, a, c, m, n);
    update(b, c, h, n, 1, 1);

    update(a, b, h, n, 0, 1);
    reset(ns);
    ns[0].calcen(0,0);
    ns[0].caldim(2*maxm(a, n));
    treebuild(ns, a, m, n, 0);
    force(ns, a, c, m, n);
    update(b, c, h, n, 1, 0);
    update(a, b, h, n, 0, 0);
}
```

Where *reset* is a routine which empties the elements of all nodes and clears set *s* - the set of 'ints' which keeps track of which node numbers have been taken, and *maxm* is a function which returns the largest *x or y* coordinate of the system, to create the dimensions of the first node (the first node has $p = 0$).

In the main program:

```
int main()
{
    .....
    Node no[50];
    .....

    while(time < 2000000000)
    {
        leapfrog(no, xb, vb, ab, m, h, n);

         .....

        time += h;
    }

    return 0;
}
```

Which is all self explanatory.
In each of the three programs - the 3 step integrator, the 7 step integrator and the Barnes-Hut algorithm - the main routine reads in data from a text file[8].

4. Testing the codes

There are numerous tests the codes must past in order to be deemed an acceptable approximation to the actual solution;

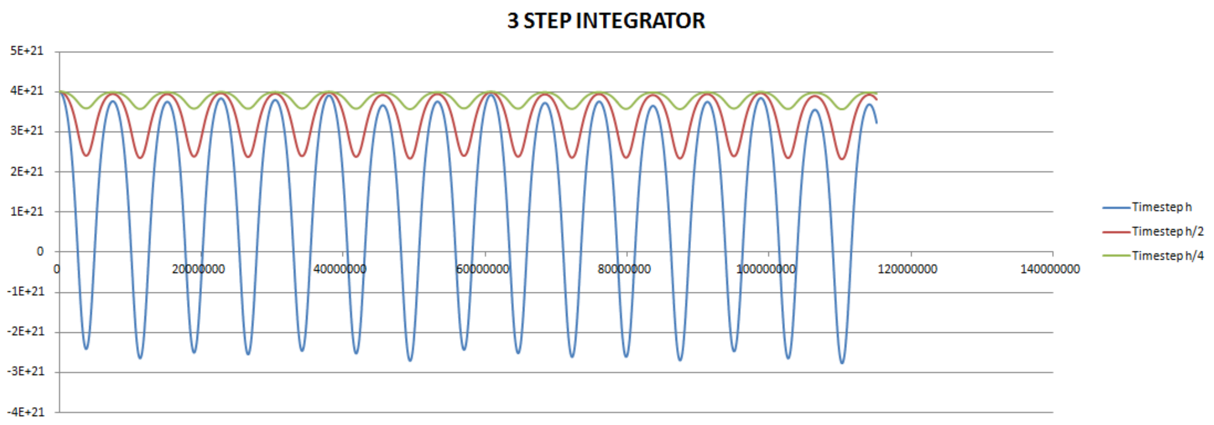
4.1 The Hamiltonian

In C++, the code for calculating the Hamiltonian (for both the 3 and 7 step integrators) in 3 dimensions is:

```
void Hamiltonian(double a[][3], double b[][3], double m[], double h, int x)
{
    double E = 0, sum_p = 0, sum_k = 0;
    for(int j = 0; j<x; ++j)
    {
        for(int i = 0; i<x; ++i)
        {
            if(i < j)
            {   sum_p -= (G*m[i]*m[j])/dist(a,i,j);
            }
        }
        sum_k += 0.5*m[j]*((b[j][0]*b[j][0]) + (b[j][1]*b[j][1]) +
(b[j][2]*b[j][2]));
    }
    E = sum_k + sum_p + 1.97655688974617e29 + 4e21;
    output2<<fixed<<setprecision(25)<<h<<"  "<<E<<endl;
}
```

[8] See Section 10.4

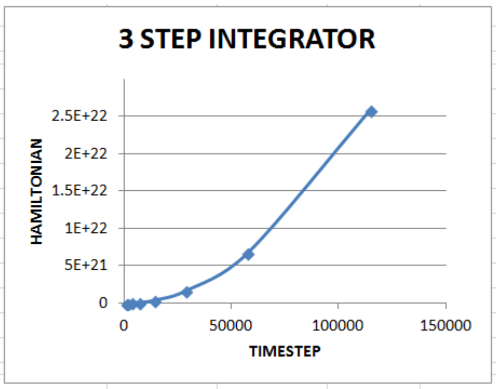
For a Leapfrog integrator, the Hamiltonian of the system is not conserved exactly, thus when computing the Hamiltonian for the system, a cosine wave which fluctuates about the actual value for the Hamiltonian is generated. However, the Hamiltonian $H \sim O(h^2)$, for a 3 step integrator, with h being the timestep, so halving the timestep results in quartering the fluctuation of the Hamiltonian, as seen in the diagram below:

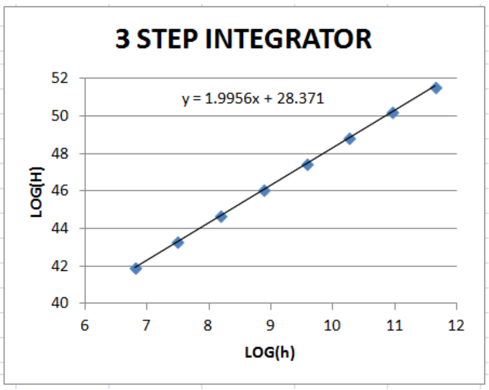


I chose a time $T = 14{,}400{,}000$ to take my measurements at, and computed $E_T(h)$, which is the difference in the Hamiltonian at time T with timestep h and $h/2$. Plotting this difference along with h shows a quadratic relationship between the two, which is seen clearer by taking the log of h and $E_T(h)$ (which gives a line of slope $1.9956 \sim 2$) proving the Hamiltonian $H \sim O(h^2)$.

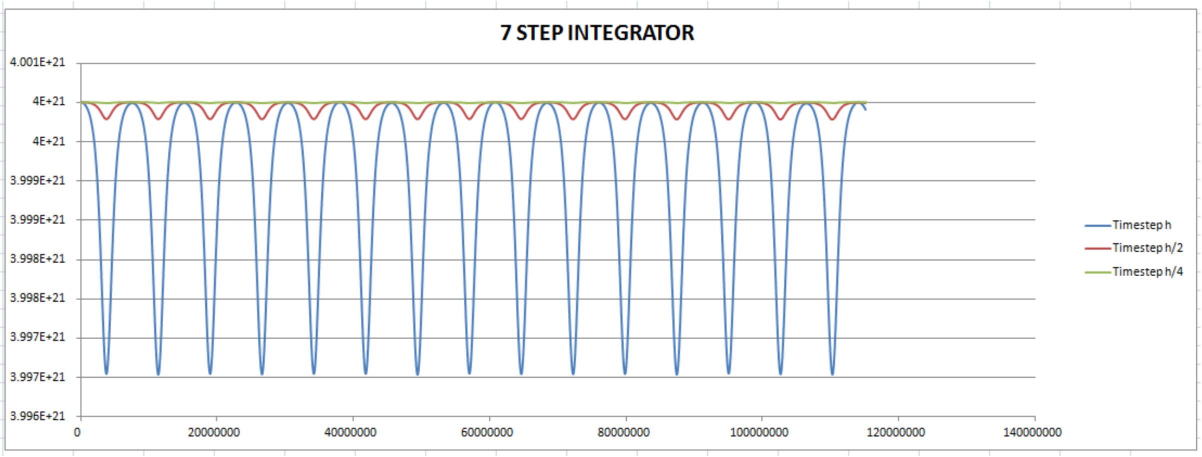
Time	E_t(h)
900	1.64E+18
1800	6.547E+18
3600	2.619E+19
7200	1.047E+20
14400	4.189E+20
28800	1.675E+21
57600	6.693E+21
115200	2.592E+22

3 STEP INTEGRATOR





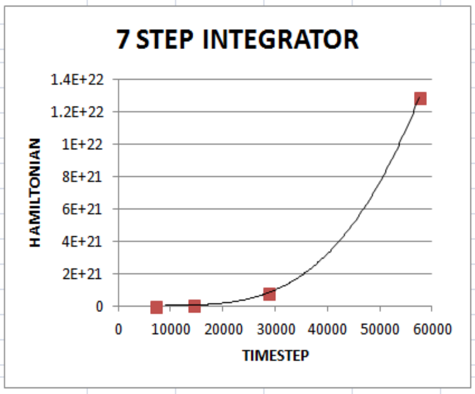
For the 7 step integrator, the Hamiltonian $H \sim O(h^4)$, so halving the timestep results in *1/16* of the Hamiltonian, as seen below:

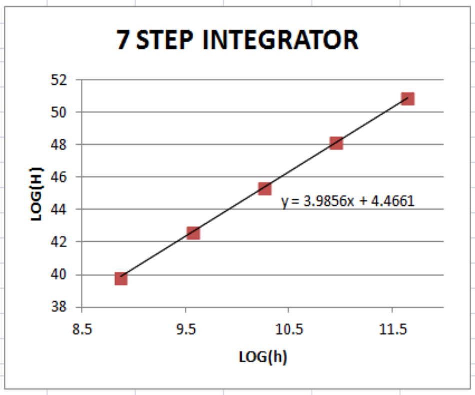


Taking the log (base 10) of both sets of data, we get a line of slope *3.9856 ~ 4*, proving this code correctly uses the 7 step integrator.

Time	E_t(h)
7200	2.0618E+17
14400	3.2403E+18
28800	5.1773E+19
57600	8.2385E+20
115200	1.2899E+22

7 STEP INTEGRATOR





Note: the smaller timesteps resulted in rounding errors which skewed the rest of the data, so I removed them.

4.2 Angular Momentum

The Leapfrog Integrator conserves angular momentum exactly, so

I. Plotting *Angular Momentum vs Time* should be a straight line
II. There should be no difference between the 3 and 7 step integrators

The C++ code for calculating angular momentum for both the 3 and 7 step integrators is:

```
double ang_mo(double a[][3], double b[][3], double m[], int n)
{
    double count = 0;
    for(int i = 0; i<n; ++i)
    {
        count += sqrt((norm(a,i)*norm(a,i))*(norm(b,i)*norm(b,i)*m[i]*m[i]) -
((a[i][0]*m[i]*b[i][0] + a[i][1]*m[i]*b[i][1] +
a[i][2]*m[i]*b[i][2])*(a[i][0]*m[i]*b[i][0] + a[i][1]*m[i]*b[i][1] +
a[i][2]*m[i]*b[i][2])));
    }
    return count;
}
```

This formula for angular momentum comes from:

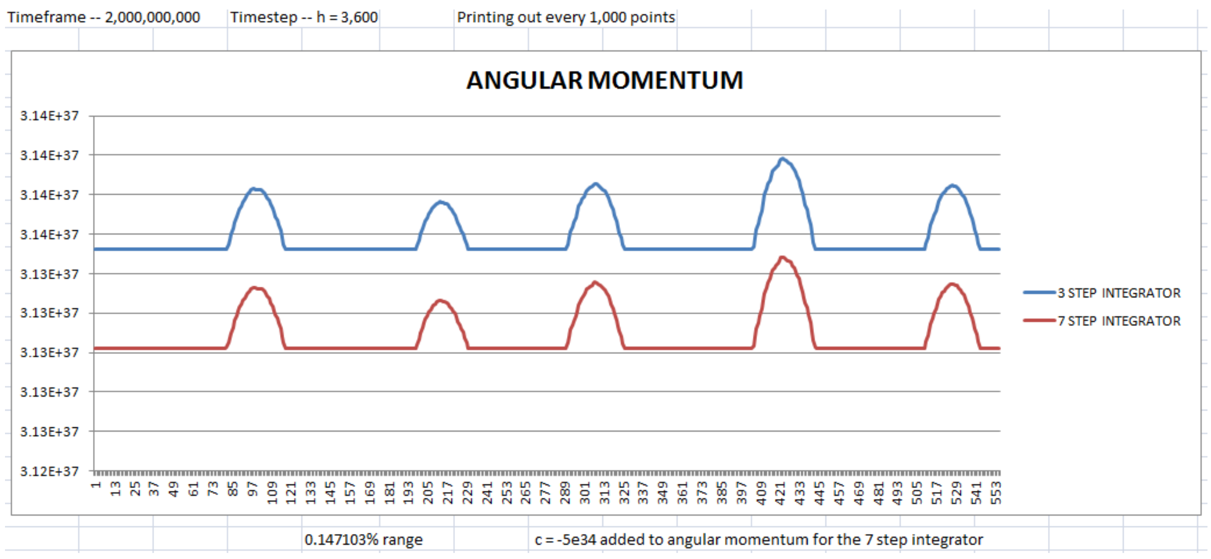
$$\vec{L} = \vec{r}\, x\, \vec{p} \quad \rightarrow \quad \|\vec{L}\| = \sqrt{\|\vec{r}\|^2\|\vec{p}\|^2(\sin\theta)^2} \quad = \quad \sqrt{\|\vec{r}\|^2\|\vec{p}\|^2 - \|\vec{r}\|^2\|\vec{p}\|^2(\cos\theta)^2}$$

....which is the formula for the scalar product, giving:

$$\|\vec{L}\| = \sqrt{\|\vec{r}\|^2\|\vec{p}\|^2 - (r_x p_x + r_y p_y + r_z p_z)^2}$$

....for 3 dimensions.

Plotting the results I get:


Timeframe -- 2,000,000,000 | Timestep -- h = 3,600 | Printing out every 1,000 points

0.147103% range | c = -5e34 added to angular momentum for the 7 step integrator

This uses data from *data.txt*[9] which contains data for 10 bodies - the Sun, the Planets and Pluto - which explain the small jumps in angular momentum, as I am working with a nontrivial system.
However, note that the jumps only constitute an overall increase of *0.147103%* - which proves that angular momentum is indeed conserved for the 3 and 7 step integrator.

[9] See Section 10.4

4.3 Verification of Kepler's 3 Laws of Planetary Motion

Another test for the code is the ability to replicate the results for Kepler's 3 Laws of Planetary Motion, which are[10]:

I. The orbit of a planet is an ellipse with the Sun at one of the two foci.
II. A line segment joining a planet and the Sun sweeps out equal areas during equal intervals of time.
III. The square of the orbital period of a planet is directly proportional to the cube of the semi major axis of its orbit.

I directly tested the second and third law, leaving visual identification to informally confirm the first.

Note: I only test this code for the 7 step integrator, because if the 7 step integrator replicates Kepler's Laws, then so will the 3 step integrator.

The C++ code for Kepler's 2nd Law is:

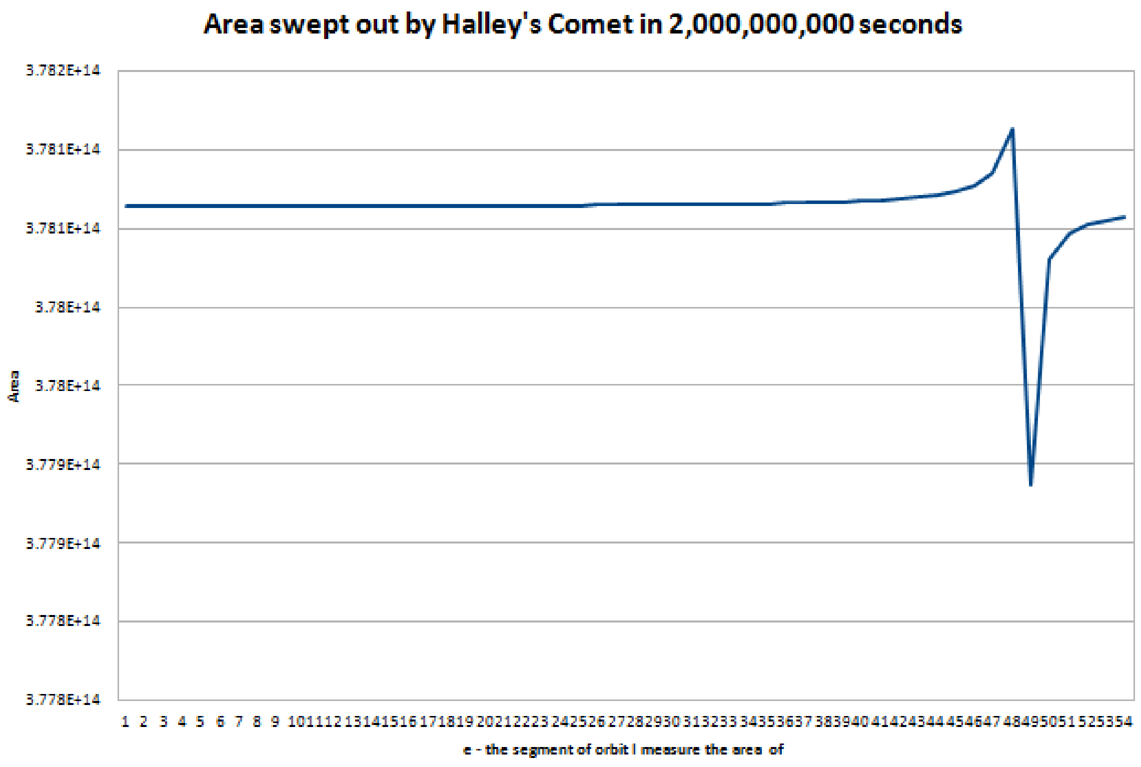
```
int p = 0;
double l[80], theta[80], area;

void law_2(double t, double b[][3], double c[2][3], int g)
{
    area = 0;
    if(t >= (0 + 25000000*g) && t <= (144000 + 25000000*g))
    {
        l[p] = dist(c,0,1);
        theta[p] = abs(abs(atan(abs(b[1][1]-b[0][1])/abs(b[1][0]-b[0][0]))) -
abs(atan(abs(c[1][1]-c[0][1])/abs(c[1][0]-c[0][0]))));
        ++p;
    }
    if(p == 40)
    {
        for(int k = 0; k<40; ++k)
        {   area += 0.5*l[k]*l[k]*theta[k];
        }
        output<<fixed<<setprecision(25)<<area<<endl;
        ++p;
    }
}

.....

int main()
{
    while(time[i] < 1300000000)
    {
        for(int k = 0; k<n; ++k)
        {
            for(int j = 0; j<3; ++j)
            {   c[k][j] = xb[k][j];
            }
        }

         .....
```

[10] http://en.wikipedia.org/wiki/Kepler's_laws_of_planetary_motion

```
        update(xb, vb, h, n, 0, 0);
        cc_acc(xb, ab, m, n);
        update(vb, ab, h, n, 1, 0);
        update(xb, vb, h, n, 0, 1);
        cc_acc(xb, ab, m, n);
        update(vb, ab, h, n, 1, 1);
        update(xb, vb, h, n, 0, 1);
        cc_acc(xb, ab, m, n);
        update(vb, ab, h, n, 1, 0);
        update(xb, vb, h, n, 0, 0);

          .....

        law_2(time[i], xb, c, e);

          .....
    }
}
```

Where the array *b* is the positions of the bodies after the timestep, *c* is the positions of the bodies before the timestep and *e* indicates which segment of the orbit to measure the area of.
In order to fully test this code, I chose to test the code with data about Halley's Comet, due to the orbits high eccentricity[11].

Plotting this result I get:



[11] Eccentricity of 0.967 - http://en.wikipedia.org/wiki/Halley's_Comet

Note: The two peaks formed when $e = 46$ *to* $e = 51$ *are because the Comet reaches aphelion at this point, and thus its speed is quite large*[12]*. However, the difference between the smallest area value and the largest area value is only* 0.059778% *of the overall area, so this proves the code permits Kepler's 2nd Law.*

The C++ code for Kepler's 3rd Law is:

```
int pp = 0;
void period(double a[][3], double c[][3], double t, int i)
{
    if(a[i][0] > 0)
    {
        if(a[i][1] > 0 && c[i][1] <= 0
        {
            if(t != 0 && pp == 0)
            {
                output<<fixed<<setprecision(25)<<t<<endl;
                ++pp;
            }
        }
    }
}
```

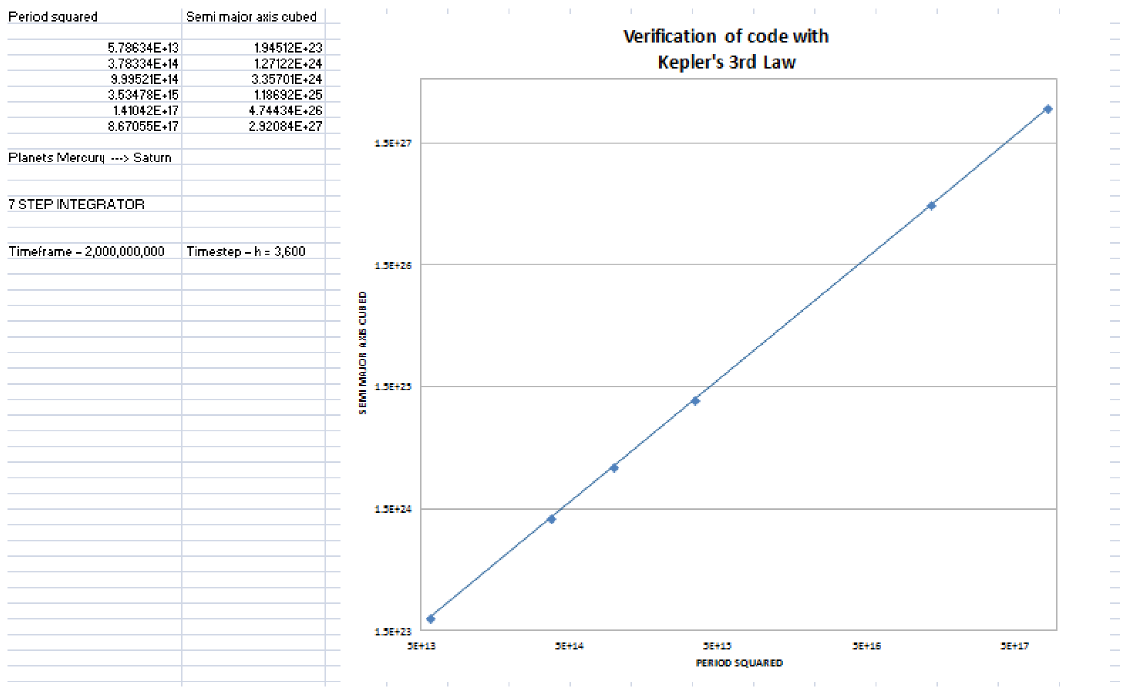
//Here, the array a is the positions of the bodies before the timestep, and the array c is after the //timestep

```
.....

int main()
{
    .....

    double max[n], min[n], len[n], cubed;
    for(int j = 0; j<n; ++j)
    {
        max[j] = 0;
        min[j] = 1e50;
    }

    while(time[i] < 1300000000)
    {
        for(int u = 1; u<n; ++u)                          //1
        {
            if(dist(xb,0,u) < min[u])
            {   min[u] = dist(xb,0,u);
            }
            if(dist(xb,0,u) > max[u])
            {   max[u] = dist(xb,0,u);
            }
        }

          .....
    }

    for(int k = 1; k<n; ++k)                              //2
    {
        len[k] = (max[k] + min[k])/2;
        cubed = (len[k]*len[k]*len[k]);
        output2<<fixed<<setprecision(25)<<cubed<<endl;
    }
    return 0;
}
```

[12] http://en.wikipedia.org/wiki/Halley's_Comet

This code uses the routine *period* to calculate the period of any one planet (body *i*), and uses 1 & 2 to calculate the semi-major axis for all of the planets.

Plotting the results, I get:



Period squared	Semi major axis cubed
5.78634E+13	1.94512E+23
3.78334E+14	1.27122E+24
9.99521E+14	3.35701E+24
3.53478E+15	1.18692E+25
1.41042E+17	4.74434E+26
8.67055E+17	2.92084E+27
Planets Mercury ---> Saturn	
7 STEP INTEGRATOR	
Timeframe – 2,000,000,000	Timestep – h = 3,600

Note: I use a logarithmic scale on the x and y axes in order to see all of the data points.

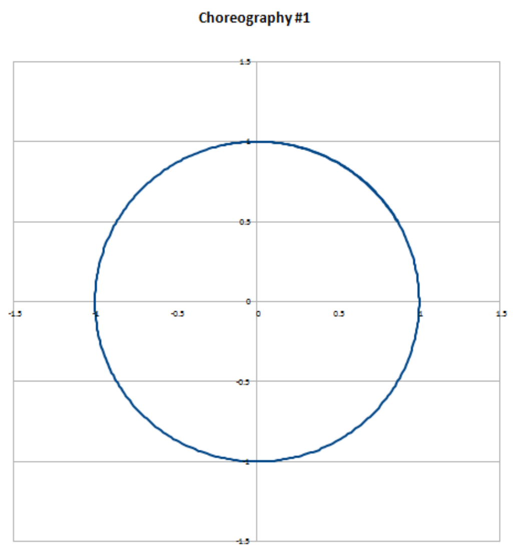
The straight line verifies that the code permits Kepler's 3rd Law.

4.4 Choreographies

Choreographies are periodic solutions to the analytic equations of Newton's Law of Motion for *n* bodies, and thus provide basis for an accuracy test for the 7 step integrator. If the 7 step integrator can produce an accurate trajectory for any choreography, this implies it is indeed a good approximation to the actual solutions to the analytic equations. Since the choreographies are exact solutions, this means they are quite fickle and small errors which build up naturally over time (due to the fact I am not performing actual integration) skew the results, so I run the choreographies for a short time period.

4.4.1 Choreography #1

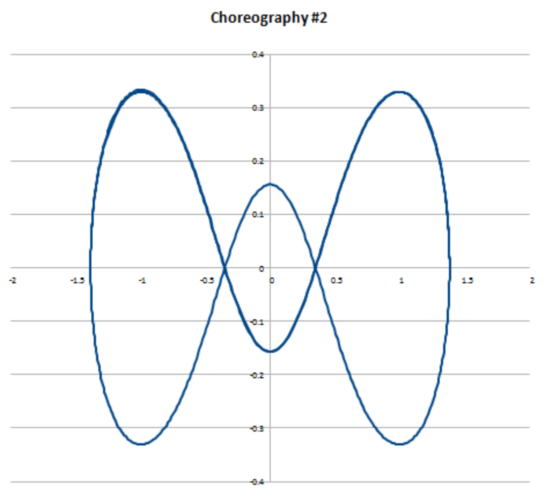
I ran this choreography for 10 seconds, with a timestep $h = 0.00005$ seconds[13]. This is one of the simplest choreographies, and contains 3 bodies which form a rotating equilateral triangle. Plotting the trajectory of one body gives:



....which is correct.

4.4.2 Choreography #2

I ran this choreography for 10 seconds with a timestep $h = 0.00000125$ seconds[14]. This choreography contains 4 bodies which orbit each other in a 'bow tie' fashion.
Plotting the trajectory of one of the bodies gives:

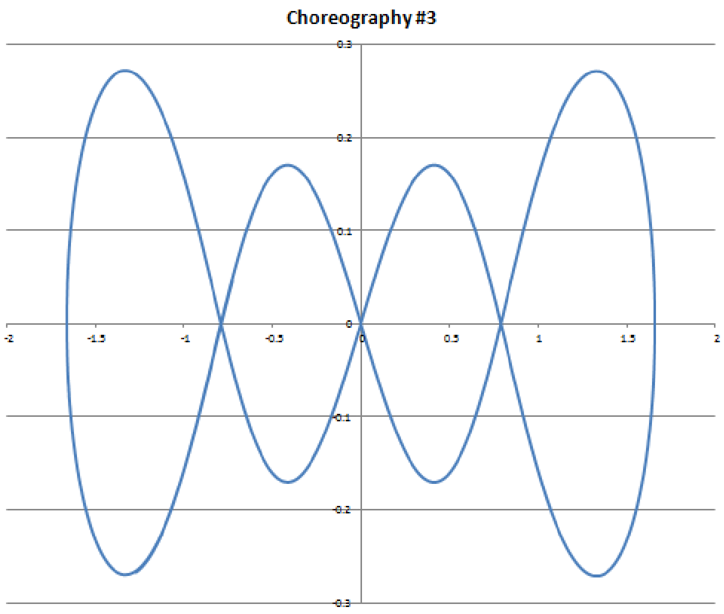


....which is correct.

[13] See Section 9 & 10.4
[14] See footnote 13

4.4.3 Choreography #3

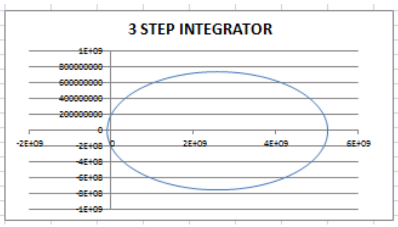
I ran this choreography for 6.75 seconds with a timestep $h = 0.000000675$ seconds[15]. This choreography contains 5 bodies which orbit each other in a 'linking' fashion.
Plotting the trajectory of one of the bodies gives:

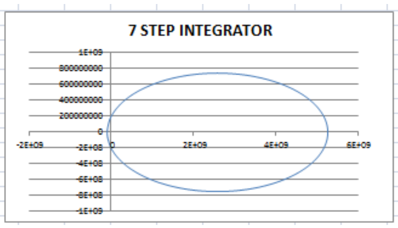


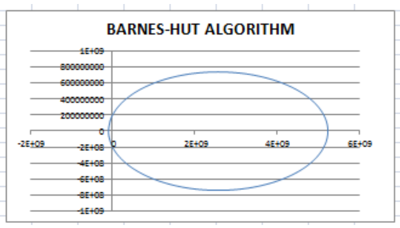
....which is correct. Since the 7 step integrator (and thus the 3 step integrator) successfully models these three choreographies this would imply that, not only is the code correctly calculating trajectories but it is also highly accurate.

4.5 Comparing the codes with each other

In order to determine how the accuracy of the codes change, going from 3 step integrator to 7 step integrator to the Barnes-Hut Algorithm, I compared the result using data which plots the trajectory of Halley's Comet, again because of the comets high eccentricity.







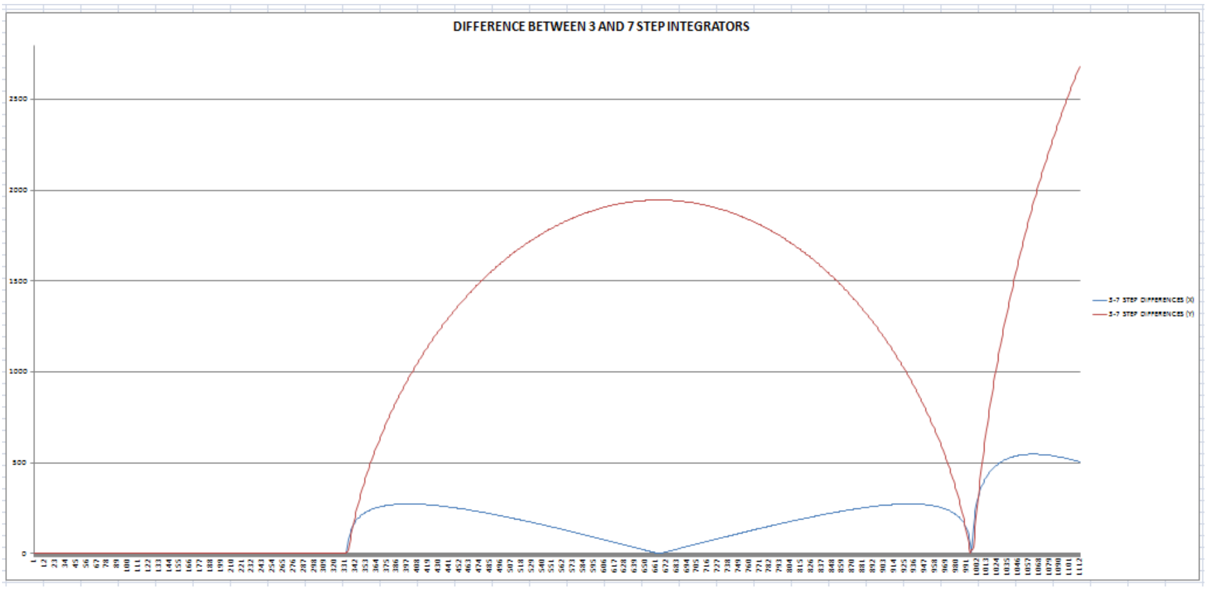
[15] See footnote 13

First, a direct visual comparison (see above) shows that all three codes correctly model the trajectory of Halley's Comet as it completes 1.5 - 2 orbits around the Sun (which was initially placed at the origin). However, a more direct comparison is needed.

Note: The above graphs are scaled to fit the entire orbit on the page, and thus are not reflective of the eccentricity of Halley's Comet's orbit.

4.5.1 Comparing the 3 step integrator with the 7 step integrator

I prepared the system[16] identically for the 3 step integrator and the 7 step integrator, using data from Halley's Comet with a timeframe of 4,000,000,000 seconds (*~126.75 years, between 1.5 and 2 orbits*) and a timestep of $h = 3{,}600$ seconds, and printed out every 1,000 points. In the data for the x direction, I took the difference between a point from the 3 step integrator and the corresponding point from the 7 step integrator, and repeated this for all of the points. I repeated this for the data in the y direction, and plotted the result:



I drew multiple conclusions from this graph:

I. There are large differences (especially in the y direction) between the 3 step integrator and the 7 step integrator - at one stage approx. 2,000 km - however this is still a small *(<<1%)* change in the range of the data
II. The accuracy of the algorithms appear to be dependant of the size of the force and the timescale on which the forces act;

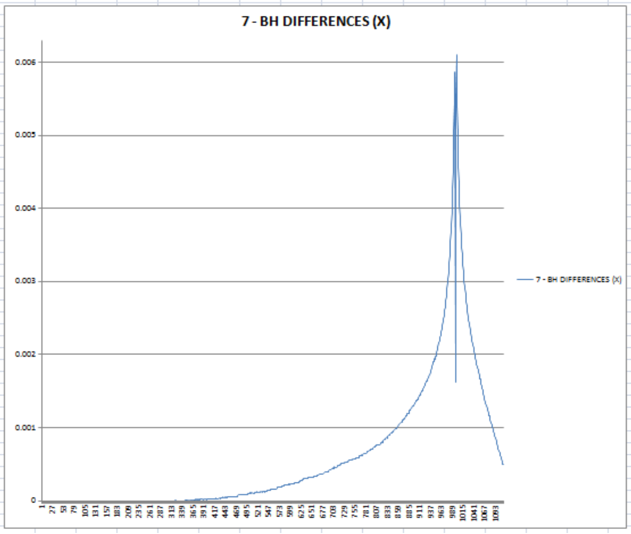
[16] See Section 10.4

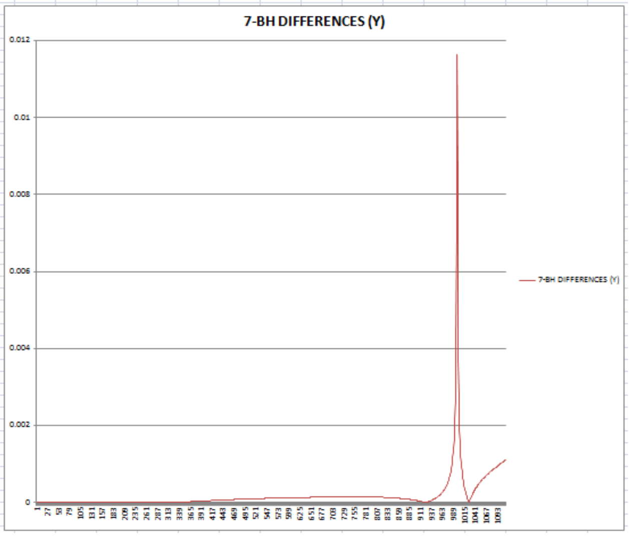
a. The difference in the *x* and *y* directions increase rapidly at point *~330*, which corresponds to when the Comet reaches perihelion, *i.e.* when the forces on the comet are the largest and when the Comet is moving its fastest.
b. The difference in the *y* direction starts to decrease at point *~660*, which corresponds to the Comet reaching aphelion, *i.e.* when the forces of the comet are smallest and when the Comet is moving its slowest.
c. The difference in the *x* direction decreases rapidly and reaches 0 at point *~660* (for the same reasons as *b*) and then increases again (for the same reasons as *a*)
d. The second peak is at point *~990*, which correspond to the comet reaching aphelion again.

III. These peaks and dips can be made much smaller by;
a. Using a smaller timestep (however this is not practical for simulations with a larger timeframe)
b. Using a variable timestep (proportional to the force on the comet). Using a variable timestep means a smaller timestep is used at (and approaching) aphelion (when the forces are largest) and a larger timestep at (and approaching) perihelion (when the forces are smallest)[17].

4.5.2 Comparing the 7 step integrator with the Barnes - Hut Algorithm

I used the same initial conditions as above, and plotted the result:





I draw multiple conclusions from these graphs:

I. Since the Barnes-Hut algorithm uses calculations nearly identical to the particle-particle method of the 7 step integrator (and since it uses a 7 step integrator) I expected the differences to be small, if not 0.

[17] See Section 6.3

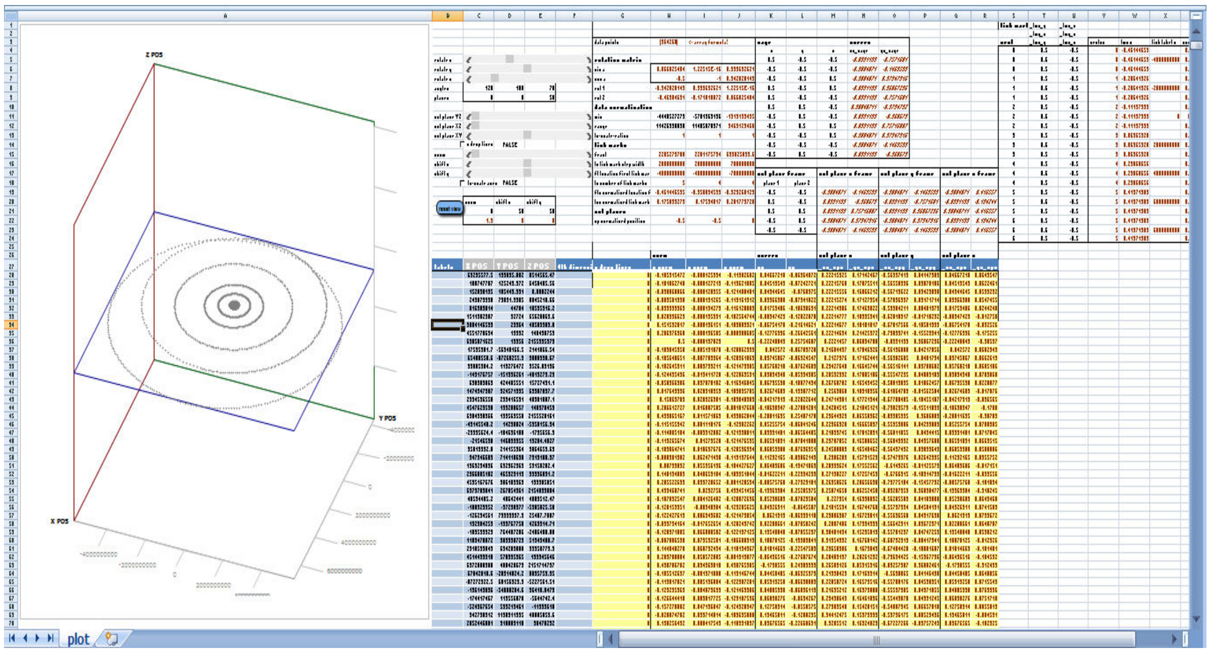
II. The differences are very small - in the *x* direction, the largest difference is *~0.006* km, and in the y direction the largest difference is *~0.012* km. I believe the errors come from two sources:
   a. These differences are small enough to be attributed to rounding errors from the additional calculations (centre of mass, etc) that the Barnes-Hut Algorithm must compute
   b. These errors are at point *~990*, which is when the Comet reaches its second aphelion, and I explained the reasoning behind the aphelion errors above.

From this, I conclude the Barnes-Hut Algorithm I wrote is fully functioning. However, in order to fully test its capabilities with *>10,000* bodies, a more powerful computer than mine is needed.

4.6 Creating a model of the Solar System

As a final test, I inputted data for the Sun, the planets of the Solar system and Pluto into the 7 step integrator algorithm and (printing every 1,000 points)[18] graphed the result in 3D (using an external macro[19]).

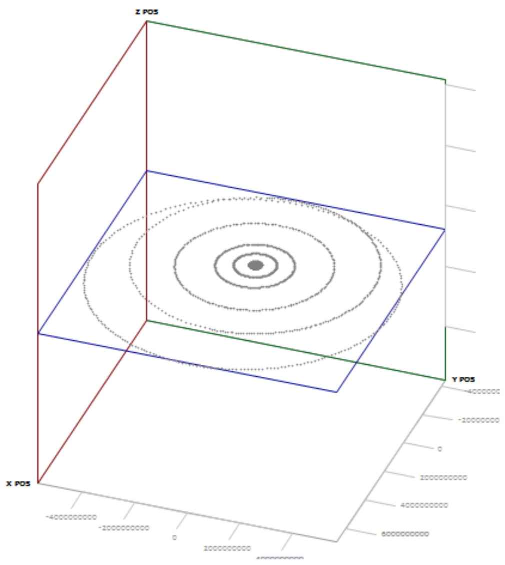
Plotting the result:



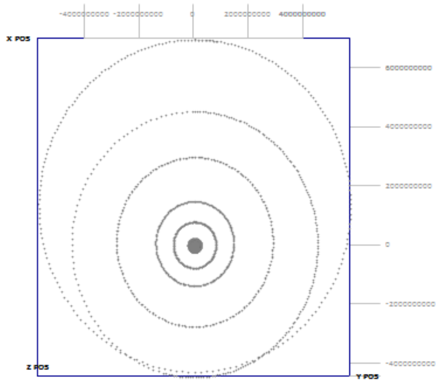
[18] See Section 10.4 for planetary data -- timeframe of 8,000,000,000 seconds & timestep here of 3,600 seconds

[19] See Section 9

A closer view of the graph (below):



A view of the elevation (below):

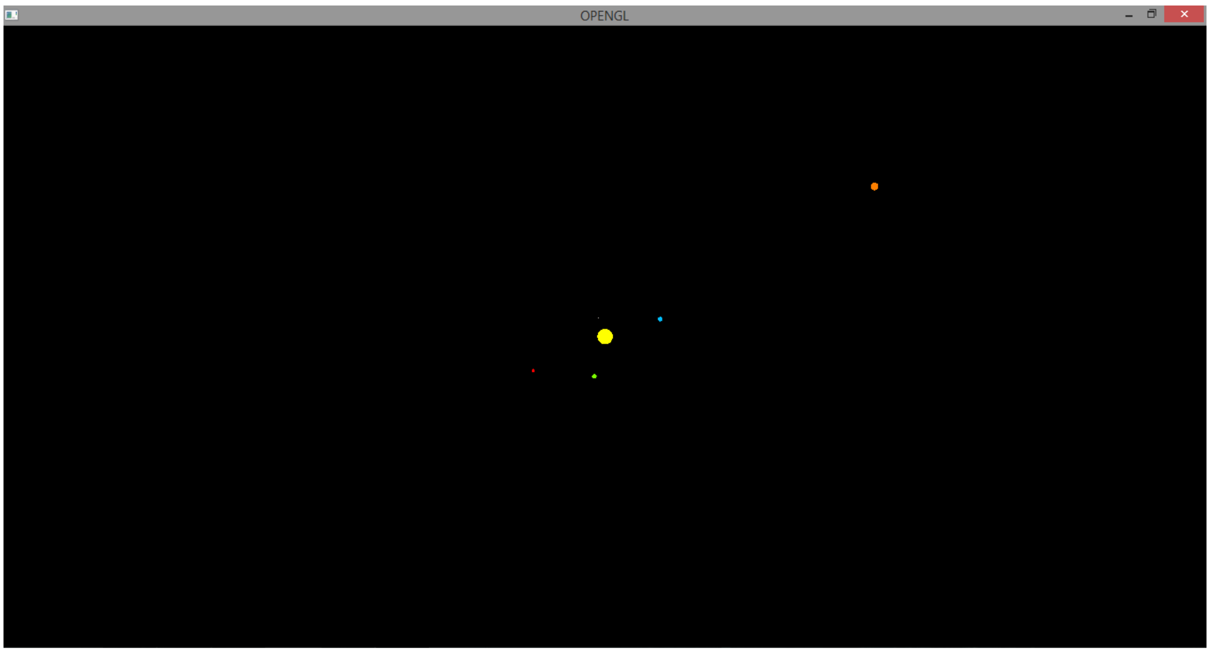


Unfortunately, at this scale the Inner Solar System cannot be seen clearly.
I believe this to be an accurate model of the solar system, because Pluto's eccentricity is clearly seen, as expected, and Pluto's orbit appears to intersect with that of Neptune's, which again is to be expected[20].

[20] http://en.wikipedia.org/wiki/Pluto#Relationship_with_Neptune

5. Animation

I animated the 7 step integrator algorithm and the Barnes - Hut Algorithm code using Microsoft Visual Studio 2013 and OpenGL (which was included on my laptop), and also using the header files *freeglut* and *glew*[21]. A full copy of the animation code of the 7 step integrator is printed at the end[22], and the Barnes - Hut Code is quite similar.
Below is a screenshot of Visual Studio running an animation which creates the trajectories of the Inner Solar System and Jupiter around the Sun:



I will not elaborate on the OpenGL code, because all of it is standard to creating an OpenGL project, and all of the code to do with creating an OpenGL window, making circles, etc, is unoriginal[19].
In addition, I found the Barnes-Hut Algorithm to run extremely slow in this scenario, because of the large amount of additional, wasted work (categorising nodes, creating a quadtree, etc) that the program needed to do, when the normal particle-particle methods are sufficient here (for a small number of bodies).

[21] See Section 10.4 for details on the files' location
[22] See Section 10.5.4

6. Future modifications

There are a number of additional modifications to the code which can be made:

6.1 Collision evasion

An additional routine to add to the codes would be collision detection & evasion. I have written the code for a program like this, however I didn't have adequate time to test, plot and animate the results.

The C++ code is:

```
void collision(double a[][2], double vb[][2], double b[], int n)
{
    double d, chk[n];
    for(int t = 0; t<n; ++t)
    {
        chk[t] = 0;
    }
    for(int i = 0; i<n; ++i)
    {
        for(int j = 0; j<n; ++j)
        {
            if(j > i)
            {
                d = b[i] + b[j] + 0.1;
                if( dist(a,i,j) <= d )
                {
                    if(chk[i] == 0)
                    {
                        if(vb[i][0] == 0 )
                        {
                            vb[i][1] = -vb[i][1];
                            chk[i] = 1;
                        }
                        else
                        {
                            vb[i][0] = -vb[i][0];
                            chk[i] = 1;
                        }
                    }


                    if(chk[j] == 0)
                    {
                        if(vb[j][0] == 0 )
                        {
                            vb[j][1] = -vb[j][1];
                            chk[j] = 1;
                        }
                        else
                        {
                            vb[j][0] = -vb[j][0];
                            chk[j] = 1;
                        }
                    }
                }
            }
        }
    }
}
```

For a body *i*, this code calculates the Euclidean distance between *i* and body *j*, and compares this answer to the sum of the respective radii (array *b*) plus a modifiable constant. If the distance is less than this value, the code proceeds to check if the velocity array *vb* has been changed before - using the array *chk*. If the velocity array of body *i* has not been modified, the code inverts the sign of the velocity in the *x* or *y* directions, and changes *chk*[*i*] to 1, ensuring the velocity of body *i* will not been changed again, in the same calling of the routine. If I had more time, I would also modify this code to be able to run in 3 dimensions, instead of running in just 2.

Also, to ensure (for testing purposes) that particles don't stray too far, I created a routine *box* which keeps the particles in a circle of any radius.

The C++ code is:

```
void box(double a[][2], double b[][2], int n)
{
    for(int i = 0; i<n; ++i)
    {
        if(norm(a,i) > 2)         //2 is an example radius
        {
            for(int j = 0; j<2; ++j)
            {   b[i][j] = -b[i][j];
            }
        }
    }
}
```

Where this routine calls the function *norm*, which returns the norm of *a* - the position array.

6.2 The 3D Barnes - Hut Algorithm

Another modification that could be made is changing the quadtree (for 2 dimensions) into an octree (for 3 dimensions), thus allowing the code to work in 3 dimensions instead of the current two. The *force* routine would remain quite similar to what it is now, however the *treebuild* routine would have to be modified drastically to include another dimension. The class *Node* would also have to be heavily modified, for the same reasons.

6.3 Variable Timesteps

Another modification (mentioned previously) is to introduce variable timesteps which increase or decrease proportional to the size of the force on a body. This would speed up the computation time and possibly allow me to animate the Barnes - Hut Algorithm data.

6.4 A faster code

The 7 step integrator is proportional to $O(n^2)$, meaning for n bodies the code must do $\sim n^2$ calculations. The Barnes - Hut Algorithm is proportional to $O(nlog(n))$ which implies that, for $n>>1$, it runs faster than the 7 step integrator. A future protraction of this project is to possibly create a faster algorithm for solving Newton's Laws of Motion, with the 'ultimate' code being proportional to $O(n)$.

7. Future applications

The future applications of this code are endless - a more modern application would perhaps be the use of a 'Dyson Swarm[23]', used to capture a significant percentage of the Sun's output. The Barnes - Hut Algorithm could be used to map globular clusters, or even whole galaxies. Another possibility is, with enough computer power, it should be possible to map the future collision between the Milky Way and Andromeda Galaxies, due in ~3.75 billion years[24].

8. Thanks and acknowledgement

Thanks to the TCD School of Mathematics for providing the resources to allow me to do this project, thanks to Dr. Mike Peardon for being my supervisor for the duration of the project, giving me direction on what tests the codes must pass, and giving me literature and sources for information on the n-Body Problem, and thanks to Robert Flood who also researched the n-Body Problem and provided a comparison to my data.

9. Bibliography

Apart from footnotes throughout the paper, other special/significant sources include:

For data.txt[25]:

- http://en.wikipedia.org/wiki/Mercury_(planet)
- http://en.wikipedia.org/wiki/Venus
- http://en.wikipedia.org/wiki/Earth

[23] http://en.wikipedia.org/wiki/Dyson_sphere
[24] http://en.wikipedia.org/wiki/Andromeda_Galaxy
[25] See Section 10.4

- http://en.wikipedia.org/wiki/Mars
- http://en.wikipedia.org/wiki/Jupiter
- http://en.wikipedia.org/wiki/Saturn
- http://en.wikipedia.org/wiki/Uranus
- http://en.wikipedia.org/wiki/Neptune
- http://en.wikipedia.org/wiki/Pluto
- http://nssdc.gsfc.nasa.gov/planetary/factsheet/

For header files and OpenGL instruction[26]:

- http://cse.spsu.edu/jchastin/
- http://stackoverflow.com/questions/19730976/opengl-and-c-drawing-shape-using-circle-algorthim
- https://www.youtube.com/watch?v=-NcnJ1Q-x50

For Choreography data[27]:

- http://www.scholarpedia.org/article/N-body_choreographies
- 'New Families of Solutions in *N*-Body Problems' - Carles Simò

Others:

- 3D Excel Macro - http://www.doka.ch/Excel3Dscatterplot.htm
- 'The Feynman Lecture on Physics, Volume I' - Richard Feynman

[26] See Section 6
[27] See Section 10.4

10. All code used in project

10.1 The 3 step Leapfrog Integrator code

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <iomanip>
using namespace std;

#define G 6.6738480e-20

ofstream output("RESULTS");
ofstream output2("RESULTS2");
ofstream output_x("RESULTS_X");
ofstream output_y("RESULTS_Y");
ofstream output_z("RESULTS_Z");

double dist(double a[][3], int i, int j)
{   return sqrt((a[i][0] - a[j][0])*(a[i][0] - a[j][0])+(a[i][1] -
a[j][1])*(a[i][1] - a[j][1])+(a[i][2] - a[j][2])*(a[i][2] - a[j][2]));
}

void update(double a[][3], double b[][3], double h, int x, int q)
{
    for(int i = 0; i<x; ++i)
    {
        for(int j = 0; j<3; ++j)
        {
            if( q == 0 )
            {   a[i][j] = a[i][j] + b[i][j]*h/2;
            }
            else
            {   a[i][j] = a[i][j] + b[i][j]*h;
            }
        }
    }
}

void cc_acc(double a[][3], double b[][3], double m[], int x)
{
    for(int i = 0; i<x; ++i)
    {
        for(int j = 0; j<3; ++j)
        {   b[i][j] = 0;
        }
    }

    for(int i = 0; i<x; ++i)
    {
        for(int j = 0; j<x; ++j)
        {
            if(j != i)
            {
                for(int k = 0; k<3; ++k)
                {   b[i][k] -= (G*m[j]*(a[i][k] -
a[j][k]))/(dist(a,i,j)*dist(a,i,j)*dist(a,i,j));
                }
            }
        }
    }
}
```

```
int main()
{
    int n, count = 0;
    double h, time[7] = {0, 0, 0, 0, 0, 0, 0};
    cin>>n>>h;
    double xb[n][3], vb[n][3], ab[n][3], m[n], r[n];

    for(int i = 0; i<n; ++i)
    {
        cin>>xb[i][0]>>xb[i][1]>>xb[i][2];
        cin>>vb[i][0]>>vb[i][1]>>vb[i][2];
        cin>>m[i]>>r[i];
    }

    for(int i = 0; i<1; ++i)
    {
        while(time[i] < 2000000000)
        {
            update(xb, vb, h, n, 0);
            cc_acc(xb, ab, m, n);
            update(vb, ab, h, n, 1);
            update(xb, vb, h, n, 0);

            for(int j = 0; j<1; ++j)
            {
                output_x<<fixed<<setprecision(25)<<xb[j][0]<<endl;
                output_y<<fixed<<setprecision(25)<<xb[j][1]<<endl;
                output_z<<fixed<<setprecision(25)<<xb[j][2]<<endl;
            }

            time[i] += h;
            ++count;
        }
    }

    return 0;
}
```

10.2 The 7 step Leapfrog Integrator code

```
#include <iostream>
#include <cmath>
#include <fstream>
#include <iomanip>
using namespace std;

#define G 6.6738480e-20             // G = 1 for choreographies & collision

#define w 1.25992105
#define f 0.74007895

ofstream output("RESULTS");
ofstream output2("RESULTS2");
ofstream output_x("RESULTS_X");
ofstream output_y("RESULTS_Y");
ofstream output_z("RESULTS_Z");

double dist(double a[][3], int i, int j)
{   return sqrt((a[i][0] - a[j][0])*(a[i][0] - a[j][0])+(a[i][1] -
a[j][1])*(a[i][1] - a[j][1])+(a[i][2] - a[j][2])*(a[i][2] - a[j][2]));
}
```

```
double norm(double a[][3], int i)
{   return sqrt((a[i][0]*a[i][0]) + (a[i][1]*a[i][1]) + (a[i][2]*a[i][2]));
}

void update(double a[][3], double b[][3], double h, int x, int q, int r)
{
    for(int i = 0; i<x; ++i)
    {
        for(int j = 0; j<3; ++j)
        {
            if( q == 0 )
            {
                if( r == 0 )
                {   a[i][j] = a[i][j] + b[i][j]*(h/(2*f));
                }
                else
                {   a[i][j] = a[i][j] + b[i][j]*(1 - w)*(h/(2*f));
                }
            }
            else
            {
                if( r == 0 )
                {   a[i][j] = a[i][j] + b[i][j]*h/f;
                }
                else
                {   a[i][j] = a[i][j] - b[i][j]*h*(w/f);
                }
            }
        }
    }
}

void cc_acc(double a[][3], double b[][3], double m[], int x)
{
    for(int i = 0; i<x; ++i)
    {
        for(int j = 0; j<3; ++j)
        {   b[i][j] = 0;
        }
    }
    for(int i = 0; i<x; ++i)
    {
        for(int j = 0; j<x; ++j)
        {
            if(j != i)
            {
                for(int k = 0; k<3; ++k)
                {   b[i][k] -= (G*m[j]*(a[i][k] -
a[j][k]))/(dist(a,i,j)*dist(a,i,j)*dist(a,i,j));
                }
            }
        }
    }
}

void Hamiltonian(double a[][3], double b[][3], double m[], double h, int x)
{
    double E = 0, sum_p = 0, sum_k = 0;
    for(int j = 0; j<x; ++j)
    {
        for(int i = 0; i<x; ++i)
        {
            if(i < j)
            {   sum_p -= (G*m[i]*m[j])/dist(a,i,j);
            }
        }
        sum_k += 0.5*m[j]*((b[j][0]*b[j][0]) + (b[j][1]*b[j][1]) +
(b[j][2]*b[j][2]));
    }
    E = sum_k + sum_p + 1.97655688974617e29 + 4e21;
    output2<<fixed<<setprecision(25)<<h<<"  "<<E<<endl;
}

double ang_mo(double a[][3], double b[][3], double m[], int n)
{
    double count = 0;
    for(int i = 0; i<n; ++i)
    {
        count += sqrt((norm(a,i)*norm(a,i))*(norm(b,i)*norm(b,i)*m[i]*m[i]) -
((a[i][0]*m[i]*b[i][0] + a[i][1]*m[i]*b[i][1] +
a[i][2]*m[i]*b[i][2])*(a[i][0]*m[i]*b[i][0] + a[i][1]*m[i]*b[i][1] +
a[i][2]*m[i]*b[i][2])));
    }
    return count;
}

int p = 0;
double l[80], theta[80], area;

void law_2(double t, double b[][3], double c[2][3], int g)
{
    area = 0;
    if(t >= (0 + 25000000*g) && t <= (144000 + 25000000*g))
    {
        l[p] = dist(c,0,1);
        theta[p] = abs(abs(atan(abs(b[1][1]-b[0][1])/abs(b[1][0]-b[0][0]))) -
abs(atan(abs(c[1][1]-c[0][1])/abs(c[1][0]-c[0][0]))));
        ++p;
    }
    if(p == 40)
    {
        for(int k = 0; k<40; ++k)
        {   area += 0.5*l[k]*l[k]*theta[k];
        }
        output<<fixed<<setprecision(25)<<area<<endl;
        ++p;
    }
}

int pp = 0;
void period(double a[][3], double c[][3], double t, int i)
{
    if(a[i][0] > 0)
    {
        if(a[i][1] > 0 && c[i][1] <= 0)
        {
            if(t != 0 && pp == 0)
            {
                output<<fixed<<setprecision(25)<<t<<endl;
                ++pp;
            }
        }
    }
}

int main()
{
    int n, e = 53, cc = 0, count = 0;
    double h, time[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    cin>>n>>h;
    double xb[n][3], vb[n][3], ab[n][3], m[n], r[n], c[n][3], s[n][3];
    for(int i = 0; i<n; ++i)
    {
        cin>>xb[i][0]>>xb[i][1]>>xb[i][2];
        cin>>vb[i][0]>>vb[i][1]>>vb[i][2];
        cin>>m[i]>>r[i];
    }

    double max[n], min[n], len[n], cubed;

    for(int j = 0; j<n; ++j)
    {
        max[j] = 0;
        min[j] = 1e50;
    }

    for(int i = 0; i<1; ++i)
    {
        while(time[i] < 1300000000)
        {
            for(int k = 0; k<n; ++k)
            {
                for(int j = 0; j<3; ++j)
//Note: each of the areas is measured wrt the planet's own ecliptic plane
                {
                    s[k][j] = vb[k][j];
                    c[k][j] = xb[k][j];
                }
            }
            for(int u = 1; u<n; ++u)
            {
                if(dist(xb,0,u) < min[u])
                {   min[u] = dist(xb,0,u);
                }
                if(dist(xb,0,u) > max[u])
                {   max[u] = dist(xb,0,u);
                }
            }

            if(count%1000 == 0)
            {   output<<fixed<<setprecision(20)<<ang_mo(xb, vb, m, n)<<endl;
            }

            update(xb, vb, h, n, 0, 0);
            cc_acc(xb, ab, m, n);
            update(vb, ab, h, n, 1, 0);
            update(xb, vb, h, n, 0, 1);
            cc_acc(xb, ab, m, n);
            update(vb, ab, h, n, 1, 1);
            update(xb, vb, h, n, 0, 1);
            cc_acc(xb, ab, m, n);
            update(vb, ab, h, n, 1, 0);
            update(xb, vb, h, n, 0, 0);

            box(xb, vb, n);
            law_2(time[i], xb, c, e);
            period(xb, c, time[i], 6);
            Hamiltonian(xb, vb, m, time[i], n);

            if(count%1000 == 0)
            {
                for(int j = 1; j<2; ++j)
                {
                    output_x<<fixed<<setprecision(25)<<xb[j][0]<<endl;
                    output_y<<fixed<<setprecision(25)<<xb[j][1]<<endl;
                    output_z<<fixed<<setprecision(25)<<xb[j][2]<<endl;
                }
            }
            time[i] += h;
            ++count;
        }
    }
    for(int k = 1; k<n; ++k)
    {
        len[k] = (max[k] + min[k])/2;
        cubed = (len[k]*len[k]*len[k]);
        output2<<fixed<<setprecision(25)<<cubed<<endl;
    }
    return 0;
}
```

10.3 The Barnes - Hut Algorithm

Just the Barnes - Hut Algorithm --- no Hamiltonian calculations, Angular Momentum, etc:
Note: Some features of the code require C++11

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>
#include <tuple>
#include <cmath>
#include <set>
using namespace std;

#define G 6.6738480e-20

#define w 1.25992105
#define f 0.74007895

ofstream output("RESULTS");
ofstream output_x("RESULTS_X");
ofstream output_y("RESULTS_Y");

typedef class Node
{
    public:
        set < tuple < double, double, double > > element;        //Elements of node
        double cen_x, cen_y;                                      //Centre of node
        double com_x, com_y;                                      //COM of node
        double d;                                                //Dimension of node

        void einspn(double, double, double);                     //Modifies elements
        void calcen(double, double);                              //Modifies centre
        void calcom();                                            //Modifies COM
        void caldim(double);                                    //Modifies dimension

        int num();                                      //Outputs number of elements
        double dim();                                            //Outputs dimension
        double width(int);                                        //Outputs width
        double height(int);                                       //Outputs height
        double centre_x();                               //Outputs x coord of centre
        double centre_y();                               //Outputs y coord of centre
        double cencom_x();                                  //Outputs x coord of COM
        double cencom_y();                                  //Outputs y coord of COM
        double mass();                                        //Outputs mass of node
        double search(double, double, double);
        void clearnode();                                         //Resets node

} Node;
```

```
int Node :: num()
{   return element.size();
}

double Node :: mass()
{
    double mtl = 0;
    set < tuple < double, double, double > > :: iterator it1;
    for(it1 = element.begin(); it1 != element.end(); ++it1)
    {   mtl += get<0>(*it1);
    }
    return mtl;
}

double Node :: cencom_x()
{   return com_x;
}

double Node :: cencom_y()
{   return com_y;
}

double Node :: dim()
{   return d;
}

double Node :: width(int p)
{   return (cen_x + p*d/2);
}

double Node :: height(int p)
{   return (cen_y + p*d/2);
}

double Node :: centre_x()
{   return cen_x;
}

double Node :: centre_y()
{   return cen_y;
}

void Node :: calcom()
{
    double s1 = 0, s2 = 0, mt = 0;
    set < tuple < double, double, double > > :: iterator it;
    if(num() >= 1)
    {
        for(it = element.begin(); it != element.end(); ++it)
        {
            s1 += (get<1>(*it))*(get<0>(*it));
            s2 += (get<2>(*it))*(get<0>(*it));
            mt += get<0>(*it);
        }
        com_x = s1/mt;
        com_y = s2/mt;
    }
    else
    {   com_x = cen_x;
        com_y = cen_y;
    }
}

double Node :: search(double a, double b, double c)
//outputs whether an element is in the node or not (1 former, 0 latter)
{   return element.count(tuple < double, double, double > (a, b, c));
}
```

```
void Node :: einspn(double a, double b, double c)
{   element.insert(tuple < double, double, double > (a, b, c));
}

void Node :: calcen(double a, double b)
{   cen_x = a;
    cen_y = b;
}

void Node :: caldim(double a)
{   d = a;
}

void Node :: clearnode()
{   element.clear();
}

double dist(double x1, double y1, double x2, double y2)
{   return sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2));
}

void update(double a[][2], double b[][2], double h, int n, int q, int r)
{
    for(int i = 0; i<n; ++i)
    {
        for(int j = 0; j<2; ++j)
        {
            if( q == 0 )
            {
                if( r == 0 )
                {   a[i][j] = a[i][j] + b[i][j]*(h/(2*f));
                }
                else
                {   a[i][j] = a[i][j] + b[i][j]*(1 - w)*(h/(2*f));
                }
            }
            else
            {
                if( r == 0 )
                {   a[i][j] = a[i][j] + b[i][j]*h/f;
                }
                else
                {   a[i][j] = a[i][j] - b[i][j]*h*(w/f);
                }
            }
        }
    }
}

void cc_acc(Node ns[], double a[][2], double c[][2], double m[], int i, int p)
{
    c[i][0] -= (G*ns[p].mass()*(a[i][0] -
ns[p].cencom_x()))/(dist(a[i][0],a[i][1],ns[p].cencom_x(),ns[p].cencom_y())*dist(a[
i][0],a[i][1],ns[p].cencom_x(),ns[p].cencom_y())*dist(a[i][0],a[i][1],ns[p].cencom_
x(),ns[p].cencom_y()));
    c[i][1] -= (G*ns[p].mass()*(a[i][1] -
ns[p].cencom_y()))/(dist(a[i][0],a[i][1],ns[p].cencom_x(),ns[p].cencom_y())*dist(a[
i][0],a[i][1],ns[p].cencom_x(),ns[p].cencom_y())*dist(a[i][0],a[i][1],ns[p].cencom_
x(),ns[p].cencom_y()));
}


double maxm(double a[][2], int n)
{
    double max = 0;
    for(int i = 0; i<n; ++i)
    {
        if(max < abs(a[i][0]))
        {   max = abs(a[i][0]);
        }

        if(max < abs(a[i][1]))
        {   max = abs(a[i][1]);
        }
    }
    return max;
}

void online(Node ns[], double a[][2], int i, int p)
{
    if(ns[p].width(-1) == a[i][0] || ns[p].width(1) == a[i][0])
    {
        if(a[i][0] >= 0)
        {   a[i][0] -= 0.000000000000001;
        }
        else
        {   a[i][0] += 0.000000000000001;
        }
    }
    if(ns[p].height(-1) == a[i][1] || ns[p].height(1) == a[i][1])
    {
        if(a[i][1] >= 0)
        {   a[i][1] -= 0.000000000000001;
        }
        else
        {   a[i][1] += 0.000000000000001;
        }
    }
}

void addelements(Node ns[], double a[][2], double m[], int n, int p)
{
    for(int i = 0; i<n; ++i)
    {
        online(ns,a,i,p);
        if(ns[p].width(-1) < a[i][0] && a[i][0] < ns[p].width(1) && ns[p].height(-
1) < a[i][1] && a[i][1] < ns[p].height(1))
        {   ns[p].einspn(m[i], a[i][0], a[i][1]);
        }
    }
}

set <int> s;
int checkpos(int q)
{
    if(s.count(q) > 0)
    {   checkpos(q+1);
    }
    else
    {   s.insert(q);
        return q;
    }
}

void treebuild(Node ns[], double a[][2], double m[], int n, int p)
{
    double temp;
    int aa, bb, cc, dd;
    addelements(ns,a,m,n,p);
    if(ns[p].num() > 1)
    {
        temp = ns[p].dim();
        p += 4;
        aa = checkpos(p+1);
        ns[aa].calcen((ns[p-4].centre_x() - temp/4), (ns[p-4].centre_y() +
temp/4));
        ns[aa].caldim(temp/2);
        treebuild(ns,a,m,n,aa);

        bb = checkpos(p+2);
        ns[bb].calcen((ns[p-4].centre_x() + temp/4), (ns[p-4].centre_y() +
temp/4));
        ns[bb].caldim(temp/2);
        treebuild(ns,a,m,n,bb);

        cc = checkpos(p+3);
        ns[cc].calcen((ns[p-4].centre_x() - temp/4), (ns[p-4].centre_y() -
temp/4));
        ns[cc].caldim(temp/2);
        treebuild(ns,a,m,n,cc);

        dd = checkpos(p+4);
        ns[dd].calcen((ns[p-4].centre_x() + temp/4), (ns[p-4].centre_y() -
temp/4));
        ns[dd].caldim(temp/2);
        treebuild(ns,a,m,n,dd);
    }
}

vector <int> nodefind(Node ns[], double a[][2], double m[], int j)
{
    vector <int> list;
    list.clear();
    for(int i = 0; i<50; ++i)
    {
        if(ns[i].search(m[j], a[j][0], a[j][1]) > 0)
        {   list.push_back(i);
        }
    }
    return list;
//returns a list of all the nodes (shallow to deep) that body j is in
}

bool theta(Node ns[], double a[][2], int i, int p)
{
    double thet, distance;
    distance = dist(a[i][0], a[i][1], ns[p].cencom_x(), ns[p].cencom_y());
    thet = ns[p].dim()/distance;
    if(ns[p].num() == 1)
    {   return true;
    }
    else
    {
        if(thet < 0.5)
        {   return true;
        }
        else
        {   return false;
        }
    }
}

void force(Node ns[], double a[][2], double c[][2], double m[], int n)
{
    vector <int> listi;
    vector <int> listj;
    vector <int> listself;

    for(int l = 0; l<n; ++l)
    {
        for(int m = 0; m<2; ++m)
        {   c[l][m] = 0;
        }
    }

    for(int u = 0; u<50; ++u)
    {   ns[u].calcom();
    }

    listi.clear();
    listj.clear();
    listself.clear();

    for(int i = 0; i<n; ++i)
    {
        listself = nodefind(ns, a, m, i);
        for(int j = 0; j<n; ++j)
        {
            if( j != i)
            {
                listj = nodefind(ns, a, m, j);
                for(int k = 0; k<listj.size(); ++k)
                {
                    if(theta(ns,a,i,listj[k]))
                    {
                        if(find(listself.begin(), listself.end(), listj[k]) ==
listself.end())
                        {
                            if(find(listi.begin(), listi.end(), listj[k]) !=
listi.end())
                            {   k += 100;
                            }
                            else
                            {   cc_acc(ns, a, c, m, i, listj[k]);
                                listi.push_back(listj[k]);
                                k += 100;
                            }
                        }
                    }
                }
            }
            listj.clear();
        }
        listi.clear();
        listself.clear();
    }
}

void reset(Node ns[])
{
    s.clear();
    for(int i = 0; i<50; ++i)
    {   ns[i].clearnode();
    }
}

void leapfrog(Node ns[], double a[][2], double b[][2], double c[][2], double m[],
double h, int n)
{
    reset(ns);
    ns[0].calcen(0,0);
    ns[0].caldim(2*maxm(a, n));
    treebuild(ns, a, m, n, 0);

    update(a, b, h, n, 0, 0);
    reset(ns);
    ns[0].calcen(0,0);
    ns[0].caldim(2*maxm(a, n));
    treebuild(ns, a, m, n, 0);
    force(ns, a, c, m, n);
    update(b, c, h, n, 1, 0);


    update(a, b, h, n, 0, 1);
    reset(ns);
    ns[0].calcen(0,0);
    ns[0].caldim(2*maxm(a, n));
    treebuild(ns, a, m, n, 0);
    force(ns, a, c, m, n);
    update(b, c, h, n, 1, 1);

    update(a, b, h, n, 0, 1);
    reset(ns);
    ns[0].calcen(0,0);
    ns[0].caldim(2*maxm(a, n));
    treebuild(ns, a, m, n, 0);
    force(ns, a, c, m, n);
    update(b, c, h, n, 1, 0);
    update(a, b, h, n, 0, 0);
}

int main()
{
    int n, count = 0;
    double h, time = 0;
    cin>>n>>h;
    double xb[n][2], vb[n][2], ab[n][2], m[n];
    Node no[50];

    for(int i = 0; i<n; ++i)
    {
        cin>>xb[i][0]>>xb[i][1];
        cin>>vb[i][0]>>vb[i][1];
        cin>>m[i];
    }

    while(time < 1300000000)
    {
        leapfrog(no, xb, vb, ab, m, h, n);

        if(count%1000 == 0)
        {
            for(int j = 1; j<2; ++j)
            {
                output_x<<fixed<<setprecision(25)<<xb[j][0]<<endl;
                output_y<<fixed<<setprecision(25)<<xb[j][1]<<endl;
            }
        }
        time += h;
        ++count;
    }

    return 0;
}
```

10.4 Data files

The data files are read into the program in the order show in the codes above - see Section 9 for the sources of this data.

data.txt - Planetary data

```
10
7200

0 0 0
0 -1.4973498e-2 0
1.98855e30
696374.5

69295752.64 0 8514586.989
0 38.86 0
3.3022e23
2440

108747859.3 0 6450489.855
0 34.79 0
4.8676e24
6052

152098232 0 0
0 29.29 0
5.9726e24
6371

249079404.3 0 8045219.103
0 21.97 0
6.4185e23
3389.6

816309015.5 0 18595916.23
0 12.44 0
1.8986e27
69914

1511902387 0 65620863.64
0 9.09 0
5.6846e26
58235

3004146593 0 40509303.77
0 6.49 0
8.6810e25
25365.5

4551778634 0 140498759.3
0 5.37 0
1.0243e26
24631.5

6985871625 0 2155995973
0 3.71 0
1.305e22
1189
```

data2.txt - Halley's Comet

```
2
7200

0 0
0 1.106361579e-16
1.98855e30


5.250885262e9 0
0 -1
2.2e14
```

data4.txt - Choreography #2

```
4
0.0000125

1.382857 0 0
0 0.584873 0
1
0.01

0 0.157030 0
1.871935 0 0
1
0.01

-1.382857 0 0
0 -0.584873 0
1
0.01

0 -0.157030 0
-1.871935 0 0
1
0.01
```

data3.txt - Choreography #1

```
3
0.000025

0.866025403 0.5 0
-0.379917842 0.658037006 0
1
1

-0.866025403 0.5 0
-0.379917842 -0.658037006
0
1
1

0 -1 0
0.759835685 0 0
1
1
```

data5.txt - Choreography #3

```
5
0.000000675

-1.268608 -0.267651 0
1.271564 0.168645 0
1
0.1

-1.268608 0.267651 0
-1.271564 0.168645 0
1
0.1

0.439775 -0.169717 0
1.822785 0.128248 0
1
0.1

0.439775 0.169717 0
-1.822785 0.128248 0
1
0.1

1.657666 0 0
0 -0.593786 0
1
0.1
```

10.5 Useful programs

10.5.1 Max/min of data set

```
#include <iostream>
using namespace std;

int main()
{
    double x1, x2, min = 1e50, max = 0;
    bool finished = false;
    while(!finished)
    {
        if(cin.eof())
        {   finished = true;
        }
        else
        {
            cin>>x1>>x2;
            if(x2 > max)
            {   max = x2;
            }
            if(x2 < min)
            {   min = x2;
            }
        }
    }
    cout<<"MAX:  "<<max<<"  MIN:  "<<min<<endl;
}
```

10.5.2 Log (base 10) of data set

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;

ofstream output("RESULTS");

int main()
{
    double x1, x2;
    bool finished;
    finished  = false;
    while(!finished)
    {
        if(cin.eof())
        {
            finished  = true;
        }
         else
        {
            cin>>x2;
            output<<fixed<<setprecision(10)<<log(x2)<<endl;
        }
    }
    return 0;
}
```

10.5.3 Sorting the data set

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <iomanip>
#include <vector>
#include <algorithm>
using namespace std;

ofstream output("RESULTS");

int main()
{
    double t, h, diff;
    vector < pair <double, double> > v;
    bool finished = false;

    while(!finished)
    {
        if(cin.eof())
        {   finished = true;
        }
        else
        {   cin>>t>>h;
            v.push_back( pair <double, double> (t, h) );
        }
    }

    sort ( v.begin(), v.end() );

    for( int i = 0; i<v.size(); ++i )
    {
        if( (i+1)%3 != 0 )
        {
            if( (i+1)%2 == 0 )
            {
                diff = abs(v[i].second - v[i-1].second);
                if( v[i].first == 14400000 )
                {   output<<fixed<<setprecision(20)<<v[i].first<<" "<<diff<<endl;
                }
            }
        }
    }

    return 0;
}
```

10.5.4 Animating the result

This code animates two bodies using a 7 step integrator -- any number of bodies can be animated, however I was limited by computation power and screen size. This algorithm runs in 2 dimensions. The C++ code is:

Note: The header files freeglut.h and glew.h must be downloaded for this program.

```
#include <GL\glew.h>
#include <GL\freeglut.h>
#include <Windows.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
using namespace std;

#define G 6.6738480e-20
#define fir 1.25992105
#define sek 0.74007895

void changeViewport(int w, int h)
{	glViewport(0, 0, w, h);
}

void render()
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	glutSwapBuffers();
}

void Timer(int iUnused)
{
	glutPostRedisplay();
	glutTimerFunc(1, Timer, 0);
}

void circle(float x, float y, float r, int segments)
{
	glBegin(GL_TRIANGLE_FAN);
	glVertex2f(x, y);
	for (int n = 0; n <= segments; ++n) {
		float const t = 2 * 3.14159265*(float)n / (float)segments;
			glVertex2f(x + sin(t)*r, y + cos(t)*r);
	}
	glEnd();
}
double dist(double a[][2], int i, int j)
{
	return sqrt((a[i][0] - a[j][0])*(a[i][0] - a[j][0]) + (a[i][1] -
a[j][1])*(a[i][1] - a[j][1]));
}

void update(double a[][2], double b[][2], double h, int n, int q, int r)
{
    for (int i = 0; i<n; ++i)
    {
            for (int j = 0; j<2; ++j)
            {
                    if (q == 0)
                    {
                          if (r == 0)
                          {   a[i][j] = a[i][j] + b[i][j] * (h / (2 * sek));
                          }
                          else
                          {   a[i][j] = a[i][j] + b[i][j] * (1 - fir)*(h / (2 * sek));
                          }
                    }
                    else
                    {
                           if (r == 0)
                           {  a[i][j] = a[i][j] + b[i][j] * (h / sek);
                           }
                           else
                           {  a[i][j] = a[i][j] - b[i][j] * h *(fir / sek);
                           }
```

```
                  }
            }
      }
}

void cc_acc(double a[][2], double b[][2], double m[], int n)
{
        for (int i = 0; i<n; ++i)
        {
                for (int j = 0; j<2; ++j)
                {       b[i][j] = 0;
                }
        }
        for (int i = 0; i<n; ++i)
        {
                for (int j = 0; j<n; ++j)
                {
                        if (j != i)
                        {
                                for (int k = 0; k<2; ++k)
                                {
                                        b[i][k] -= (G*m[j] * (a[i][k] - a[j][k])) /
(dist(a, i, j)*dist(a, i, j)*dist(a, i, j));
                                }
                        }
                }
        }
}

void leapfrog(double a[][2], double b[][2], double c[][2], double m[], double h,
int n)
{
        update(a, b, h, n, 0, 0);
        cc_acc(a, c, m, n);
        update(b, c, h, n, 1, 0);
        update(a, b, h, n, 0, 1);
        cc_acc(a, c, m, n);
        update(b, c, h, n, 1, 1);
        update(a, b, h, n, 0, 1);
        cc_acc(a, c, m, n);
        update(b, c, h, n, 1, 0);
        update(a, b, h, n, 0, 0);
}

int ccnt = 0, n;
double hh;
double xb[2][2], vb[2][2], ab[2][2], m[2];

//Code must be modified here to allow for more bodies

void getpoints()
{
        cout << "Enter file name: ";
        ifstream myfile;
        char filename[50];
        cin.getline(filename, 50);
        myfile.open(filename);

        if (myfile.is_open())
        {
                while (!myfile.eof())
                {
                        myfile >> n >> hh;
                        for (int i = 0; i < n; ++i)
                        {
                                myfile >> xb[i][0] >> xb[i][1];
                                myfile >> vb[i][0] >> vb[i][1];
                                myfile >> m[i];
                        }
                }
        }
        else
        {       cout << "Error -- Cannot find file" << endl;
        }

        myfile.close();
        system("PAUSE");
}

void display()
{
        glClear(GL_COLOR_BUFFER_BIT);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        double w = glutGet(GLUT_WINDOW_WIDTH);
        double h = glutGet(GLUT_WINDOW_HEIGHT);
        double ar = w / h;
        glOrtho(-4 * ar, 4 * ar, -4, 4, -1, 1);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glColor3ub(255, 255, 0);
        circle((float)(xb[0][0] * 5 / 5.250885262e9), (float)(xb[0][1] * 5 /
5.250885262e9), 0.2, 20);

        glColor3ub(0, 191, 255);
        circle((float)(xb[1][0] * 5 / 5.250885262e9), (float)(xb[1][1] * 5 /
5.250885262e9), 0.05, 20);


//Code must be modified here to allow for more bodies


        glutSwapBuffers();

        if (ccnt == 0)
        {
                getpoints();
        }

        leapfrog(xb, vb, ab, m, hh, n);
        ++ccnt;
}

int main(int argc, char** argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
        glutInitWindowSize(800, 600);
        glutCreateWindow("OPENGL");
        glutReshapeFunc(changeViewport);
        glutDisplayFunc(render);
        glutDisplayFunc(display);
        Timer(0);

        GLenum err = glewInit();
        if (GLEW_OK != err) {
                fprintf(stderr, "GLEW error");
                return 1;
        }

        glutMainLoop();
        return 0;
}
```